PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Doctorado
en Informática

Software unit testing techniques: An empirical study

Diego Vallespir

Marzo 2012

Software unit testing techniques: an empirical study
Vallespir, Diego
ISSN 0797-6410
Tesis de Doctorado en Informática
Reporte Técnico RT 12-06
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, 5 de marzo de 2012

UNIVERSIDAD DE LA REPÚBLICA
URUGUAY

Tesis Doctoral

Software Unit Testing Techniques: An Empirical Study

Diego Vallespir

Director de Tesis: Juliana Herbert

Director de Estudios: Álvaro Tasistro

Programa de Doctorado en Informática

Marzo de 2012

PEDECIBA
Programa de Desarrollo de las Ciencias Básicas
Universidad de la República - Ministerio de Educación y Cultura - PNUD

Tesis presentada por Diego Vallespir para aspirar al Título de Doctor en Informática.

Vista y aprobada por Juliana Herbert.

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

El tribunal docente integrado por los abajo firmantes aprueba la Tesis de Investigación:

Software Unit
Testing Techniques:
An Empirical Study

Autor: MSc. Diego Vallespir
Directora de Tesis: PhD. Juliana Herbert
Director Académico: PhD. Álvaro Tasistro
Carrera: Doctorado en Informática - PEDECIBA
Calificación: Aprobado con Mención

TRIBUNAL

PhD. William Nichols (Revisor)	
PhD. Eduardo Miranda (Revisor)	
PhD. Claudia Pons	
PhD. Cristina Cornes	
PhD. Omar Viera	

Montevideo, 5 de marzo de 2012

To my family and
to the team.

Agradecimientos [Acknowledgements]

Quiero agradecer primero a las personas que trabajaron conmigo en los proyectos de investigación que fueron ejecutados durante el desarrollo de esta tesis: Apa, Cecilia; Ávila, Adriana; Bogado, Carmen; Camilloni, Lucía; De León, Stephanie; Grazioli, Fernanda; Herbert, Juliana; Marotta, Adriana; Marotta, Fernando; Moreno, Silvana; Nichols, William; Robaina, Rosana; Valverde, Carolina. Muchas gracias por todo el trabajo compartido y por el esfuerzo realizado.

También quiero agradecer al resto del Grupo de Ingeniería de Software de la UdelaR por el apoyo brindado. En este sentido agradezco sobre todo a Jorge Triñanes, con quien varias veces conversé sobre mi trabajo de tesis.

Agradezco a Luis Sierra por su ayuda con LaTeX; siempre que tuve dudas Luis estuvo ahí. A Lorena Ibaceta y a María Inés Imas por las traducciones al inglés. A Santiago por el pasaje del formato de los artículos publicados en la revista 1024 para que quedaran igual en inglés que en español.

A Juliana Herbert por ser la tutora y guía de este trabajo de tesis y a Álvaro Tasistro por ser mi director de estudios.

A Watts Humphrey por plantear la idea inicial del tema de tesis. Aprovecho para agradecerle a Watts todo lo que le ha dado a la ingeniería de software.

Agradezco mucho a mi familia: mi esposa, mis padres, mi hermana y mi cuñado. Sobre todo a Mari, por acompañar, apoyar y entender.

Resumen en Español

El tamaño y la complejidad del software aumentan cada año. A su vez, el software cada vez ocupa más lugares en nuestro mundo y en nuestra vida (casas inteligentes, autos, teléfonos, televisiones, bancos, oficinas, aviones, etc. contienen una importante porción de software). El desarrollo de software es una actividad creativa e intelectual realizada por seres humanos. Durante un proyecto de software es normal que el equipo de desarrollo cometa errores; esto se debe a la complejidad actual del software y debido a la naturaleza humana en si misma. Normalmente estos errores introducen defectos en el producto de software y cuando el software es utilizado esos defectos pueden causar fallas. La investigación de cómo construir software que no contenga defectos ha llevado a desarrollar, entre otras cosas, diversas y variadas técnicas de pruebas de software. El objetivo de estas técnicas es detectar defectos antes de que el producto sea usado por los usuarios finales. Desafortunadamente, es difícil elegir qué técnicas utilizar. Esto se debe a que no es suficiente el conocimiento actual que se tiene sobre la efectividad y el costo de usar cada técnica de pruebas. Una forma de aumentar este conocimiento es a través de experimentos controlados.

El objetivo general de esta tesis es contribuir al conocimiento de distintas técnicas de pruebas. Más específicamente, la intención es conducir una investigación empírica para conocer los resultados de usar distintas técnicas de pruebas; principalmente en lo que refiera a su efectividad y costo. Este objetivo general es dividido en dos objetivos particulares. El primero y principal es el estudio de las siguientes técnicas de pruebas: inspección de escritorio, partición en clases de equivalencia y análisis de valores límite, tablas de decisión, trayectorias linealmente independientes, cubrimiento de condición múltiple, cubrimiento de sentencias, y todos los usos. El segundo es el estudio de las técnicas de pruebas utilizadas en el Proceso Personal de Software (*Personal Software Process* - PSP).

Para conocer la efectividad y el costo de las 7 técnicas de pruebas mencionadas, realizamos dos experimentos controlados. Llamamos al primer experimento Experimento 2008 y al segundo Experimento 2009. Los nombres provienen de los años en los cuales comenzaron los experimentos.

En el análisis de resultados encontramos que la efectividad de todas las técnicas de pruebas fue baja. Ninguna de las técnicas tuvo más de un 35%

de efectividad. Además, la efectividad de las técnicas se reduce cuando estas fueron utilizadas en los programas de nuestro experimento en relación a cuando estas fueron usadas durante el entrenamiento. Esto seguramente se debe a que es más fácil encontrar defectos en programas triviales que encontrar defectos en programas complejos.

También realizamos algunos estudios iniciales usando el PSP. Analizamos el comportamiento de las técnicas de pruebas y el costo de encontrar y corregir defectos usando el PSP. Usamos datos de cursos del PSP que fueron tomados desde octubre de 2005 hasta enero de 2010. Estos cursos fueron dictados por el Software Engineering Institute (SEI) de la Universidad Carnegie Mellon o por socios (*partners*) del SEI.

En este caso encontramos que la revisión de diseño tiene una alta efectividad (más de un 50%) para remover defectos que fueron inyectados durante la fase de diseño detallado . También encontramos que, la revisión de código tienen una alta efectividad (más del 60%) para remover defectos inyectados durante la fase de codificación.

Finalmente, realizamos otros trabajos de investigación que complementan la línea de investigación central de esta tesis. Estos trabajos fueron muy diversos. Estudiamos diferentes propuestas de taxonomías de defectos que existen en la literatura, las evaluamos y las comparamos. También propusimos un marco de comparación de experimentos que buscan evaluar las técnicas de pruebas. Empaquetamos el Experimento 2008. Por último, también analizamos la calidad de los datos recolectados por los sujetos del Experimento 2008.

Durante esta tesis hemos hecho hincapié en usarla como un marco para la creación del Grupo de Investigación en Ingeniería de Software de nuestra Universidad. Esto implicó un esfuerzo extra importante pero, gratificante al mismo tiempo ya que consideramos que hemos avanzado en la consolidación de este grupo.

Abstract

The size and complexity of the software increase every year. Besides, it takes up more places in our world and in our life (intelligent houses, cars, telephones, televisions, banks, offices, aircraft, etc, contain an important portion of software). Software development is a creative and intellectual activity performed by human beings. During a software project it is normal for the development team to make mistakes, both because of the current complexity of the software and because of human nature itself. Normally these mistakes end up in defects in the software product and when the software is being executed these can cause failures. The search of how to develop software that does not contain defects has led to, among other things, the development of a varied number of testing techniques. These aim at the detection of defects before the product is used by the users. Unfortunately, it is difficult to choose which technique to use, since the knowledge currently available about the effectiveness and the cost of each of the testing techniques is not enough. A way of increasing that knowledge is through controlled experiments.

The general goal of this thesis is to contribute to the knowledge of different testing techniques. More specifically, we intend to conduct an empirical investigation of the results of using them, mainly in what refers to their effectiveness and cost. This general goal is divided into two particular ones. The first and principal is the study of the following testing techniques: desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, multiple condition coverage, sentence coverage and all uses. The second is the study of the Personal Software Process (PSP) testing techniques.

In order to know the effectiveness and cost of the 7 testing techniques mentioned we conducted two controlled experiments. We called the first one Experiment 2008 and the second one Experiment 2009. Their names come from the years in which we started the experiments.

In the analysis of the results we found that the effectiveness of all the testing techniques was low. None of the techniques was more than 35% effective. Besides, the effectiveness of the techniques decreased when they were used in the experiment programs in relation to when they were used in the training. This is certainly due to the fact that it is easier to find defects

in trivial programs than it is to find them in complex ones.

We also conducted some initial studies using the PSP. We analyzed the behavior of the testing techniques and the cost of finding and fixing defects in the PSP. We used data from PSP courses taught between October 2005 and January 2010. These courses were taught by the Software Engineering Institute (SEI) of the Carnegie Mellon University or by SEI partners.

In this case we found that the design review has a high effectiveness to remove the defects injected during the detailed design phase (more than 50%). On the other hand, the code review has high effectiveness to remove the defects injected during the code phase (more than 60%).

Finally in the thesis we did research work that complements the central research line. These works have been very diverse. Different defect taxonomies proposed in the literature were studied, evaluated and compared. A comparison frame of controlled experiments that intend to evaluate verification techniques was created. A packaging of the Experiment 2008 was done. The quality of the data collected by the subjects in the Experiment 2008 was evaluated.

During this thesis we have stressed its use as a frame for the creation of a Software Engineering Research Group in our University. This has involved an important effort, but it has been gratifying at the same time since we understand that we have been able to advance in the consolidation of this group.

Contents

Chapter 1

Introduction

The size and complexity of the software increase every year. Besides, it takes up more places in our world and in our life (intelligent houses, cars, telephones, televisions, banks, offices, aircraft, etc, contain an important portion of software). Software development is a creative and intellectual activity performed by human beings. During a software project it is normal for the development team to make mistakes, both because of the current complexity of the software and because of human nature itself. Normally these mistakes end up in defects in the software product and when the software is being executed these can cause failures. The search of how to develop software that does not contain defects has led to, among other things, the development of a varied number of testing techniques. These aim at the detection of defects before the product is used by the users. Unfortunately, it is difficult to choose which technique to use, since the knowledge currently available about the effectiveness and the cost of each of the testing techniques is not enough. A way of increasing that knowledge is through controlled experiments. The main topic of this thesis is the study, through controlled experiments, of the effectiveness and the cost of different unit testing techniques.

1.1 Motivation

Software development is performed by human teams and while it is being done they make mistakes. This is due to human nature and also to the current complexity of software that is developed. Most of these mistakes end up being defects in some software artifact that is being developed. When the software is being executed, these defects may lead to failures with serious negative impacts, among which we can mention the loss of human lives.

Trying to find how to develop software that does not contain defects has created a diversity of techniques, processes, methods, etc. In particular, several software testing techniques have been developed. These could be classified into static and dynamic. The software product is reviewed by means of the static techniques without being executed. On the other hand, the dynamic testing techniques aim at executing the product in different ways in order to provoke failures in it.

Unfortunately, it is difficult to choose which technique or techniques to use in a certain software development project since the current knowledge available about the effectiveness and the cost of each one of them is not enough. The effectiveness is calculated as the number of defects found applying a technique divided by the total number of defects of the product under test. Normally it is expressed in percentage. The cost of a technique is how much it costs to execute it. It can be measured in time, test cases developed, the inherent complexity of applying the technique, etc. In this work we use as a measure of the cost the time it takes to apply the technique.

Both the effectiveness and the cost can vary for the same technique. This variation may be due to the type of product that is being tested, the type of defects it presents, the complexity of the product, the individual who applies the technique, among other reasons. We are far from knowing in such depth the characteristics of the testing techniques.

One way of analyzing and evaluating the testing techniques is through controlled experiments. Controlled experiments use a methodical process to establish a correspondence between certain ideas or theories with reality. Normally they are conducted in a laboratory environment where certain variables (independent variables) can be controlled while others are observed (dependent variables). The data collected during the experiment are analyzed statistically to know more about the topic as well as to accept or reject certain work hypothesis previously established.

The central topic of this thesis is the study through controlled experiments of the effectiveness and cost of different unit testing techniques. We studied 7 testing techniques in "isolated form" to know about their effectiveness and cost. Studying the techniques in isolated form implies studying them out of any software development process. In Appendix B there is a brief introduction to controlled experiments in software engineering (in Spanish).

We also want to study the testing techniques in the context of the Personal Software Process (PSP). In this way we study testing techniques within a software development process and somehow we get closer to what happens in the industry. In the next section there is a brief introduction to the PSP.

1.2 Background: The Personal Software Process

"The Personal Software Process is a self-improvement process that helps you to control, manage, and improve the way you work."

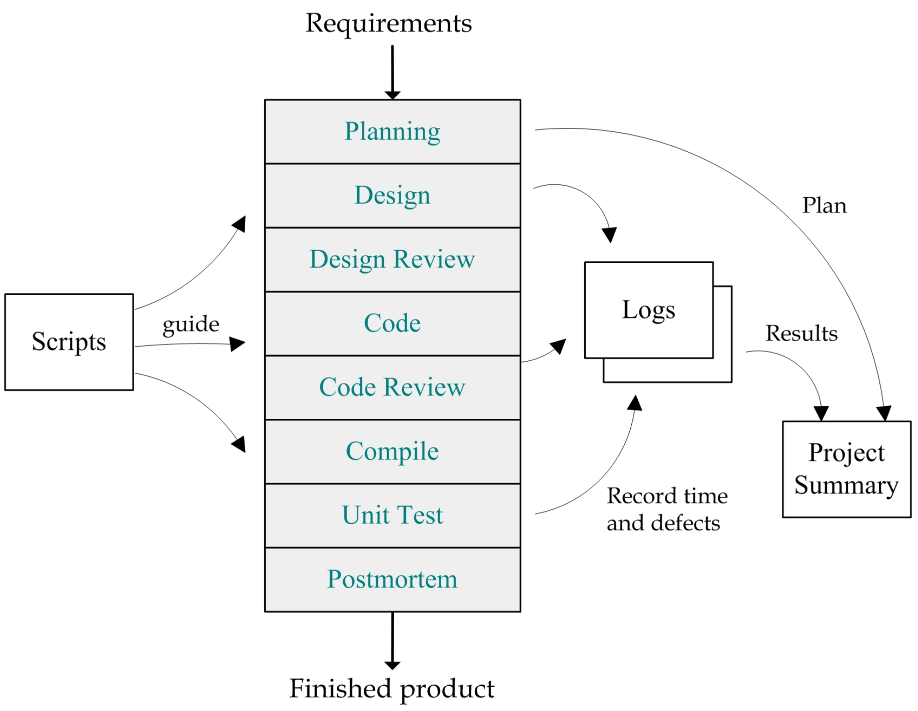


Figure 1.1: The PSP phases, scripts, logs and project summary

– W. S. Humphrey, 2005.

The PSP is a software development process for the individual. The process helps the engineer to control, manage and improve his work. The PSP establishes a highly instrumented development process that includes a rigorous measurement framework for effort and defects. This process includes phases that the engineer completes while building the software.

For each software development phase, the PSP has scripts that help the software engineer to follow the process correctly. The phases include Planning, Detailed Design, Detailed Design Review, Code, Code Review, Compile, Unit Test, and Post Mortem. For each phase, the engineer collects data on the time spent in the development phase and the defects injected and removed. The defect data include the defect type, the time to find and fix the defect, the phase in which the defect was injected, and the phase in which it was removed. Figure 1.1 shows the script, phases, and data collection used with the PSP.

The PSP is taught through a course. During the course, the engineers build programs while they are progressively learning PSP planning, development, and process assessment practices. For the first exercise, the engineer starts with a simple, defined process (the baseline process, called PSP 0); as the class progresses, new process steps and elements are added, from Esti-

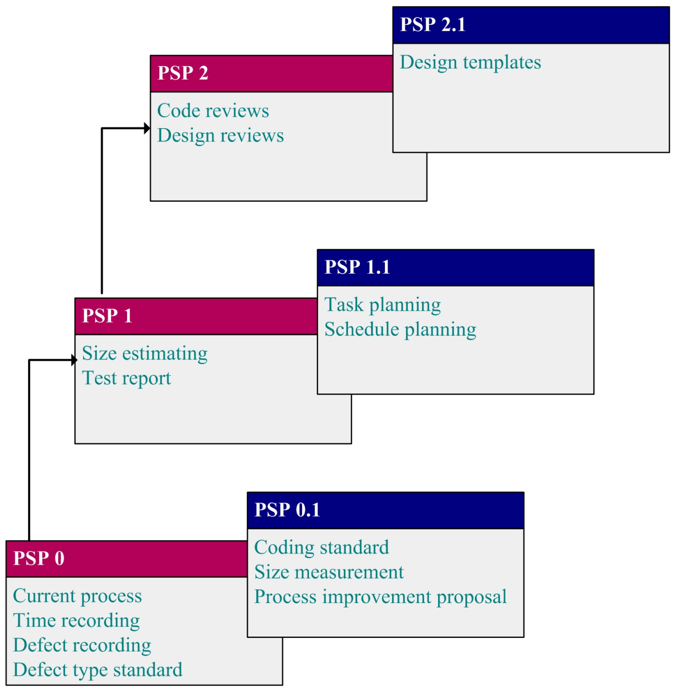


Figure 1.2: PSP process level introduction during course

mation and Planning to Code Reviews, to Design, and Design Review. As these elements are added, the process changes. The name of each process and which elements are added in each one are presented in Figure 1.2. The PSP 2.1 is the complete PSP process.

A more detailed description of the PSP is presented in the second article of Chapter 10.

1.3 Research Goals and Methodology

The general goal of this thesis is to contribute to the knowledge of different testing techniques. More specifically, we intend to conduct an empirical investigation of the results of using them, mainly in what refers to their effectiveness and cost.

This general goal is divided into two particular ones. The first is the study of the following testing techniques: desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, multiple condition coverage, sentence coverage and all uses. The second is the study of the injection and removal of defects in the PSP.

The testing techniques were chosen to vary between static and dynamic techniques. Among the dynamic ones black box and white box testing techniques are included. Techniques based on data flow and control flow are included among the white box testing techniques. On the other hand, the PSP provides naturally an excellent frame for conducting empirical studies due to the collection of data the engineer makes while using it.

Below we present the goals with the associated research questions:

Goal 1 - Investigate empirically the 7 techniques mentioned.

Research question 1: Which is the effectiveness of each one of those techniques?

Research question 2: Which is the cost of each one of those techniques?

Goal 2 - Investigate empirically the PSP defect injection and removal.

Research question 3: Which is the effectiveness of the different PSP's phases?

Research question 4: Which is the cost of finding and fixing a defect in the PSP?

In order to answer the first two research questions we used controlled experiments. The subjects of these experiments were students of Computer Science of the Universidad de la República. The 7 techniques mentioned were used by the students in different programs written in Java. This is the principal goal of the thesis.

In order to answer the research questions 3 and 4 we made an analysis of the data collected in different courses of the PSP. The courses were taught between October 2005 and January 2010 by the Software Engineering Institute (SEI) of the Carnegie Mellon University or by SEI partners. It should be made clear that our intention for the future is to study the variation in effectiveness and cost of the PSP unit test phase when the testing techniques mentioned (except desktop inspection) are used in that phase of the process. However, in order to conduct this empirical study, first we should have data concerning how the PSP behaves when no specific technique is applied in the unit test phase. This is the reason why our second objective is only to have information about the injection and removal of defects in the PSP.

1.4 Controlled Experiments with Seven Testing Techniques

In order to know the effectiveness and cost of the 7 testing techniques mentioned we conducted two controlled experiments. We called the first one Experiment 2008 and the second one Experiment 2009. Their names come from the years in which we started the experiments.

The subjects in both experiments were Computer Science students from the Universidad de la República in Uruguay. All of them were in fourth or fifth year (the last one) of their degree.

During the experiment the subjects executed testing techniques and collected data about time and defects. Each subject was allotted testing techniques to apply on software programs written in Java. While they were executing the tests, the subjects recorded the defects they found. This information is useful to calculate the effectiveness each subject obtained when applying a technique. They also recorded the time it took them to develop a set of test cases that satisfies the prescription of the allotted technique. This data is the cost of applying the technique.

All the subjects were provided with training that aimed at making them use the allotted technique properly. The training consisted of theoretical-practical classes both for Experiment 2008 and for Experiment 2009. During the training the subjects learned not only the techniques to use but also where to record the data they collected (defects and time) during the experiment. In the final part of the training the subjects executed the allotted techniques on a small program.

The training performed during both experiments was considered an experiment in itself. This was possible due to the execution of the techniques on the small program. Chapter 3 contains two articles that present the training of Experiments 2008 and 2009 and the results obtained.

After the training the subjects start the testing task on larger and more complex programs. In the Experiment 2008 each subject used a different technique on three different programs. In the Experiment 2009 the subjects used only one technique on an only program. In the Experiment 2008 the five techniques mentioned first (out of the seven mentioned) were used, while in the Experiment 2009 the last two were used. Chapter 4 contains two articles that present Experiment 2008 and Experiment 2009 in a complete form.

1.5 Complementary Works

We also conducted some research work that complement the main line of research (the study of the effectiveness and cost of different testing techniques) in different ways. All the complementary works are presented in Part II.

An article that presents and evaluates different taxonomies of software defects entitled "A Framework to Evaluate Defect Taxonomies" is included. The evaluation is done through a Comparison Framework we developed as part of our research. Several experiments that aim at knowing the effectiveness of the testing techniques segment the results by type of defect. They aim at knowing how effective a certain technique is for each type of defect. That is why it is important to know different taxonomies and evaluate them.

Different research groups in different places in the world conduct formal experiments in software engineering. In particular, many experiments aim at knowing the cost and effectiveness of different testing techniques. Normally these experiments are very difficult to compare and to add. The article

“Towards a Framework to compare Formal Experiments” presents a first construction of a framework to compare but above all, to identify the most important items of an experiment that intends to evaluate different testing techniques.

Another article included “Construction of a Laboratory Package for a Software Engineering Experiment” presents a packaging experience of the Experiment 2008. Packaging an experiment makes it possible, among other things, for different research groups to replicate it easily (or at least more easily than if it was not packaged). Besides, packaging is also important since it provides complete information on the experiment that has been conducted. It is important to clarify that in the packaging work of the Experiment 2008 I collaborated only providing information about the experiment and making reviews of the article mentioned. That is to say, that the main research work was done by the rest of the authors and not by me.

The article “Calidad de los Datos en Experimentos en Ingeniería de Software. Un Caso de Estudio” [Quality of the data in Software Engineering Experiments. A Case Study] presents a study on the quality of the data of the Experiment 2008. This article is presented in Spanish.

Also, Appendix A presents a collection of short articles that deal with support tools to unit tests. The main topic of these articles is the code coverage and support through tools to know that coverage. These tools can be used in the controlled experiments that use white box testing techniques.

1.6 Analysis of Defect Injection and Removal in PSP

In the controlled experiments we conducted the testing techniques were used in an isolated form. However, this is not what happens in the software industry. The techniques are used as part of a software development process within a project. The process develops a context in which the techniques are not used in a pure form and probably, due to this, the effectiveness and cost of the same change.

Consequently, we conducted a study using the PSP. We analyzed the effectiveness of the PSP phases and the cost of finding and fixing defects in the PSP. We used data from PSP courses taught between October 2005 and January 2010. These courses were taught by the Software Engineering Institute (SEI) of the Carnegie Mellon University or by SEI partners.

In the phases design review and code review of the PSP, as their names indicate, design and code reviews respectively are conducted. These are done using a checklist and that is why the code review is similar to the desktop inspection technique we used in the controlled experiments.

The study done using the PSP is presented in Part III. Two articles are included in it. The first one analyzes the injection and removal of the defects injected during the detailed design phase. The second conducts a similar analysis but with the defects injected during the code phase.

These two articles contain initial results concerning the behavior of the injection and removal of defects of the PSP. This will make it possible to have the necessary basis to later conduct formal experiments using the different verification techniques mentioned.

1.7 About the Thesis Document

This document is made up by this introduction and five Parts. Parts I, II and III have an introduction chapter and a conclusions chapter. The central content of each Part is a compendium of articles most of which have already been published. Since each article has an introduction and its conclusions and we want to avoid being repetitive, the introduction and the conclusions of each Part are very brief.

The state of the art is presented in each article included according to the topic that is treated in it. Due to this, the thesis does not have a chapter entitled “State of the Art” or “Related Work” that is global to it.

Part I presents the controlled experiments conducted: Experiment 2008 and Experiment 2009. Part II presents articles that complement the research on the effectiveness and cost of the testing techniques. Part III presents the analysis of the effectiveness of the PSP phases and the cost of finding and removing defects in the PSP. Part IV presents the conclusions and future work. Finally Part V contains the Appendices.

1.8 Articles that Make Up the Thesis

Below are listed the articles that make up this Thesis in the order they appear in it:

1. Diego Vallespir, Juliana Herbert; *Effectiveness and Cost of Verification Techniques: Preliminary Conclusions on Five Techniques*, Proceedings of the Mexican International Conference on Computer Science, pp. 264-271, Ciudad de México, México, September, 2009. Published by IEEE Computer Society Press.

2. Diego Vallespir, Carmen Bogado, Silvana Moreno, Juliana Herbert; *Comparing Verification Techniques: All Uses and Statement Coverage*, Proceedings of the Ibero-American Symposium on Software Engineering and Knowledge Engineering 2010, pp. 85-95, Mérida, México, November, 2010.

3. Diego Vallespir, Cecilia Apa, Stephanie De León, Rosana Robaina, Juliana Herbert; *Effectiveness of five verification techniques*, Proceed-

ings of the XXVIII International Conference of the Chilean Computer Society, pp. 86-93, Santiago de Chile, Chile, November, 2009.

4. Diego Vallespir, Silvana Moreno, Carmen Bogado, Juliana Herbert; *All Uses and Statement Coverage: A controlled experiment*, This article hasn't been published yet. Submitted to a Software Engineering Open Journal.

5. Diego Vallespir, Fernanda Grazioli, Juliana Herbert; *A Framework to Evaluate Defect Taxonomies*, Proceedings of the Argentine Congress of Computer Science 2009, pp. 643-652, San Salvador de Jujuy, Argentina, October, 2009.

6. Diego Vallespir, Silvana Moreno, Carmen Bogado, Juliana Herbert; *Towards a Framework to Compare Formal Experiments that Evaluate Testing Techniques*, Research in Computing Science, pp. 69-80, ISSN 1870-4069, 2009.

7. Cecilia Apa, Martín Solari, Diego Vallespir, Sira Vegas; *Construction of a Laboratory Package for a Software Engineering Experiment*, Proceedings of the Ibero-American Conference on Software Engineering, pp. 101-114, Rio de Janeiro, Brazil, April 2011.

8. Carolina Valverde, Adriana Marotta, Diego Vallespir; *Calidad de Datos en Experimentos en Ingeniería de Software Un caso de estudio [Data quality in Software Engineering Experiments: A Case Study]* , This article hasn't been published yet. Submitted to a Regional Conference. Article in Spanish.

9. Diego Vallespir, William Nichols; *Analysis of Design Defect Injection and Removal in PSP*, Proceedings of the TSP Symposium 2011: A Dedication to Excellence, pp. 19-24, Atlanta, GA, United States, September, 2011.

10. Diego Vallespir, William Nichols; *Analysis of Code Defects Injection and Removal in PSP*, This article hasn't been published yet. Submitted to a Special Issue of a Software Engineering Open Journal.

11. Collection of articles *Unit Testing in Java*

    (a) Adriana Ávila, Lucía Camilloni, Fernando Marotta, Diego Vallespir, Cecilia Apa; *Unit Testing in Java: JUnit and TestNG*, Journal Milveinticuatro, Thomas Alva Edison Edition, pp 46-48, ISSN: 1688-6941, 2010.

    (b) Adriana Ávila, Lucía Camilloni, Diego Vallespir, Cecilia Apa; *Unit Testing in Java: Code Coverage with Clover*, Journal Milveinticuatro, Alexander Graham Bell Edition, pp 50-53, ISSN: 1688-6941, 2011.

(c) Carmen Bogado, Diego Vallespir; *Unit Testing in Java: Code Coverage with CodeCover*, Journal Milveinticuatro, Blaise Pascal Edition, pp 32-34, ISSN: 1688-6941, 2011.

(d) Diego Vallespir, Fernando Marotta; *Unit Testing in Java: Code Coverage with CoView*, Journal Milveinticuatro, Blaise Pascal Edition, pp 60-62, ISSN: 1688-6941, 2011.

(e) Diego Vallespir, Fernando Marotta, Carmen Bogado; *Unit Testing in Java: Code Coverage with CodeCover and CoView*, Journal Milveinticuatro, Antonio Meucci Edition, pp 32-34, ISSN: 1688-6941, 2011.

(f) Adriana Ávila, Lucía Camilloni, Diego Vallespir; *Unit Testing in Java: Data Driven with TestNG*, Journal Milveinticuatro, Antonio Meucci Edition, pp 60-61, ISSN: 1688-6941, 2011.

12. Cecilia Apa, Rosana Robaina, Stephanie De León, Diego Vallespir; *Conceptos de Ingeniería de Software Empírica*, Technical Report PEDECIBA-Informática 10-02. Article in Spanish.

Part I

Controlled Experiments with Seven Testing Techniques

Chapter 2

Introduction

We conducted two controlled experiments in order to evaluate the cost and the effectiveness of different unit testing techniques. One of the experiments started in the year 2008 (Experiment 2008) and the other experiment in 2009 (Experiment 2009).

The techniques used in these experiments were: desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, multiple condition coverage, statement coverage and all uses. In the Experiment 2008 the five first techniques mentioned above were used, while the other two were used in the 2009 Experiment.

The techniques were chosen in order to vary between static and dynamic techniques. Among the dynamic ones we used white and black box techniques. Finally, among the white box techniques, we chose techniques based on control flow and data flow.

The subjects, advanced Computer Science students of the Universidad de la República in Uruguay, were trained as part of the experiment they participated in. Thus we try to make sure they have the necessary knowledge and practice to use the testing techniques properly. Another aim of the training is to bring all the subjects to the same level.

The training for both experiments has a theoretical part where the subjects learn the testing techniques, and a practical one where the subjects apply them. In this practical part the subjects use the allotted techniques on a small Java program.

The training given to the subjects can be taken as a formal experiment in itself. The articles "Effectiveness and Cost of Verification Techniques: Preliminary Conclusions on Five Techniques" and "Comparing Verification Techniques: All Uses and Statement Coverage", from Chapter 3 "Controlled Experiments with a Small Program", present respectively the trainings of Experiments 2008 and 2009 as formal experiments.

After finishing the training, the subjects used the allotted techniques on bigger and more complex programs. The articles of Chapter 4 "Controlled

Experiments to Evaluate Testing Techniques", present in a complete way the Experiments 2008 and 2009. The first of the articles, "Effectiveness of Five Verification Techniques", presents Experiment 2008. The second, "All Uses and Statement Coverage: A Controlled Experiment" presents the 2009 Experiment.

Chapter 3

Controlled Experiment with a Small Program

In this chapter we present the trainings of the Experiments 2008 and 2009 as formal experiments. The first article corresponds to Experiment 2008 and the second to Experiment 2009.

The second article was published in Spanish. This thesis includes a translation into English. It sticks to the formats of the proceedings of the conference where it was published. This article is 10 pages long whereas the original is 11, due to the translation and the format.

The papers included in this chapter are:

Effectiveness and Cost of Verification Techniques: Preliminary Conclusions on Five Techniques
Diego Vallespir and Juliana Herbert
Proceedings of the Mexican International Conference on Computer Science, pp. 264-271, Ciudad de México, México, September, 2009. Published by IEEE Computer Society Press.

Comparing Verification Techniques: All Uses and Statement Coverage
Diego Vallespir, Carmen Bogado, Silvana Moreno and Juliana Herbert
Proceedings of the Ibero-American Symposium on Software Engineering and Knowledge Engineering 2010, pp. 85-95, Mérida, México, November, 2010.

ARTICLE

Published by the IEEE Computer Society
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314

IEEE Computer Society Order Number P3882
ISBN 978-0-7695-3882-7
Library of Congress Number 2009935369
BMS Part Number CFP0937I-PRT

Effectiveness and Cost of Verification Techniques: Preliminary Conclusions on Five Techniques

Diego Vallespir and Juliana Herbert

Proceedings of the Mexican International Conference on Computer Science, pp. 264-271, Ciudad de México, México, September, 2009. Published by IEEE Computer Society Press.

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.

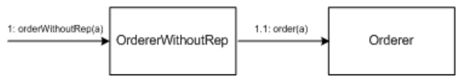


Figure 1. UML Collaboration Diagram of the Program

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.

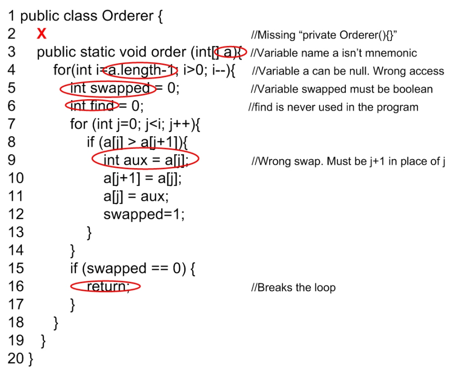
The class ***Orderer*** has 7 defects to consider. They are named with uppercase letters from letter *A* to letter *G*. From the class *OrdererWithoutRep* 6 defects are analyzed, which are named with lowercase letters from letter *a* to letter *f*.

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.

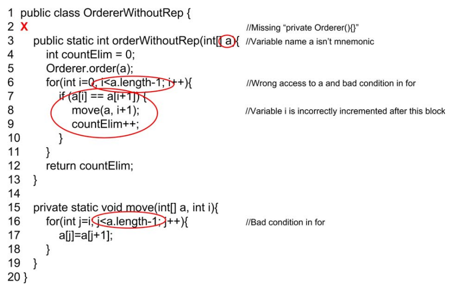
Defect E - NF
The variable name for the array `a` is not mnemonic. Replacing the name affects several lines of code.

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.

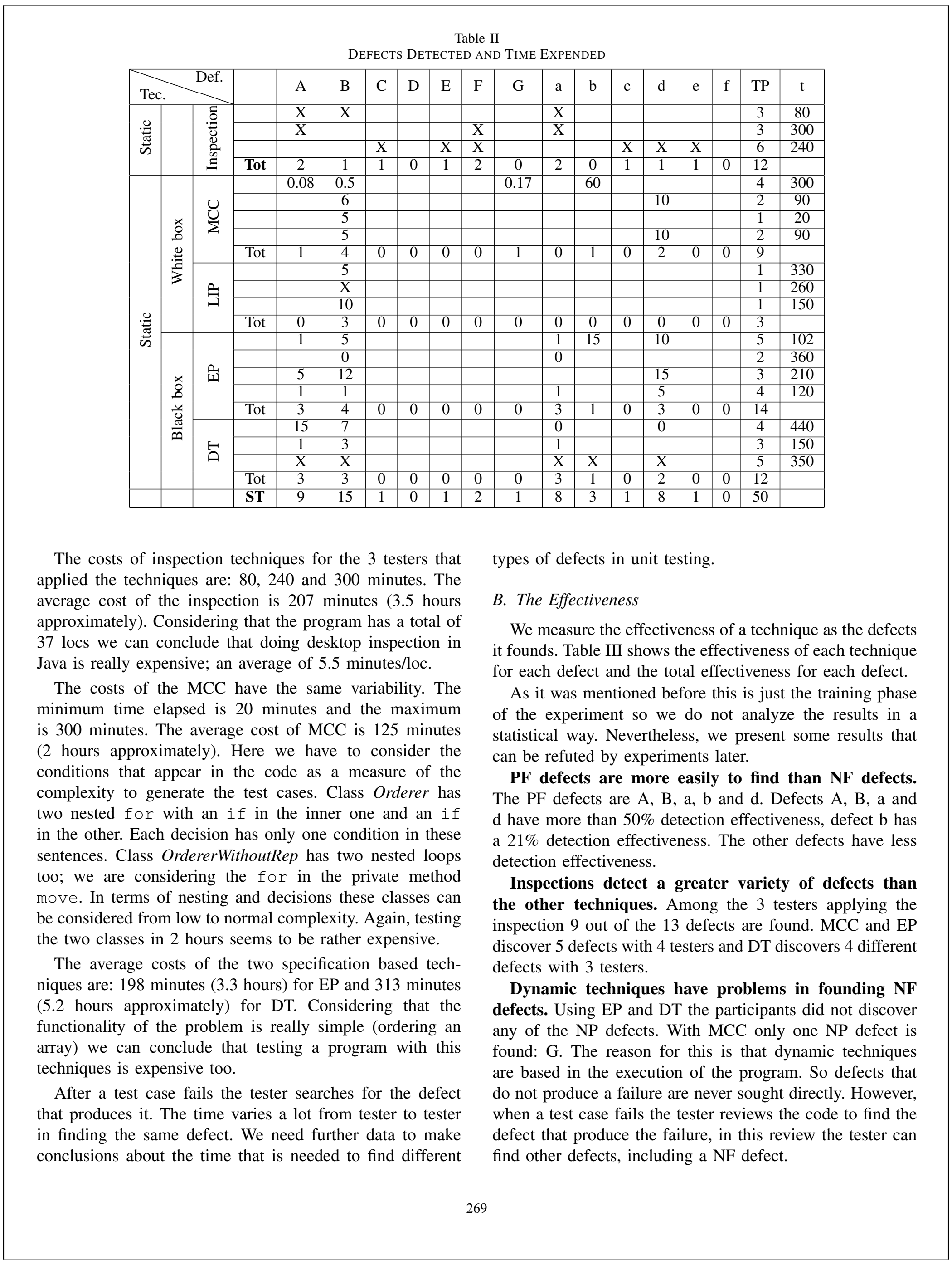


Table II
DEFECTS DETECTED AND TIME EXPENDED

Def. / Tec.				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

Tec. \ Def.	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, 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.

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

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.

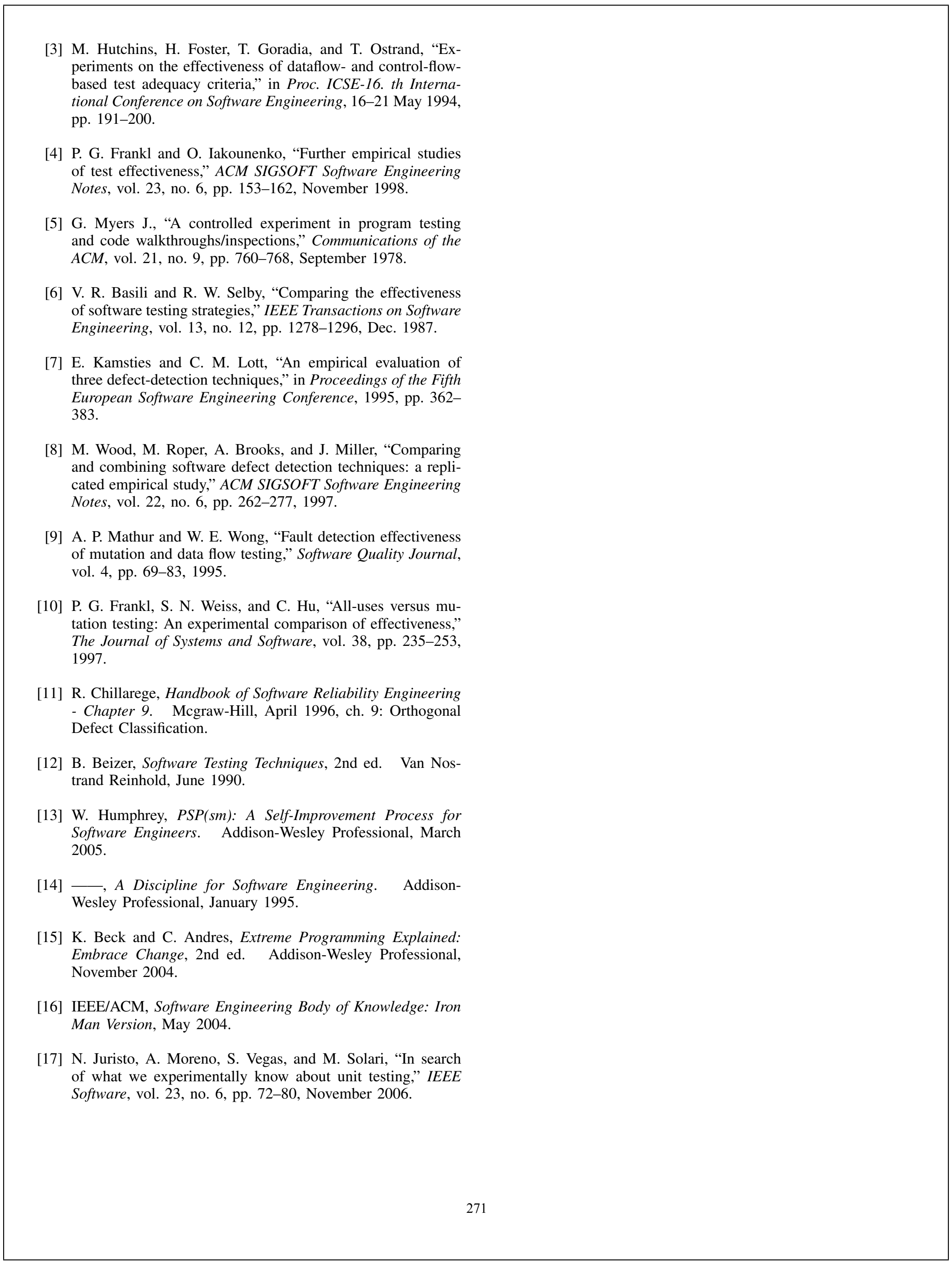
ACKNOWLEDGMENT

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.

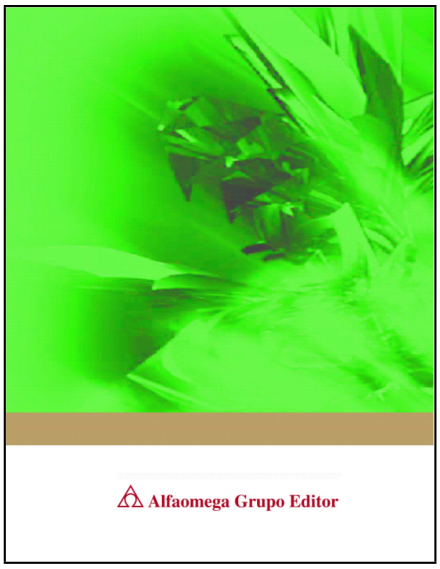
[15] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley Professional, November 2004.

[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.

ARTICLE

Ingeniería de Software e Ingeniería del Conocimiento:
Tendencias de Investigación e Innovación Tecnológica en Iberoamérica
Editores:
Raúl A. Aguilar Vera
Julio C. Díaz Mendoza
Gerzon E. Gómez Cruz
Edwin León Bojórquez
ISBN: 978-607-707-096-2



Comparing Verification Techniques: All Uses and Statement Coverage

Diego Vallespir, Carmen Bogado, Silvana Moreno and Juliana Herbert

Proceedings of the Ibero-American Symposium on Software Engineering and Knowledge Engineering 2010, pp. 85-95, Mérida, México, November, 2010.

Comparing Verification Techniques All Uses and Statement Coverage

Diego Vallespir[1], Carmen Bogado[1], Silvana Moreno[1], and Juliana Herbert[2]

[1] Instituto de Computación, Universidad de la República
Montevideo, Uruguay.
[2] Herbert Consulting
Porto Alegre, RS, Brazil.
dvallesp@fing.edu.uy, {silvanamoren, cmbogado}@gmail.com,
juliana@herbertconsulting.com

Abstract. This article presents a formal experiment that compares the behavior of the testing techniques Statement Coverage and All uses. The design of this experiment is typical for a factor with two alternatives. A total of 21 subjects carry out tests on a single small program. The results indicate that statistically it is not possible to differentiate between the effectiveness of both techniques. However, there is enough statistical evidence to state that the cost of executing All uses is higher than that of executing Statement coverage, result that we expected to find.

1 Introduction

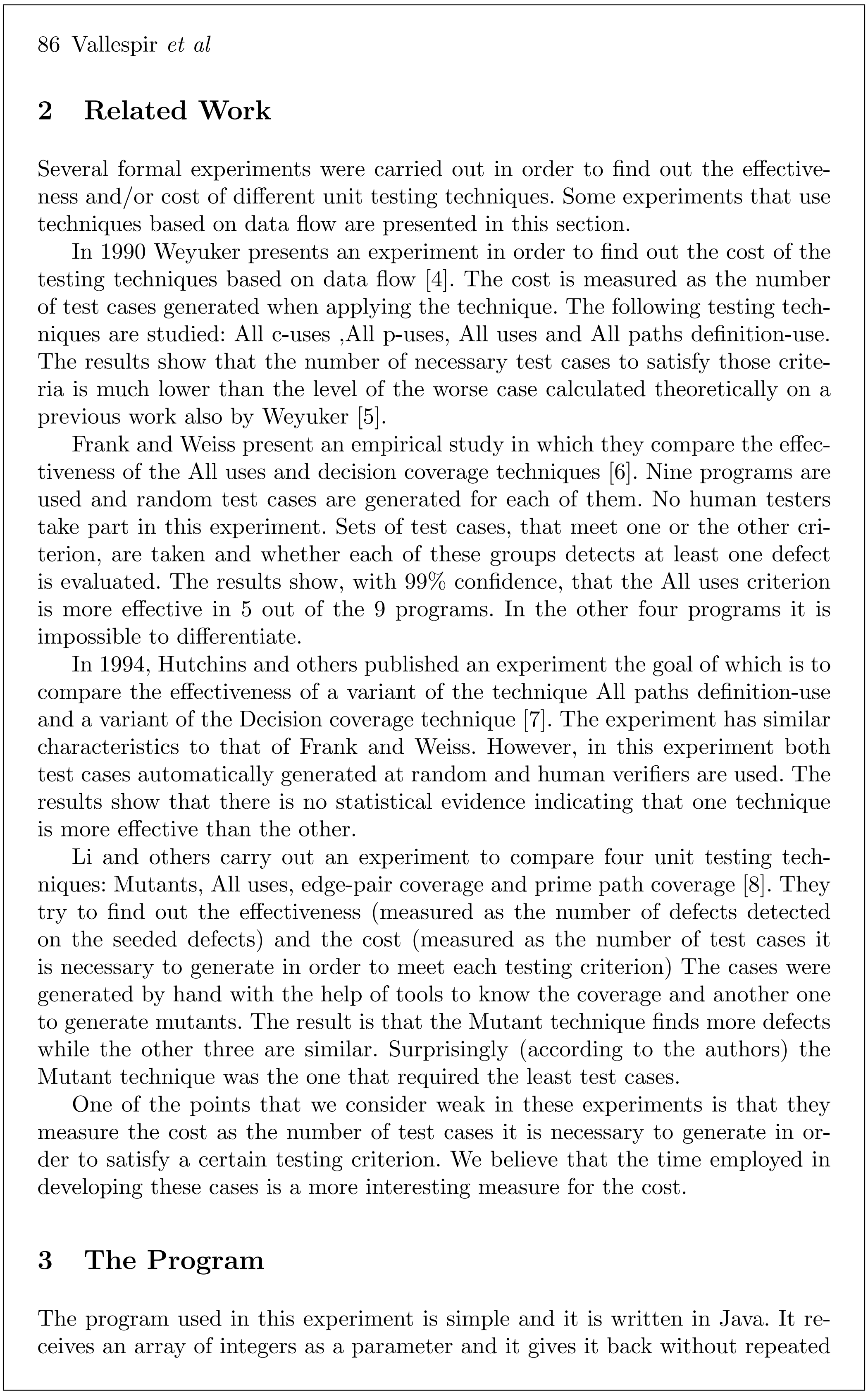
Software unit testing is strongly established in industry. However, the effectiveness and cost of each different unit testing techniques is not known with certainty. This makes the decision of which technique to use hardly trivial.

Many years of empirical research have gone by and yet final results have not been achieved. *In A look at 25 years of data* the authors examine in depth different experiments of software testing reaching the same conclusion [1].

A series of formal experiments are currently being carried out at the Computer Science Institute of the Universidad de la República in order to gather more precise data in this direction. Four experiments have been finished at present and this article describes one of them. The results of other experiments of this series are included in [2, 3].

The experiment hereby described compares the unit testing All uses and Statement coverage in order to know its cost and effectiveness. The cost is defined as the time it takes to develop the testing cases in order to comply with the coverage demanded by the technique. Effectiveness is defined as the number of defects encountered when executing the technique divided by the number of total defects of the program being tested.

Section 2 presents the related works. Section 3 presents the program used in the experiment. The techniques used are presented in section 4. The design of the experiment and its execution are presented in section 5. The results of the experiment and the conclusions are presented in section 6. Section 7 contains future work.



2 Related Work

Several formal experiments were carried out in order to find out the effectiveness and/or cost of different unit testing techniques. Some experiments that use techniques based on data flow are presented in this section.

In 1990 Weyuker presents an experiment in order to find out the cost of the testing techniques based on data flow [4]. The cost is measured as the number of test cases generated when applying the technique. The following testing techniques are studied: All c-uses ,All p-uses, All uses and All paths definition-use. The results show that the number of necessary test cases to satisfy those criteria is much lower than the level of the worse case calculated theoretically on a previous work also by Weyuker [5].

Frank and Weiss present an empirical study in which they compare the effectiveness of the All uses and decision coverage techniques [6]. Nine programs are used and random test cases are generated for each of them. No human testers take part in this experiment. Sets of test cases, that meet one or the other criterion, are taken and whether each of these groups detects at least one defect is evaluated. The results show, with 99% confidence, that the All uses criterion is more effective in 5 out of the 9 programs. In the other four programs it is impossible to differentiate.

In 1994, Hutchins and others published an experiment the goal of which is to compare the effectiveness of a variant of the technique All paths definition-use and a variant of the Decision coverage technique [7]. The experiment has similar characteristics to that of Frank and Weiss. However, in this experiment both test cases automatically generated at random and human verifiers are used. The results show that there is no statistical evidence indicating that one technique is more effective than the other.

Li and others carry out an experiment to compare four unit testing techniques: Mutants, All uses, edge-pair coverage and prime path coverage [8]. They try to find out the effectiveness (measured as the number of defects detected on the seeded defects) and the cost (measured as the number of test cases it is necessary to generate in order to meet each testing criterion) The cases were generated by hand with the help of tools to know the coverage and another one to generate mutants. The result is that the Mutant technique finds more defects while the other three are similar. Surprisingly (according to the authors) the Mutant technique was the one that required the least test cases.

One of the points that we consider weak in these experiments is that they measure the cost as the number of test cases it is necessary to generate in order to satisfy a certain testing criterion. We believe that the time employed in developing these cases is a more interesting measure for the cost.

3 The Program

The program used in this experiment is simple and it is written in Java. It receives an array of integers as a parameter and it gives it back without repeated elements and ordered from small to large. It has two classes (Orderer and OrdererWithoutRep), one of them contains 18 lines of code with no comments and the other one contains 19. The interaction between the classes is also simple: the class OrdererWithoutRep invokes a method of the class Orderer for the array to be ordered before the repeated elements are eliminated.

Below the signature, specification and source code of the method *order* of the class *Orderer* and the method *orderWithoutRep* of the class *OrdererWithoutRep* are presented

---

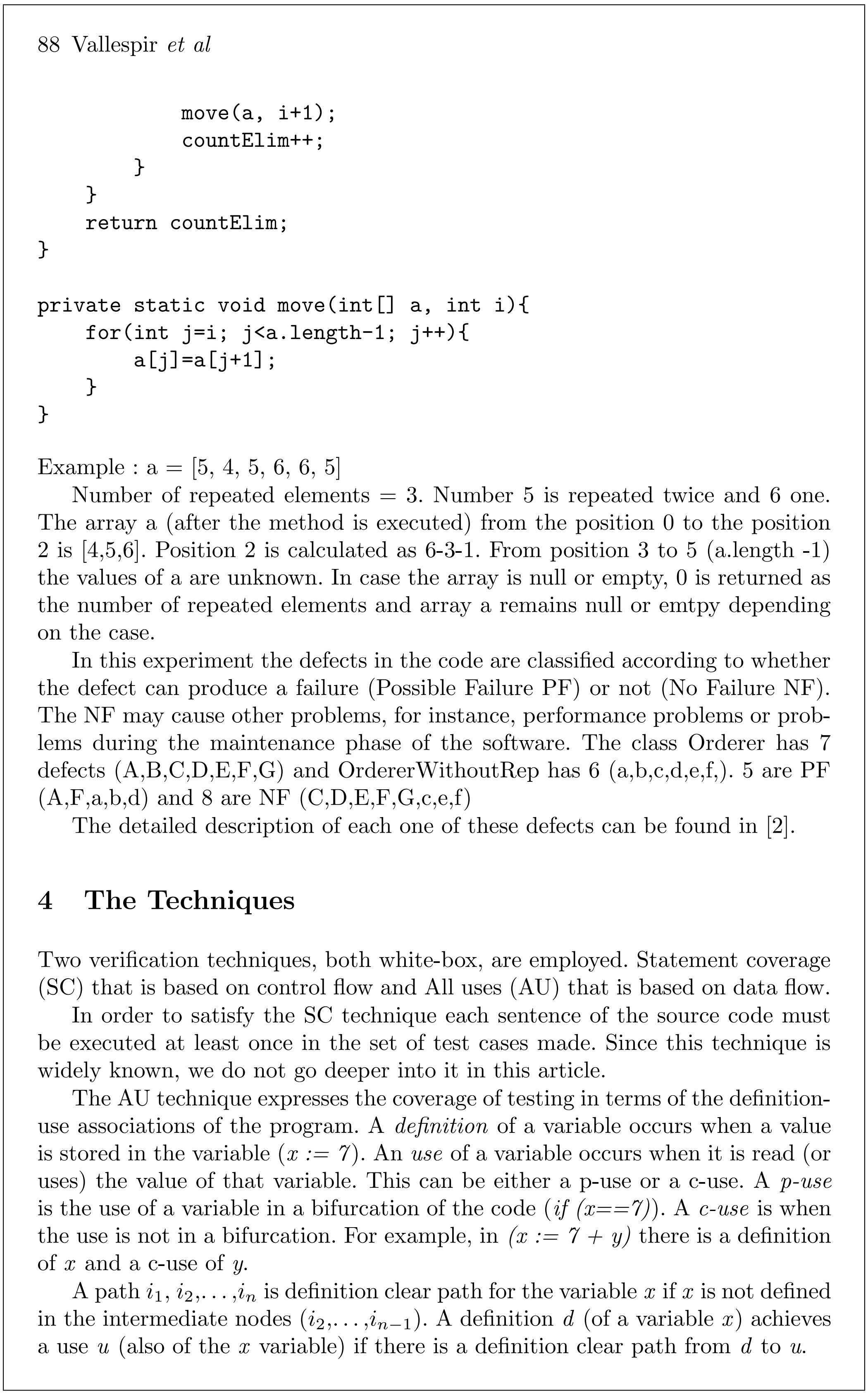
public static void order(int[] a)
The method returns the array ordered from smaller to larger. In case the array is null or empty, it remains unchanged.
a: entry parameter that contains the integers to be ordered.

```
public static void order(int[] a){
    for(int i=a.length-1; i>0; i--){
        int swapped = 0;
        int find = 0;
        for (int j=0; j<i; j++){
            if (a[j] > a[j+1]){
                int aux = a[j];
                a[j+1] = a[j];
                a[j] = aux;
                swapped=1;
            }
        }
        if (swapped == 0) {
            return;
        }
    }
}
```

---

public static int orderWithoutRep (int[] a)
The method returns the array ordered from smaller to larger and without repeated elements from the position 0 to the position a.length - the number of elements repeated - 1. And from there up to a.length -1 the values are unknown (that is to say, they are irrelevant).

```
public static int orderWithoutRep(int[] a){
    int countElim = 0;
    Orderer.order(a);
    for(int i=0; i<a.length-1; i++){
        if (a[i] == a[i+1]) {
```



```
            move(a, i+1);
            countElim++;
        }
    }
    return countElim;
}

private static void move(int[] a, int i){
    for(int j=i; j<a.length-1; j++){
        a[j]=a[j+1];
    }
}
```

Example : a = [5, 4, 5, 6, 6, 5]

Number of repeated elements = 3. Number 5 is repeated twice and 6 one. The array a (after the method is executed) from the position 0 to the position 2 is [4,5,6]. Position 2 is calculated as 6-3-1. From position 3 to 5 (a.length -1) the values of a are unknown. In case the array is null or empty, 0 is returned as the number of repeated elements and array a remains null or emtpy depending on the case.

In this experiment the defects in the code are classified according to whether the defect can produce a failure (Possible Failure PF) or not (No Failure NF). The NF may cause other problems, for instance, performance problems or problems during the maintenance phase of the software. The class Orderer has 7 defects (A,B,C,D,E,F,G) and OrdererWithoutRep has 6 (a,b,c,d,e,f,). 5 are PF (A,F,a,b,d) and 8 are NF (C,D,E,F,G,c,e,f)

The detailed description of each one of these defects can be found in [2].

4 The Techniques

Two verification techniques, both white-box, are employed. Statement coverage (SC) that is based on control flow and All uses (AU) that is based on data flow.

In order to satisfy the SC technique each sentence of the source code must be executed at least once in the set of test cases made. Since this technique is widely known, we do not go deeper into it in this article.

The AU technique expresses the coverage of testing in terms of the definition-use associations of the program. A *definition* of a variable occurs when a value is stored in the variable ($x := 7$). An *use* of a variable occurs when it is read (or uses) the value of that variable. This can be either a p-use or a c-use. A *p-use* is the use of a variable in a bifurcation of the code (*if* $(x{=}{=}7)$). A *c-use* is when the use is not in a bifurcation. For example, in $(x := 7 + y)$ there is a definition of x and a c-use of y.

A path i_1, i_2,. . . ,i_n is definition clear path for the variable x if x is not defined in the intermediate nodes (i_2,. . . ,i_{n-1}). A definition d (of a variable x) achieves a use u (also of the x variable) if there is a definition clear path from d to u.

AU requires that at least one definition clear path be executed from each definition d to each achievable use. The classical definitions of the techniques based on data flow and particularly AU are presented in an article by Rapss and Weyuker [9].

In Object Oriented languages the basic testing unit is the class. It is necessary to test its methods in an individual and in a collective way, so as to test the interactions generated through the sequence of calls originated by the invocation of a particular method. AU can be applied both for the tests of individual method belonging to a class and for the methods that interact with other methods of the same class or of other classes.

The tests of a class in AU can be carried out in two levels: Intra-method (Intra) and Inter-method (Inter). In **Intra**, only the method under test is considered for the code coverage. Therefore, in this case, the methods that interact with the method under test are not considered at the moment of developing the test cases. On the other hand, in **Inter**, the methods that interact with the method under test are considered for the code coverage.

Two types of definition-use pairs to be tested are identified in relation to the levels presented previously. The **Intra-method Pairs** are those which take place in individual methods and test the data flow limited to such methods. Both definition and use belong to the method under test. **Inter-method Pairs** occur when there is interaction between methods. They are pairs where the definition belongs to a method and the corresponding use is located in another method that belongs to the chain of invocations.

In most of the literature that presents techniques based on data flow, the examples that are given contain simple variables such as integers and Booleans. However, criteria that normally are not treated should be defined at the moment of applying these techniques in array or even more difficult in objects.

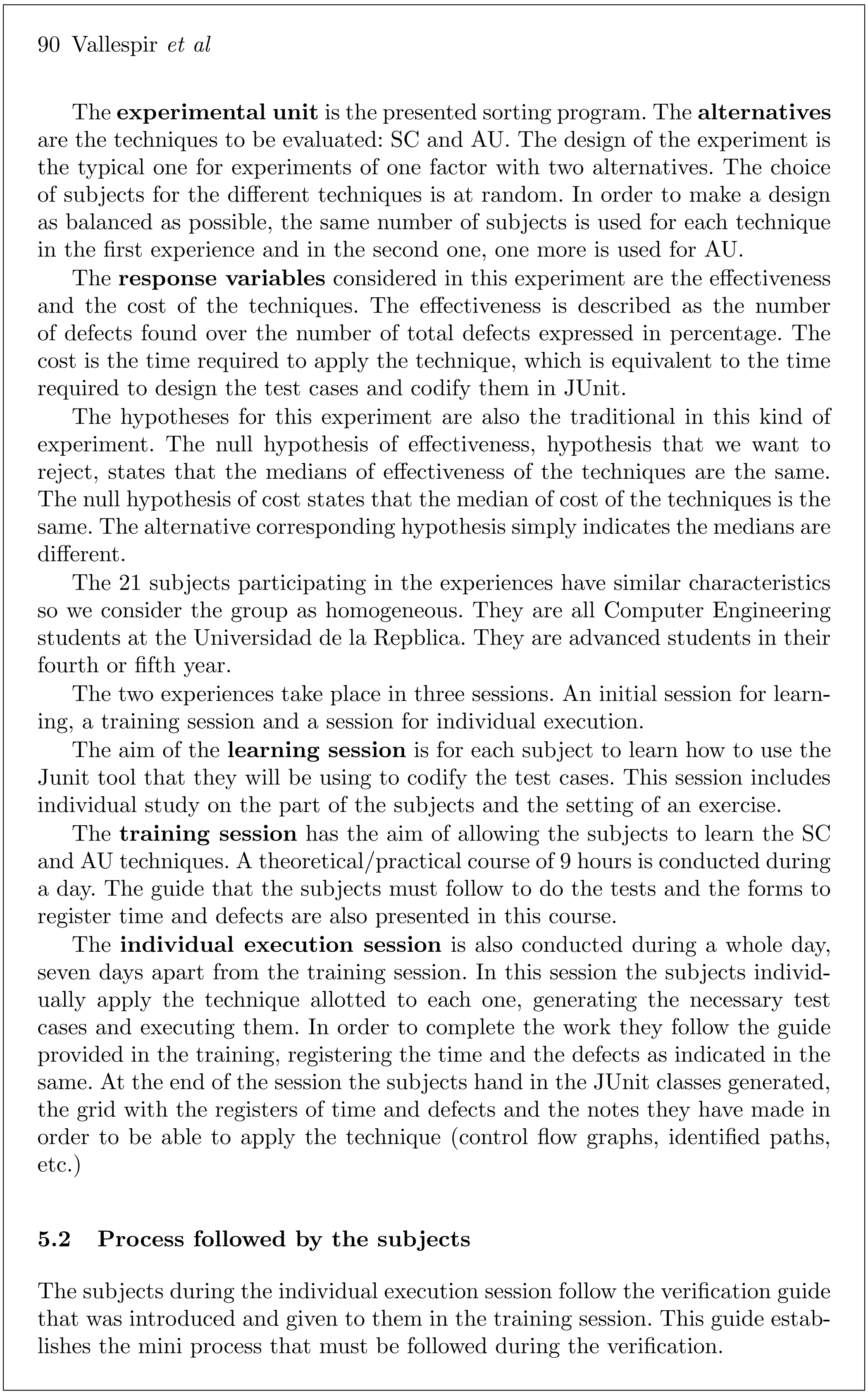
Establishing these criteria is essential in order to know under which conditions the technique is applied. Different conditions can produce different results in the effectiveness and cost of AU since in fact, they are different techniques with the same name. Many of these conditions refer to how the Inter-method Pairs should be considered. This experiment establishes the conditions for the application of the AU technique based on what is proposed in [10–12].

5 Design and Conduction of the Experiment

The general considerations of the design and conduction of the experiment are presented in this section as well as the process followed by the subjects to test the program.

5.1 General Considerations of the Design and Execution

This experiment comprises two experiences. The first one was carried out with 10 subjects and the second one with 11.These experiences were carried out with a difference of 3 weeks between each other.



90 Vallespir *et al*

The **experimental unit** is the presented sorting program. The **alternatives** are the techniques to be evaluated: SC and AU. The design of the experiment is the typical one for experiments of one factor with two alternatives. The choice of subjects for the different techniques is at random. In order to make a design as balanced as possible, the same number of subjects is used for each technique in the first experience and in the second one, one more is used for AU.

The **response variables** considered in this experiment are the effectiveness and the cost of the techniques. The effectiveness is described as the number of defects found over the number of total defects expressed in percentage. The cost is the time required to apply the technique, which is equivalent to the time required to design the test cases and codify them in JUnit.

The hypotheses for this experiment are also the traditional in this kind of experiment. The null hypothesis of effectiveness, hypothesis that we want to reject, states that the medians of effectiveness of the techniques are the same. The null hypothesis of cost states that the median of cost of the techniques is the same. The alternative corresponding hypothesis simply indicates the medians are different.

The 21 subjects participating in the experiences have similar characteristics so we consider the group as homogeneous. They are all Computer Engineering students at the Universidad de la Repblica. They are advanced students in their fourth or fifth year.

The two experiences take place in three sessions. An initial session for learning, a training session and a session for individual execution.

The aim of the **learning session** is for each subject to learn how to use the Junit tool that they will be using to codify the test cases. This session includes individual study on the part of the subjects and the setting of an exercise.

The **training session** has the aim of allowing the subjects to learn the SC and AU techniques. A theoretical/practical course of 9 hours is conducted during a day. The guide that the subjects must follow to do the tests and the forms to register time and defects are also presented in this course.

The **individual execution session** is also conducted during a whole day, seven days apart from the training session. In this session the subjects individually apply the technique allotted to each one, generating the necessary test cases and executing them. In order to complete the work they follow the guide provided in the training, registering the time and the defects as indicated in the same. At the end of the session the subjects hand in the JUnit classes generated, the grid with the registers of time and defects and the notes they have made in order to be able to apply the technique (control flow graphs, identified paths, etc.)

5.2 Process followed by the subjects

The subjects during the individual execution session follow the verification guide that was introduced and given to them in the training session. This guide establishes the mini process that must be followed during the verification.

The guide establishes that both techniques must be executed only as white-box testing. This means that no black-box testing cases should be generated, then the coverage achieved must be observed with an adequate tool. Finally, they must complement with missing cases to reach the coverage. Then each subject generates their test cases from the source code trying to achieve the coverage required by the technique assigned (SC or AU). No code coverage tools are used during the experiment.

The guide establishes that the subjects that must apply AU, must first generate the cases based in intra-method and then generate the inter-method cases. There is no other aspect established about how to apply each technique and this can be chosen by each subject.

Finally the mini process establishes which data must be collected. The subjects must record the total time used in developing and codifying in JUnit the test cases, each of the defects found and the time used in detecting a defect after provoking a failure.

6 Results and Discussion

The following data is available from each subject: the defects detected by the subject, the total time of design and codification of the test cases and the detection time for each defect. According to the length of the paper only the analysis of the effectiveness and cost of the techniques (design and codification time) are included.

6.1 1 Effectiveness of the Techniques

The average effectiveness obtained using the Statement coverage technique was of 29.2% and the standard deviation was 10.8%. All uses obtained a higher effectiveness. Its average effectiveness was 34.3% and the standard deviation was 7.2%.

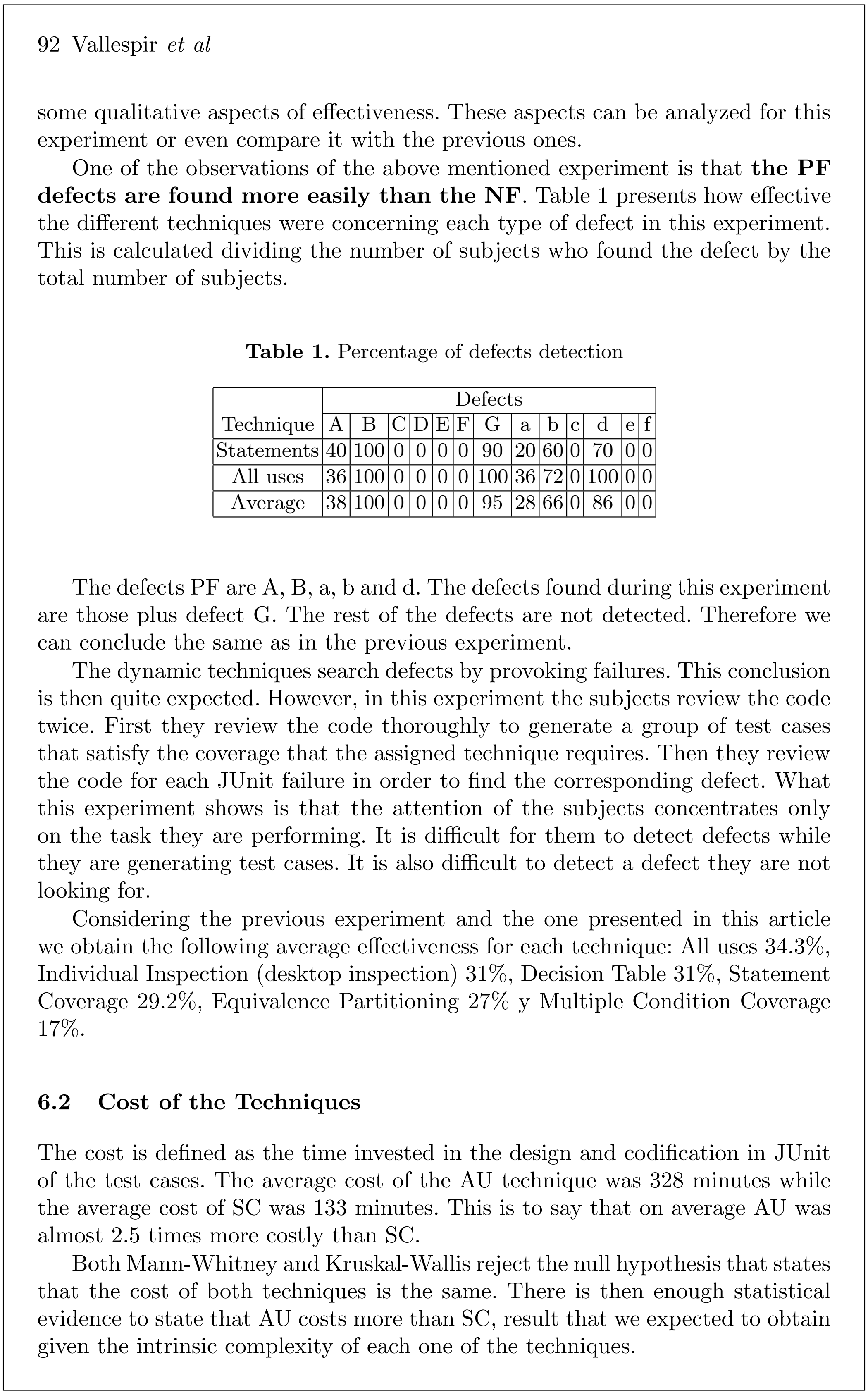
The nonparametric tests of Mann-Whitney and Kruskal-Wallis are applied to find out if it is possible to state that AU was more effective than SC with statistical validity. The null hypothesis states the effectiveness medians are the same, the alternative hypothesis states that they are different.

$H_0 : \mu_{SC} = \mu_{AU}$

$H_1 : \mu_{SC} <> \mu_{AU}$

None of the tests rejects H_0 with $\alpha \leq 0.1$ (habitual value of α with which we work in our experiments). We understand that more observations are needed in this respect (only 21 subjects participate in this experiment) to be able to conclude about the effectiveness with statistical validity. It is also reasonable to vary the programs used in the experiment and to use more complex ones.

We mentioned before that we have conducted other experiments. One of them was conducted with the same program used in this one, but with other subjects and other verification techniques [2]. In those experiments we studied



some qualitative aspects of effectiveness. These aspects can be analyzed for this experiment or even compare it with the previous ones.

One of the observations of the above mentioned experiment is that **the PF defects are found more easily than the NF**. Table 1 presents how effective the different techniques were concerning each type of defect in this experiment. This is calculated dividing the number of subjects who found the defect by the total number of subjects.

Table 1. Percentage of defects detection

	Defects												
Technique	A	B	C	D	E	F	G	a	b	c	d	e	f
Statements	40	100	0	0	0	0	90	20	60	0	70	0	0
All uses	36	100	0	0	0	0	100	36	72	0	100	0	0
Average	38	100	0	0	0	0	95	28	66	0	86	0	0

The defects PF are A, B, a, b and d. The defects found during this experiment are those plus defect G. The rest of the defects are not detected. Therefore we can conclude the same as in the previous experiment.

The dynamic techniques search defects by provoking failures. This conclusion is then quite expected. However, in this experiment the subjects review the code twice. First they review the code thoroughly to generate a group of test cases that satisfy the coverage that the assigned technique requires. Then they review the code for each JUnit failure in order to find the corresponding defect. What this experiment shows is that the attention of the subjects concentrates only on the task they are performing. It is difficult for them to detect defects while they are generating test cases. It is also difficult to detect a defect they are not looking for.

Considering the previous experiment and the one presented in this article we obtain the following average effectiveness for each technique: All uses 34.3%, Individual Inspection (desktop inspection) 31%, Decision Table 31%, Statement Coverage 29.2%, Equivalence Partitioning 27% y Multiple Condition Coverage 17%.

6.2 Cost of the Techniques

The cost is defined as the time invested in the design and codification in JUnit of the test cases. The average cost of the AU technique was 328 minutes while the average cost of SC was 133 minutes. This is to say that on average AU was almost 2.5 times more costly than SC.

Both Mann-Whitney and Kruskal-Wallis reject the null hypothesis that states that the cost of both techniques is the same. There is then enough statistical evidence to state that AU costs more than SC, result that we expected to obtain given the intrinsic complexity of each one of the techniques.

As refers to the cost, the most important aspect is that AU costs much more. However, the gain in effectiveness was not so important.

7 Conclusions

This work presents a formal experiment to compare the SA and AU techniques as refers to cost and effectiveness. The cost is measured as the time used by the subject to develop test cases. The effectiveness is defined as the number of defects found divided by the total number of defects.

Only one program written in Java is used as unit under test. This program has at least two problems: it is too simple and its defect density is too high. It is therefore important to conduct other experiments with more complex programs and a lower defect density.

The design of the experiment is typical of one factor with two alternatives. In this case it is divided in two experiences, one with 10 subjects (5 apply SC and 5 apply AU) and another one with 11 subjects (5 apply SC and 6 apply AU).

The subjects are all advanced students of Computer Engineering at the Universidad de la República in Uruguay. There are two sessions of leveling and teaching; learning and training sessions.

During the execution the subjects develop the test cases following a guide that was introduced and handed out in the training session. The test cases must be codified in JUnit. The construction of these is exclusively white-box and no code-coverage tools were used. The coverage must be ensured by the construction of the test cases.

The results obtained show that the AU technique has a higher effectiveness than the SC technique; 34.3% is the average effectiveness of AU and 29.2% is the one of SC. However, the statistical tests do not reject the equality hypothesis in the effectiveness of both techniques.

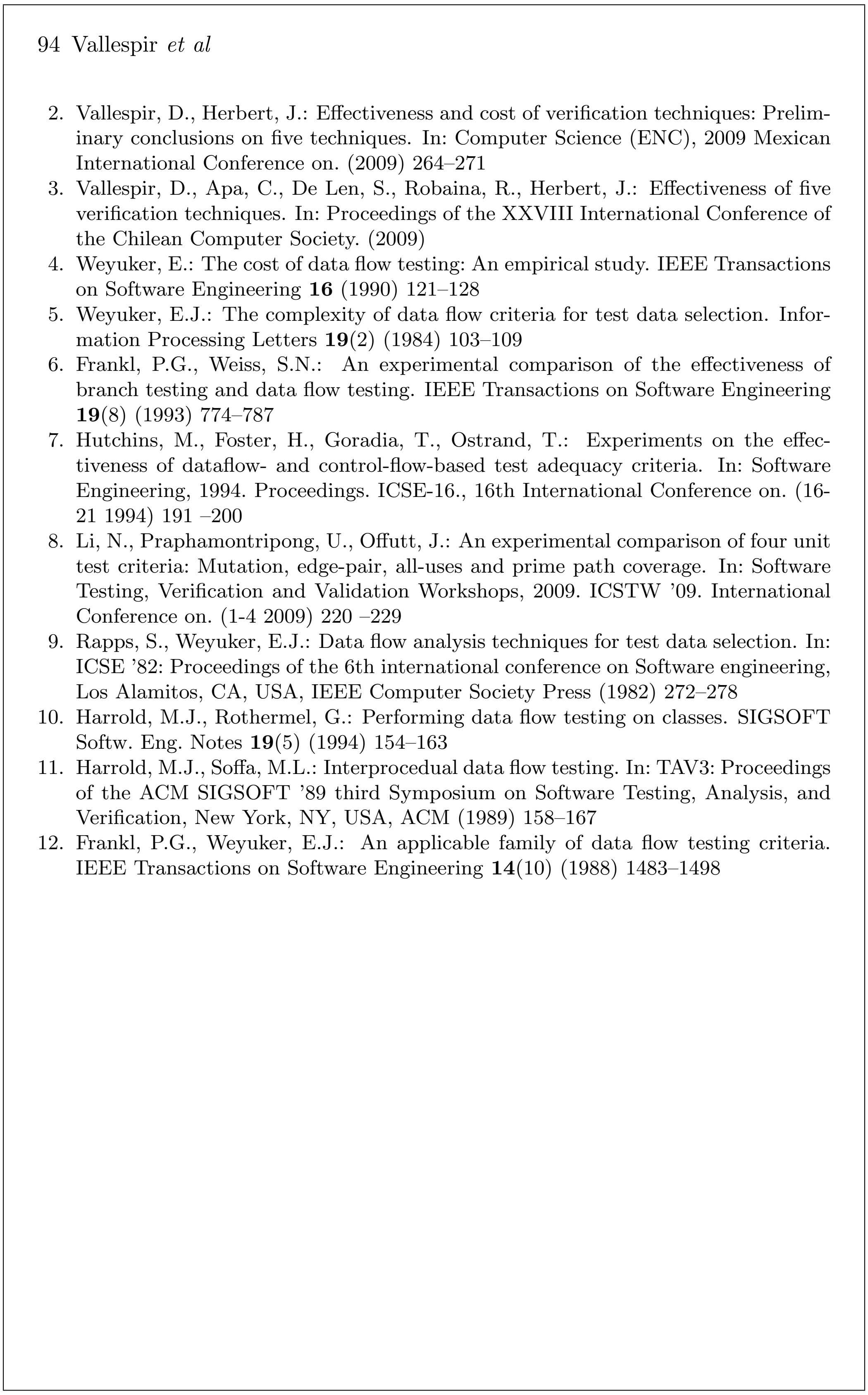
We also found that the AU technique is notoriously more costly than the SC technique; 328 minutes for AU and 133 minutes for SC were needed on average to develop the test cases. The statistical tests applied show that there is enough statistical evidence in this experiment to state that AU is more costly than SC.

At the moment we are analyzing data from a new experiment that we have conducted and also analyzing data from previous experiments from a qualitative point of view. This new experiment is conducted with a more real program of 1820 lines of code without comments and 14 classes, also developed in Java.

In the near future we are planning to conduct experiments with a higher population of subjects. We are also interested in finding out the impact of using code coverage tools concerning both the cost and the effectiveness of white-box techniques.

References

1. Moreno, A., Shull, F., Juristo, N., Vegas, S.: A look at 25 years of data. IEEE Software **26**(1) (Jan.–Feb. 2009) 15–17



2. Vallespir, D., Herbert, J.: Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. In: Computer Science (ENC), 2009 Mexican International Conference on. (2009) 264–271
3. Vallespir, D., Apa, C., De Len, S., Robaina, R., Herbert, J.: Effectiveness of five verification techniques. In: Proceedings of the XXVIII International Conference of the Chilean Computer Society. (2009)
4. Weyuker, E.: The cost of data flow testing: An empirical study. IEEE Transactions on Software Engineering **16** (1990) 121–128
5. Weyuker, E.J.: The complexity of data flow criteria for test data selection. Information Processing Letters **19**(2) (1984) 103–109
6. Frankl, P.G., Weiss, S.N.: An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transactions on Software Engineering **19**(8) (1993) 774–787
7. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on. (16-21 1994) 191 –200
8. Li, N., Praphamontripong, U., Offutt, J.: An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on. (1-4 2009) 220 –229
9. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: ICSE '82: Proceedings of the 6th international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1982) 272–278
10. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. SIGSOFT Softw. Eng. Notes **19**(5) (1994) 154–163
11. Harrold, M.J., Soffa, M.L.: Interprocedual data flow testing. In: TAV3: Proceedings of the ACM SIGSOFT '89 third Symposium on Software Testing, Analysis, and Verification, New York, NY, USA, ACM (1989) 158–167
12. Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering **14**(10) (1988) 1483–1498

Chapter 4

Controlled Experiments to Evaluate Testing Techniques

Experiments 2008 and 2009 are presented in their complete form in this chapter. The first article corresponds to Experiment 2008 and the second to Experiment 2009.

The second article was sent to an international journal to be published and is in the process of being evaluated.

The papers included in this chapter are:

Effectiveness of five verification techniques
Diego Vallespir, Cecilia Apa, Stephanie De León, Rosana Robaina and Juliana Herbert
Proceedings of the XXVIII International Conference of the Chilean Computer Society, pp. 86-93, Santiago de Chile, Chile, November, 2009.

All Uses and Statement Coverage: A controlled experiment
Diego Vallespir, Silvana Moreno, Carmen Bogado and Juliana Herbert
Submitted to a Software Engineering Open Journal. The format is not the Journal format.

ARTICLE

Effectiveness of five verification techniques

Diego Vallespir, Cecilia Apa, Stephanie De León, Rosana Robaina and Juliana Herbert

Proceedings of the XXVIII International Conference of the Chilean Computer Society, pp. 86-93, Santiago de Chile, Chile, November, 2009.

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: 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 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.

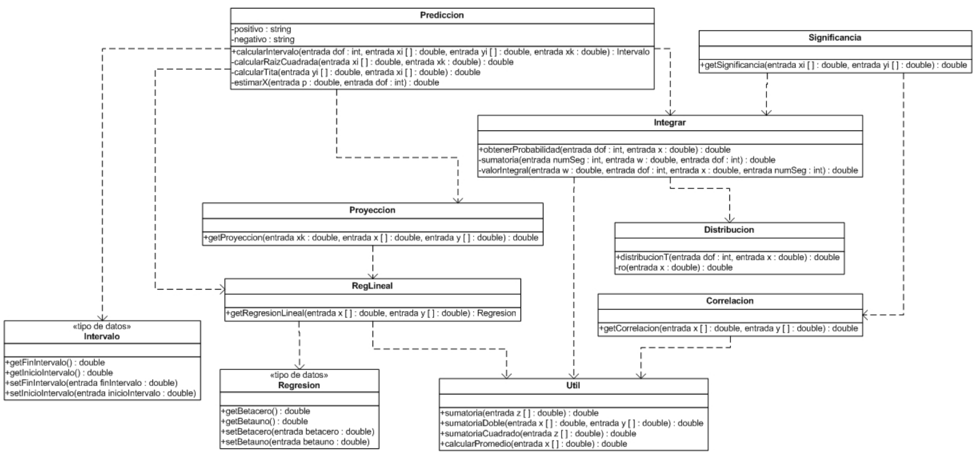

Prediccion
-positivo : string
-negativo : string
+calcularIntervalo(entrada dof : int, entrada xi [ ] : double, entrada yi [ ] : double, entrada xk : double) : Intervalo
-calcularRaizCuadrada(entrada xi [ ] : double, entrada xk : double) : double
-calcularTita(entrada yi [ ] : double, entrada xi [ ] : double) : double
-estimarX(entrada p : double, entrada dof : int) : double
Significancia
+getSignificancia(entrada xi [ ] : double, entrada yi [ ] : double) : double
Integrar
+obtenerProbabilidad(entrada dof : int, entrada x : double) : double
-sumatoria(entrada numSeg : int, entrada w : double, entrada dof : int) : double
-valorIntegral(entrada w : double, entrada dof : int, entrada x : double, entrada numSeg : int) : double
Proyeccion
+getProyeccion(entrada xk : double, entrada x [ ] : double, entrada y [ ] : double) : double
Distribucion
+distribucionT(entrada dof : int, entrada x : double) : double
-ro(entrada x : double) : double
RegLineal
+getRegresionLineal(entrada x [ ] : double, entrada y [ ] : double) : Regresion
Correlacion
+getCorrelacion(entrada x [ ] : double, entrada y [ ] : double) : double
«tipo de datos»
Intervalo
+getFinIntervalo() : double
+getInicioIntervalo() : double
+setFinIntervalo(entrada finIntervalo : double)
+setInicioIntervalo(entrada inicioIntervalo : double)
«tipo de datos»
Regresion
+getBetacero() : double
+getBetauno() : double
+setBetacero(entrada betacero : double)
+setBetauno(entrada betauno : double)
Util
+sumatoria(entrada z [ ] : double) : double
+sumatoriaDoble(entrada x [ ] : double, entrada y [ ] : double) : double
+sumatoriaCuadrado(entrada z [ ] : double) : double
+calcularPromedio(entrada x [ ] : double) : double

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

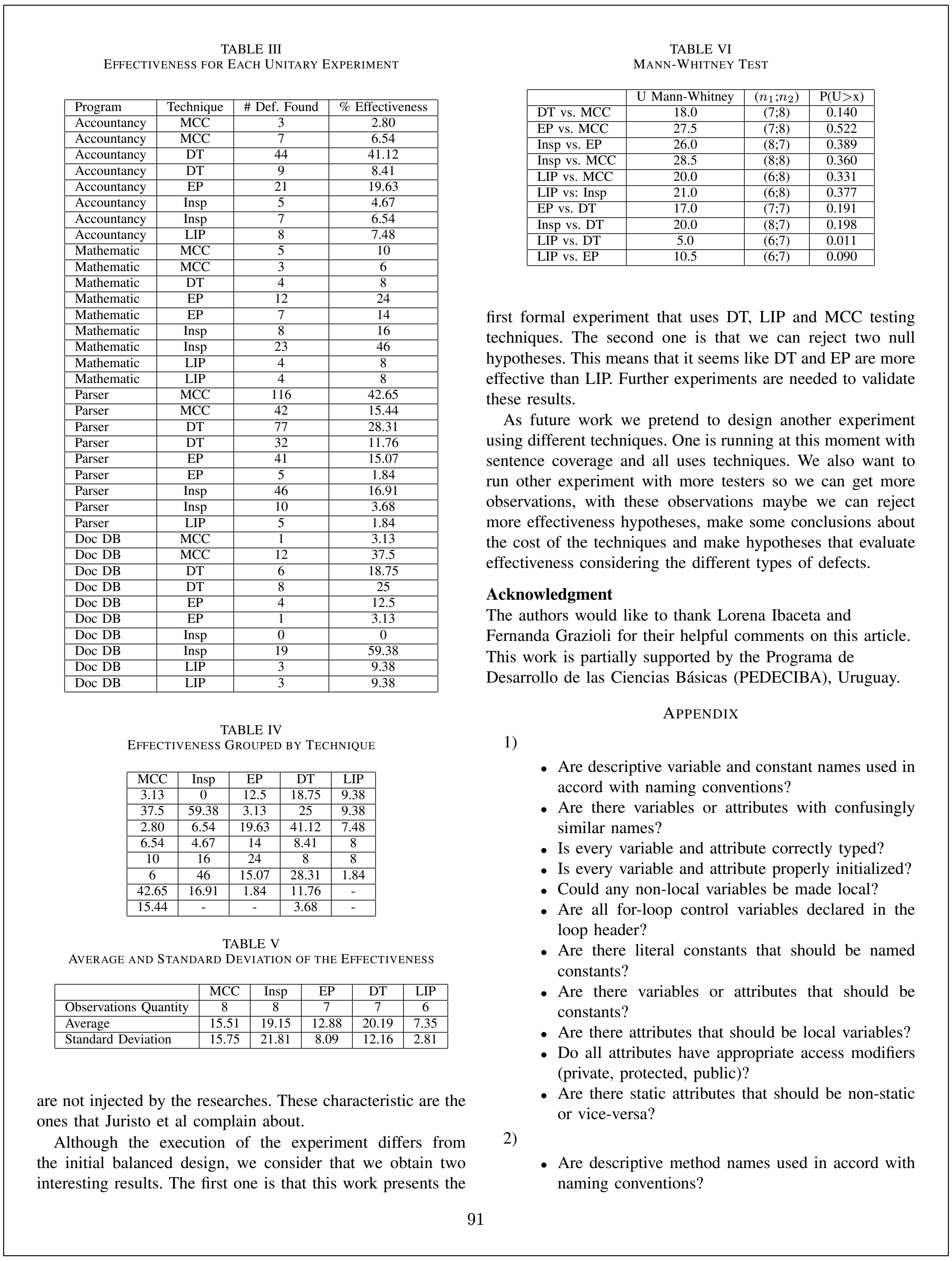


TABLE III
EFFECTIVENESS FOR EACH UNITARY EXPERIMENT

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

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

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.

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.

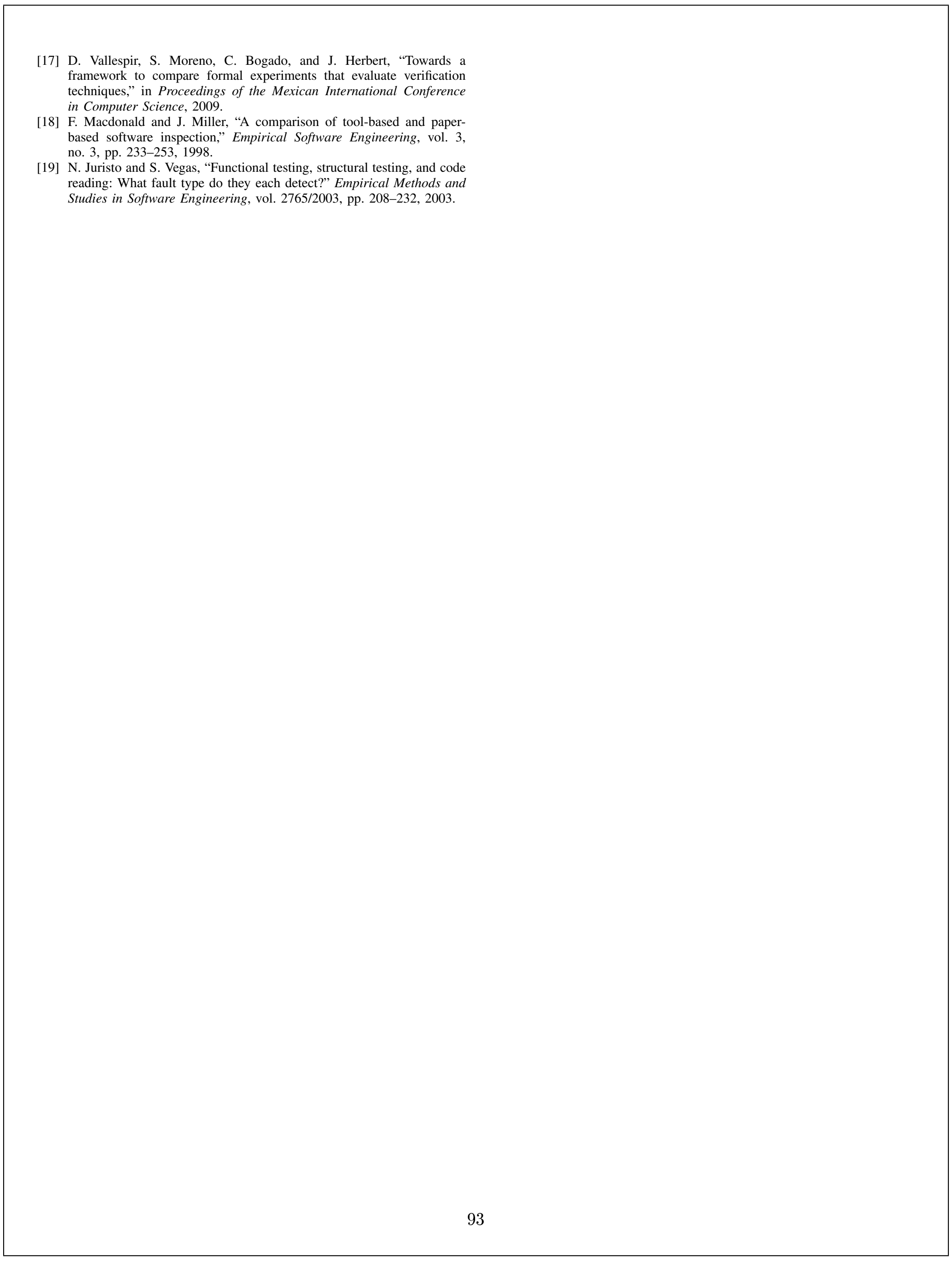
[12] IEEE/ACM, *Software Engineering Body of Knowledge: Iron Man Version*, May 2004.

[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.

ARTICLE

All Uses and Statement Coverage: A controlled experiment

Diego Vallespir, Silvana Moreno, Carmen Bogado and Juliana Herbert

Submitted to a Software Engineering Open Journal. The format is not the Journal format.

All Uses and Statement Coverage
A controlled experiment

Diego Vallespir[1], Silvana Moreno[1],
Carmen Bogado[1], and Juliana Herbert[2]

[1] School of Engineering, Universidad de la República
Julio Herrera y Reissig 565
Montevideo, 11.300, Uruguay
dvallesp@fing.edu.uy,smoreno@fing.edu.uy,cmbogado@gmail.com
[2] Herbert Consulting
Porto Alegre, Brazil
juliana@herbertconsulting.com

Abstract. This article presents a controlled experiment that compares the behavior of the testing techniques Statement Coverage and All Uses. The design of this experiment is typical for a factor with two alternatives. A total of 14 subjects carry out tests on a single program. The results indicate that there is enough statistical evidence to state that the cost of executing All Uses is higher than that of executing Statement Coverage; result that we expected to find. However, no statistical differences were found as regards the effectiveness of the techniques.

Keywords: Empirical Software Engineering, Testing techniques, Test effectiveness, Test cost

1 Introduction

Software unit testing is strongly established in industry. However, the effectiveness and cost of each different unit testing technique is not known with certainty. This makes the decision of which technique to use hardly trivial.

Many years of empirical research have gone by and yet final results have not been achieved. *In A look at 25 years of data* the authors examine in depth different experiments of software testing reaching the same conclusion [8].

A series of formal experiments are currently being carried out at the Computer Science Institute of the School of Engineering of the Universidad de la República in Uruguay in order to gather more precise data in this direction. Four experiments have finished at present and this article describes one of them. The results of other experiments of this series are included in [10–12].

The experiment hereby described compares the unit testing techniques All Uses (AU) and Statement Coverage (SC) in order to know its cost and its effectiveness. The cost is defined as the time it takes to develop the test cases in order to comply with the coverage demanded by the technique. Effectiveness is defined as the number of defects encountered when executing the technique divided by the number of total defects of the program being tested.

A group of undergraduate students of Computer Science of the Universidad de la República in Uruguay generate test cases using SC and AU to execute a program written in Java. These students record the defects found as well as the time employed in the development of the test cases. We analyzed the effectiveness and the cost of each subject and we suggested hypothesis tests to find out whether among the techniques used there are differences of effectiveness and/or cost.

The rest of the article is organized in the following way: section 2 describes the techniques that were used in the experiment. Section 3 presents the related work. The experiment set up is presented in section 4. The results of the experiment are presented in section 5 and the discussion in section 6. The most important threats to validity are presented in section 7. The conclusions and future work are presented in section 8.

2 Background: Sentence Coverage and All Uses

Two testing techniques, both white-box, are employed. SC that is based on control flow and AU that is based on data flow.

In order to satisfy the prescription of SC technique each sentence of the source code must be executed at least once when running the tests. Since this technique is widely known, we do not go deeper into it in this article.

The AU technique expresses the coverage of testing in terms of the definition-use associations of the program. A *definition* of a variable occurs when a value is stored in the variable ($x := 7$). An *use* of a variable occurs when it is read (or uses) the value of that variable. This can be either a p-use or a c-use. A *p-use* is the use of a variable in a bifurcation of the code (*if (x==7)*). A *c-use* is when the use is not in a bifurcation. For example, in *(y := 7 + x)* there is a c-use of x (there is also a definition of y).

The control flow graph is a representation through a graph of the different execution paths that can be taken by a program. The nodes of the graph represent the statements (or code blocks) and the edges the bifurcations (if, for, while, etc.). We will use i_j to refer to a particular node in the graph.

Then, an execution path of the program can be represented as a sequence of nodes. For example, i_1, i_4, i_7 represent an execution path where node i_1 is executed first, then node i_4 and finally node i_7.

A definition of a variable x in a node i_d *achieves* a use of the same variable in a node i_u, if there is a definition clear path from i_d to i_u in the control flow graph and the path is executable. A path i_1, i_2,…,i_n is a *definition clear path* for a variable x if the variable x is not defined in the intermediate nodes of the path (i_2,…,i_{n-1}).

AU requires that at least one definition clear path be executed from each definition (of every variable) to each achievable use (of the same variable).

The classical definitions of the techniques based on data flow and particularly AU are presented in an article by Rapss and Weyuker [9].

In Object Oriented languages the basic testing unit is the class. It is necessary to test its methods in an individual and in a collective way, so as to test the interactions generated through the sequence of calls originated by the invocation of a particular method. AU can be applied both for the tests of individual method belonging to a class and for the methods that interact with other methods of the same class or of other classes.

The tests of a class in AU can be carried out in two levels: Intra-method (Intra) and Inter-method (Inter). In **Intra**, only the method under test is considered for the code coverage. Therefore, in this case, the methods that interact with the method under test are not considered at the moment of developing the test cases. On the other hand, in **Inter**, the methods that interact with the method under test are considered for the code coverage too.

Two types of definition-use pairs to be tested are identified in relation to the levels presented previously. The **Intra-method Pairs** are those which take place in individual methods and test the data flow limited to such methods. Both definition and use belong to the method under test.

The **Inter-method Pairs** occur when there is interaction between methods. They are pairs where the definition belongs to a method and the corresponding use is located in another method that belongs to the chain of invocations.

In most of the literature that presents techniques based on data flow, the examples that are given contain simple variables such as integers and Booleans. However, criteria that normally are not treated should be defined at the moment of applying these techniques in arrays or even more difficult in objects.

Establishing these criteria is essential in order to know under which conditions the technique is applied. Different conditions can produce different results in the effectiveness and cost of AU since in fact, they are different techniques with the same name. Many of these conditions refer to how the Inter-method Pairs should be considered. This experiment establishes the conditions for the application of the AU technique based on what is proposed in [3–5].

3 Related Work

Several formal experiments were carried out in order to find out the effectiveness and/or cost of different unit testing techniques. Some experiments that use techniques based on data flow are presented in this section.

In 1990 Weyuker presents an experiment in order to find out the cost of the testing techniques based on data flow [13]. The cost is measured as the number of test cases generated when applying the technique. The following testing techniques are studied: All c-uses, All p-uses, All Uses and All Paths Definition-Use. The results show that the number of necessary test cases to satisfy those criteria is much lower than the level of the worse case calculated theoretically on a previous work also by Weyuker [14].

Frank and Weiss present an empirical study in which they compare the effectiveness of the All Uses and Decision Coverage techniques [2]. Nine programs are used and random test cases are generated for each of them. No human testers take part in this experiment. Sets of test cases, that meet one or the other criterion, are taken and whether each of these groups detects at least one defect is evaluated. The results show, with 99% confidence, that the All Uses criterion is more effective in 5 out of the 9 programs. In the other 4 programs it is impossible to differentiate.

In 1994, Hutchins and others published an experiment the goal of which is to compare the effectiveness of a variant of the technique All Paths Definition-Use and a variant of the Decision Coverage technique [6]. The experiment has similar characteristics to that of Frank and Weiss. However, in this experiment both test cases automatically generated at random and human testers are used. The results show that there is no statistical evidence indicating that one technique is more effective than the other.

Li and others carry out an experiment to compare four unit testing techniques: Mutants, All Uses, Edge-pair Coverage and Prime Path Coverage [7]. They try to find out the effectiveness (measured as the number of defects detected on the seeded defects) and the cost (measured as the number of test cases it is necessary to generate in order to meet each testing criterion) The cases were generated by hand with the help of tools to know the coverage and another one to generate mutants. The result is that the Mutant technique finds more defects while the other three are similar. Surprisingly (according to the authors) the Mutant technique was the one that required the least test cases.

One of the points that we consider weak in these experiments is that they measure the cost as the number of test cases it is necessary to generate in order to satisfy a certain testing criterion. We believe that the time employed in developing these cases is a more interesting measure for the cost.

4 Experiment Setup

The design of the experiment and its execution are presented in this section.

4.1 Goals, Hypotheses and Metrics

The aim of our experiment is to evaluate and compare the SC and AU techniques concerning their effectiveness and cost. To document our goals, hypothesis and variables we use the GQM approach [1].

Analyze Statement Coverage and All Uses techniques
for the purpose of their evaluation
with respect to their effectiveness and cost
from the viewpoint of the researcher
in the context of a course thought and done especially for this experiment in

the School of Engineering, Universidad de la República of Uruguay.

Since the generic objective is clearly divided in two (effectiveness and cost) we propose different objectives for each of these options. The viewpoint and the context do not change as regards the general objective.

Goal 1:
Analyze Statement Coverage and All Uses techniques
for the purpose of their evaluation
with respect to their effectiveness…

Goal 2:
Analyze Statement Coverage and All Uses techniques
for the purpose of their evaluation
with respect to their cost…

The model to evaluate the effectiveness of each individual is defined as the number of defects found by that individual divided by the number of total defects of the program under test.

$$Effectiveness\,of\,the\,individual = \frac{Number\,of\,defects\,found}{Total\,number\,of\,defects\,of\,the\,program}$$

The model to evaluate the cost of executing a technique is defined as the time in minutes it takes to design and codify the test cases using the technique.

The questions, metrics and hypotheses associated with each goal are the following:

Goal 1:
Q1. Which is the average effectiveness of the individuals when executing a technique?
H1. The individuals that apply All Uses have a better performance than those who apply Statement Coverage as regards the average effectiveness.
M1.1. Number of defects found by each subject.
M1.2. Total number of defects in the program under test.

Goal 2:
Q1. Which is the average cost obtained by individuals when executing a technique?
H1. The individuals that apply All Uses obtain a higher cost than those who apply Statement Coverage as regards the average of the cost.
M2. Total time of design and codification of test cases by each subject.

The **experimental unit** of the experiment is a program written in Java language. Its main feature is payroll accounting. The program is presented in subsection 4.5.

The **factor** is the testing technique. The **alternatives** of this factor are the techniques to be evaluated: SC and AU.

The **response variables** considered in this experiment are the effectiveness and the cost of the techniques.

The **hypotheses** for this experiment are also the traditional in this kind of experiment. The null hypothesis of effectiveness, hypothesis that we want to reject, states that the medians of effectiveness of the techniques are the same. The null hypothesis of cost states that the medians of cost of the techniques is the same. The alternative corresponding hypothesis simply indicates the medians are different.

4.2 Subjects

The subjects of the experiment are graduate students of Computer Science of the Universidad de la República of Uruguay. All of them are advanced students since they are coursing fourth or fifth year. They have passed the course "Programming Workshop" in which Java language is learned and they have completed at least another three courses in Programming and a course in Object Oriented Programming. All of them have completed a Software Engineering course in which, among other things, different testing techniques are studied. We consider, therefore, that the group that participates in the experiment is homogeneous due to the fact that they are at the same level in their studies and that they have been provided with the same training as part of the experiment.

The students participate in the experiment in order to get credits for their studies and this is their motivation. The formal framework is a course in the degree of Computer Science designed especially for the experiment. Attendance to the training is mandatory as well as the execution of the testing technique following the material provided by the researchers. They know that completing the course successfully depends on following the script provided to apply the technique correctly and not on the number of defects found.

The students are not aware that they are taking part in an experiment; they believe they are taking a course with an important component of laboratory practice.

It is the students who enroll in the course. This course is not mandatory for the degree they are taking, consequently their participation is voluntary.

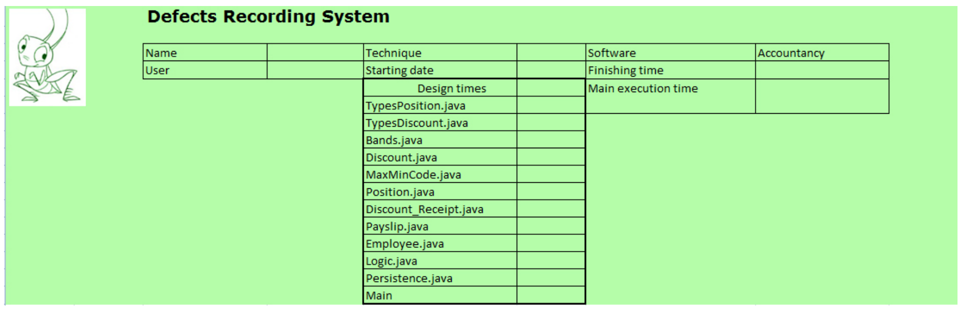
4.3 Experimental Materials

The experiment material consists of the Class Diagram of the program under test, the program user documentation, the program's Javadoc, an electronic spreadsheet to record defects and times, a script to conduct the testing activity and the source code of the program. The script and the program are presented in independent sections.

The java classes of the program must be tested following the bottom-up approach during the experiment. The Class Diagram serves this purpose and it is given to the subjects with a list that specifies the order in which each class has to be tested.

While the test is being conducted the subjects record the defects they find in the program as well as the design and JUnit codification time of the test cases. These data is recorded in the spreadsheet for recording defects and time that was created for this purpose.

The spreadsheet contains two sheets, the first is presented in figure 1. The name of the subject, the name of the program under test, the testing technique being applied, the starting and finishing date, and the design and codification times of the test cases of each Java class are recorded here.



Defects Recording System

Name		Technique		Software	Accountancy
User		Starting date		Finishing time	
		Design times		Main execution time	
		TypesPosition.java			
		TypesDiscount.java			
		Bands.java			
		Discount.java			
		MaxMinCode.java			
		Position.java			
		Discount_Receipt.java			
		Payslip.java			
		Employee.java			
		Logic.java			
		Persistence.java			
		Main			

Fig. 1. Defects and times recording spreadsheet - Sheet 1

In the second sheet, a new row is entered for every defect that is found. The number of the code line where the defect is and its description is recorded for each defect found, as well as the name of the class that contains it. Finally, it is specified whether the defect was found during the design (design and codification of cases) or during the execution of the test cases.

4.4 Testing Script

The subjects use a script to conduct the tests during the experiment. This script is presented and explained to the subjects during the training provided as part of the experiment.

The script is the process that must be followed by the subjects. It is made up of three phases: Preparation, Design and Execution. The complete script can be found in Appendix A.

During the Preparation phase the subject must carry out an initial check-up that guarantees that the tests can be conducted. He must check that he has the source files to test, a file with the class diagram of the program under test, the Javadoc of the classes and the spreadsheet to record defects and times that he will use during the tests.

During the Design phase the subject develops the test cases following the prescriptions of the testing technique previously assigned. The subject must design the test cases following a bottom-up approach. After having designed the test cases, the subject codifies them in JUnit. In this phase, the time it took to design and codify the test cases is recorded in the spreadsheet.

The execution of the test cases is conducted during the Execution phase. The cases must also be executed using a bottom-up approach. The phase ends when there are no test cases that produce a failure. While there are test cases that produce failures the subject must:

– Chose one of the cases that produce a failure.
– Find the defect in the program that produces the failure.
– Record the required data in the spreadsheet for each defect found.
– Request the correction of the defect to the research team.
– Run the test case again to confirm that the correction has been made properly by the research team.

4.5 The Program

The program used in this experiment was built especially for an experiment conducted in 2008 [10]. This program was developed by a fourth-year student of Computer Science. The student developed the program using a specification written by the research team. This specification describes a program to calculate the salaries for educational staff and non-educational employees in a fictitious university.

The functionalities of the program are basic as regards the calculation of salaries. It allows recording employees and their positions. It offers the functionality of reallocating positions, but an employee cannot hold more than one position (be it educational or non-educational). It makes it possible to raise salaries in different ways. It also has a functionality of making statements in which the calculations of all the employees of the system are made generating their corresponding salary slips. It also provides the possibility of creating the salary slip for an employee in particular.

The student must develop the program in Java language taking the specification as a starting point, making sure only that the program could be compiled. That is to say, the student was not allowed to conduct any testing on the developed program. He was also asked not to conduct any type of static review. Therefore the delivered program only had corrections to the defects detected during the compilation.

Therefore, the program contains real defects that were not seeded in the code by the researchers. Besides, since the developer is not allowed to conduct any type of verification, the program is tested for the first time during the experiment, situation which we wanted to simulate.

The student that builds the program generates its documentation using Javadoc. He hands in installation and configuration manuals and a user manual that covers all the functionalities of the program.

The program has a size of 1820 LOCs (without comments) distributed into 14 classes. It is made up of 5 DataTypes, a class that implements the persistence of the data, 5 classes that contain the logic, 2 interfaces and a main class that implements the interface with the user.

The program has a small database (that use the HSQLDB database manager) to support its functionalities. This database is made up by 8 tables. The subjects are given a script for the creation of the tables and load of the basic data.

The defects of the program are not totally known since they are not seeded defects, but they correspond to those that appear during the development. During the 2008 experiment and during the experiment presented in this article, several subjects conducted tests on the program with different techniques. We consider that the number of defects that result from the union of the detected defects in both experiments is a good estimate of the total defects of the program. This number is 187.

4.6 Experiment Design

The design of an experiment corresponds to the design of a factor (testing technique) with two alternatives (SC and AU). The subjects that participated in the execution of the experiment are a subgroup of the subjects that participated in the training; 4 from the first training and 10 from the second. Out of these 14 subjects, 8 apply AU and 6 apply SC.

The subjects in the training could choose freely to participate in the experiment in order to get more academic credits. The final design is not balanced due to the fact that each subject carried out the practice only with one technique during the training.

All the subjects use only one technique (the allotted one: CS or AU) on an only program (the Accounting program).

4.7 Training

The subjects who enroll in the course must go through the training that aims at ensuring that they have the necessary knowledge and practice to use the SC and AU techniques correctly. The training is made up of three parts: JUnit Learning session, Techniques Learning session and Execution session. Figure 2 presents the different activities in each session.

The aim of the JUnit Learning session is for each subject to learn how to use the JUnit tool that they will be using to codify the test cases. Each subject is given a simple program specification and they are asked to implement that specification in Java and to develop a JUnit class to test its functionality. The idea is that the subject should study JUnit individually since this tool is not explained during the training.

This session is done at home by the students and takes a week. Once it finishes, the students hand in the Java class and the JUnit class with the codified test cases. These are reviewed by the researchers in order to verify that the student has acquired the necessary knowledge with JUnit.

Once the JUnit Learning session is finished the Techniques Learning takes place. This session has the aim of allowing the subjects to learn the SC and

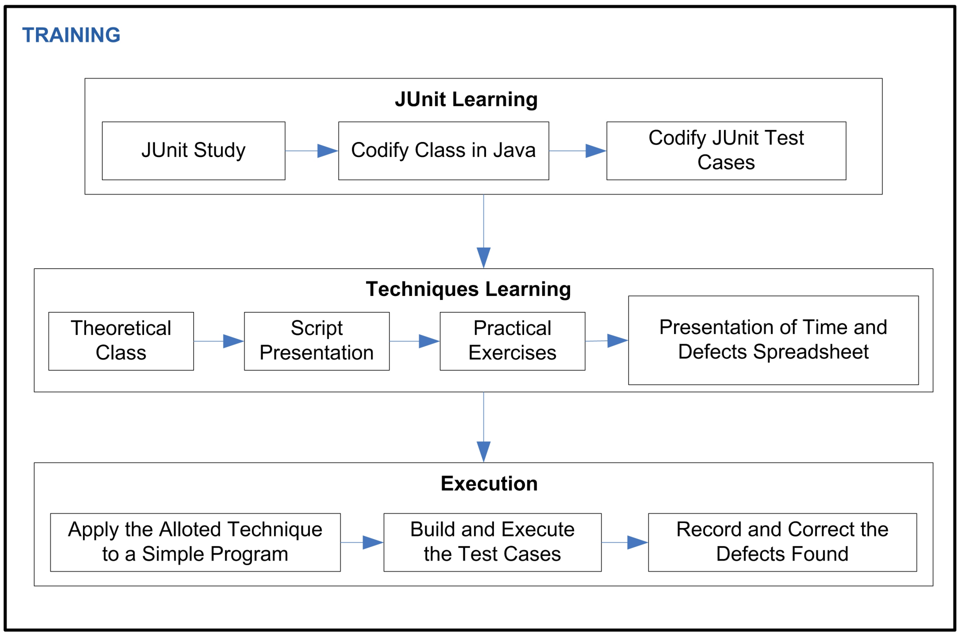

TRAINING
JUnit Learning
JUnit Study
Codify Class in Java
Codify JUnit Test Cases
Techniques Learning
Theoretical Class
Script Presentation
Practical Exercises
Presentation of Time and Defects Spreadsheet
Execution
Apply the Alloted Technique to a Simple Program
Build and Execute the Test Cases
Record and Correct the Defects Found

Fig. 2. Training of the subjects

AU techniques. A theoretical/practical course of 9 hours is conducted during a day. A theoretical class to explain the testing techniques to be used is given in the first half of the day. The verification script to be followed when conducting the tests and the spreadsheet to register times and defects are also explained. During the second half of the day practical exercises to be solved in groups or individually are done. These exercises are an intense practice of the techniques of the experiment.

The Execution session is also conducted during a whole day, seven days apart from the training session. In this session the subjects individually apply the technique allotted to each one, generating the necessary test cases and executing them. This training session is considered also as an experiment in itself [11].

In order to complete the work they follow the script provided in the Techniques Learning session, registering the time and the defects as indicated in the same. The students who are unable to finish their task during the day have a week to finish it. At the end of the session the subjects hand in the JUnit classes generated, the spreadsheet with the registers of time and defects and the notes they have made in order to be able to apply the technique (control flow graphs, identified paths, etc.) Once all the deliveries have been done, the research team reviews them and they are given back individually to each student.

The execution session serves to achieve several objectives. The subjects familiarize with the verification script they must follow, with the techniques and with the spreadsheet to use to keep the records. While the work is reviewed, it is possible to make adjustments and correct the mistakes that the subjects could be making while applying the technique.

4.8 Operation

The complete training takes place twice with two different groups of subjects. The first training and the second were carried out some weeks apart from one another.

10 subjects participated in the first training. 5 used AU technique and the other 5 applied SC. 11 subjects participated in the second training. 6 used AU and 5 used SC. The choice of subjects for the different techniques is at random. Each complete training (the three sessions) took about three weeks.

The experiment with the Accounting program is conducted in 8 one-week sessions. Each session is allotted a certain number of Java classes to be tested by the subjects. The choice of classes to be tested each week is made based on the bottom-up approach and their complexity, and was made by the research team.

The execution of the experiment by the subjects is done at their home. At the beginning of each week the classes to be tested are sent to them via e-mail.

During the week the subjects report the defects found to the researchers requesting their correction. The research team sends the corresponding correction for each defect. The subjects are not allowed to make the corrections by themselves. The purpose of this is that the subjects have the same correction for the same defects.

It is important to point out that we have no knowledge of other experiments conducted in which the defects are corrected. The correction of defects during the experiment simulates better the testing activity in the industry.

Each weekly delivery made by a subject is validated by the research team. The aim is to make sure that the spreadsheet is complete. It is also controlled that the test cases handed in codified in JUnit do not fail, which means that all the defects that produce failure have been corrected.

A form that records the defects of each Java class of the program is completed using the defects found by the different subjects. It is made taking the defects form of the 2008 experiment as a starting point. This form is used to keep all the defects found by all the subjects per class of the program.

5 Results

The results are presented in two parts. The first sub-section considers goal 1: effectiveness of the techniques. The second considers goal 2: cost of the techniques.

5.1 Effectiveness

In this section we present the descriptive statistics and the hypothesis test for the effectiveness of the techniques.

Descriptive Statistics

Figure 3 presents a box and whisker chart of the effectiveness both of the SC technique and the AU technique. The medians of both techniques are similar, for SC it is 11% and for AU it is 10%. The distribution of both techniques is a bit different.

Table 1 presents the number of subjects that used each one of the techniques, the mean and the standard deviation of the effectiveness. Both the average and the standard deviation are similar.

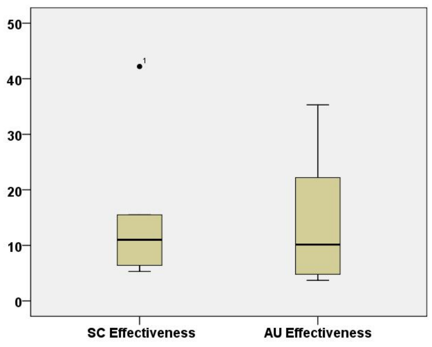


Fig. 3. Box and Whisker of the Effectiveness

	No. of Subjects	Mean	Std. Dev.
SC	6	15.2%	13.9%
AU	8	14.2%	11.6%

Table 1. Mean and Std. Deviation of the Effectiveness

Hypothesis Testing

The hypothesis test done to compare the effectiveness of the techniques is presented below. The quantity of observations we have (6 and 8) are too few to make parametric tests. Therefore, the Mann-Whitney test is applied, since it is suitable for our experiment design. The null hypothesis that indicates that the medians of effectiveness of both techniques are the same is proposed, together with the corresponding alternative hypothesis:

$H0 : Mdn\, ESC = Mdn\, EAU$

$H1 : Mdn\, ESC <> Mdn\, EAU$

The test of Mann-Whitney indicates that it is not possible to reject the null hypothesis. Therefore, we do not find there is a statistical difference between the effectiveness of both techniques.

5.2 Cost

In this section we present the descriptive statistics and the hypothesis test for the cost of the techniques.

Descriptive Statistics

Figure 4 presents a box and a whisker chart of the SC and the AU cost. The median of SC is 38.5 hours and that of AU is 54.1 hours. The distribution of both techniques is very different. Although the minimum values are similar, the percentiles from 25% to 75% are totally different.

Table 2 presents the number of subjects that used each one of the techniques, the mean and the standard deviation of the cost (in hours). The average of design hours to design test cases in AU was 50% higher than that of SC. The standard deviations and the box and whisker chart show that the dispersal of cost in AU is much higher than that of SC.

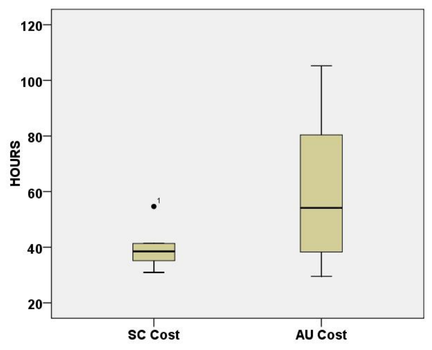


Fig. 4. Box and Whisker of the Cost

	No. of Subjects	Mean	Std. Dev.
SC	6	39.8	8.3
AU	8	60.0	27.2

Table 2. Mean and Std. Deviation of the Cost in Hours

Hypothesis Testing

The Mann Whitney test is used again. The null hypothesis in this case is that the median of the cost of the techniques is the same:

$H0 : Mdn\, CSC = Mdn\, CAU$

$H1 : Mdn\, CSA <> Mdn\, CAU$

The calculated p value is lower than the α chosen level (0.1), therefore we conclude that there is statistical evidence that indicates that AU is more expensive than SC.

6 Discussion

In this experiment we expected to find that the effectiveness of the AU technique is higher than that of SC. However, this could not be proved statistically.

We found that both techniques had a very low effectiveness, the average of SC being 15.2% and that of AU 14.2%. This effectiveness might be because the students did not know how to apply the techniques well. This can become more negative for the more complex technique, in this case AU. We are currently carrying out a study to know, taking the test cases generated by the subjects as a starting point, how far the subjects satisfy the prescription of the technique they use. This study can help discuss the low effectiveness of both techniques in our experiment.

On the other hand, as far as the cost analysis is concerned, we found what we expected to find. The AU technique is more expensive that the SC technique. As we consider the design and codification time of the test cases as the cost of the technique, we can conclude that the AU technique is more complex to apply than SC.

The AU technique presents a greater variability in the cost than the SC technique. It could be thought from a practical point of view that the costs of applying SC are kept more under control than the costs of applying AU.

However, this does not have to be so. Our ongoing study that studies the compliance with the prescription of the technique by the subjects indicates that those who apply SC normally comply with it while those who apply AU have a great variability. This could explain the variability of costs; those who apply AU correctly could be those subjects who have a high application cost. These are assumptions under analysis at the moment.

7 Threats to Validity

There are various threats to the validity of this experiment. These make it important to replicate the experiment in order to know if the conclusions can be generalized. Some of the threats that we consider most important are mentioned below.

Only 14 subjects participated in the experiment. Thus, we can only use non-parametric tests. It is necessary to replicate the experiment with more subjects in order to generalize the findings.

The subjects are all undergraduate students. Although they are all advanced students and they are carefully trained, they are not professionals in software development. This might imply that the test cases generated could be “worse” than those an expert could generate.

Only one program is used in the experiment. Therefore, the obtained results may be due to the program’s special features and not to the techniques studied. It is necessary to replicate the experiment with different types of programs in order to generalize the findings.

The obtained results are only useful for SC and AU. Although in this experiment both techniques show very low effectiveness (about 15%) this cannot be generalized to the rest of white-box techniques. Future replications must consider other white-box techniques and include black-box techniques.

8 Conclusions and Future Work

This article presents and empirical study that aims to find out more about the effectiveness and cost of the SC and AU techniques. As regards the effectiveness, we could not reject the null hypothesis; therefore we cannot say that one technique is more effective than the other.

As far as the cost is concerned, although the subjects taking part in the experiment were few, we can say that the AU technique is more expensive than the SC technique. It is on average 50% more expensive to execute.

It is necessary to make replications in order to generalize the findings due to the different threats to the validity of this experiment. In fact, it would be interesting to increase the number of subjects, vary the testing techniques and have different programs to test.

If future replications of this experiment show the same result, we could say it is convenient, concerning the cost-benefit relation of executing white-box techniques, to use simple white-box techniques. The use of complex techniques such as AU would not be worth while, since their cost is much higher and there are no differences as regards effectiveness.

As far as future work is concerned, it is our intention to replicate this experiment with the same type of subjects but a bigger number of them. This would eliminate one of the threats to the validity that were mentioned. In a longer period we would have more programs to test and finally we would add other testing techniques.

A The Script

The script for white box testing used in the experiment is presented below. The script describes the process to conduct the tests by the subjects. It entails three phases: Preparation, Design and Execution.

Script for white box testing

Step	Phases	Description
1	Preparation	– Preparation activities for testing.
2	Design	– Design the test cases that satisfy the prescription of the technique. – Build the test cases in JUnit. – Record the required data of the phase.
3	Execution	– Execute the designed test cases. – Find the defects associated to the cases that produce failures. – Record the required data of the phase.

Script for the Preparation Phase

Preparation Phase: Carry out an initial check-out to guarantee that verification can be done.

Step	Activities	Description
1	Verify Files	– Verify that the source files are available. – Verify that you have the Class Diagram. – Verify that you have the Javadoc of the classes to test. – Verify that you have the defect and times spreadsheet.

Script for the Design Phase

Design Phase: Make the design of the test cases satisfying the prescription of the testing technique to apply. The cases must be designed according to the bottom-up approach.

Step	Activities	Description
1	Define the data test set	– Record the starting time of the activity. – Define for each method the set of input values that satisfy the prescription of the technique.
2	Define the expected results	– Define the expected result (or expected behavior) for each element of the set of input values and thus make the test cases.
3	Debuggin	– Eliminate the test cases that cannot be executed (imposible paths, etc.). Check if the prescription is still satisfied. In case it is not go back to step 1 trying to satisfy it.
4	Codification of test cases in JUnit	– Codify all the designed test cases in JUnit.
5	Record of finishing	– Record the design finishing time

Script for the Execution Phase

Execution Phase: Carry out the execution of the designed test cases. The cases must be executed following the bottom-up approach.

Step	Activities	Description
1	Execute the test cases	– Execute the test cases
2	Analyze the obtained output	– If there was no failure the phase is finished
3	Find defects	– While there are test cases that produce a failure: • Chose one of the cases that produce a failure. • Find the defect in the program that produces the failure. • Execute step 4. • Request the correction of the defect to the research team. • Run the test case again to confirm that the correction has been made properly by the research team.
4	Defect recording	– For each defect found record the following data: • General description of the defect. It is important that the description is clear and accurate. • Name of the file that contains the defect. • Line in which the defect is. In case the defect is not on a specific line record 0 (zero). Beware: if the file that is being tested has been modified compared to the original (because some defect has been corrected), the line of the original file must be indicated. • Structure associated to the defect (i.e. IF,FOR, WHILE, name of the method, etc).If the defect has no associated line this field must be completed necessarily. • Starting line of the structure (mandatory if an associated structure is indicated).

References

1. Basili, V., Caldiera, G., Rombach, H.: Goal question metric approach. Encyclopedia of Software Engineering pp. 528–532 (1994)
2. Frankl, P.G., Weiss, S.N.: An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transactions on Software Engineering 19(8), 774–787 (1993)
3. Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering 14(10), 1483–1498 (1988)
4. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. SIGSOFT Softw. Eng. Notes 19(5), 154–163 (1994)
5. Harrold, M.J., Soffa, M.L.: Interprocedual data flow testing. In: TAV3: Proceedings of the ACM SIGSOFT '89 third Symposium on Software Testing, Analysis, and Verification. pp. 158–167. ACM, New York, NY, USA (1989)
6. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on. pp. 191 –200 (16-21 1994)
7. Li, N., Praphamontripong, U., Offutt, J.: An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on. pp. 220 –229 (1-4 2009)
8. Moreno, A., Shull, F., Juristo, N., Vegas, S.: A look at 25 years of data. IEEE Software 26(1), 15–17 (Jan–Feb 2009)
9. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: ICSE '82: Proceedings of the 6th international conference on Software engineering. pp. 272–278. IEEE Computer Society Press, Los Alamitos, CA, USA (1982)
10. Vallespir, D., Apa, C., De Len, S., Robaina, R., Herbert, J.: Effectiveness of five verification techniques. In: Proceedings of the XXVIII International Conference of the Chilean Computer Society (2009)
11. Vallespir, D., Bogado, C., Moreno, S., Herbert, J.: Comparing verification techniques: All uses and statement coverage. In: Ibero-American Symposium on Software Engineering and Knowledge Engineering. pp. 85–95 (2010)
12. Vallespir, D., Herbert, J.: Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. In: Computer Science (ENC), 2009 Mexican International Conference on. pp. 264–271 (2009)
13. Weyuker, E.: The cost of data flow testing: An empirical study. IEEE Transactions on Software Engineering 16, 121–128 (1990)
14. Weyuker, E.J.: The complexity of data flow criteria for test data selection. Information Processing Letters 19(2), 103–109 (1984)

Chapter 5

Conclusions

In this Part the effectiveness and cost of the different testing techniques were analyzed. In order to do this, two controlled experiments executed in the School of Engineering of the Universidad de la República in Uruguay were conducted during 2008 and 2009.

The techniques used in the experiment were: desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, multiple condition coverage, sentence coverage and all uses. These techniques combine static and dynamic techniques, white and black box techniques, as well as techniques based on control flow and data flow.

The subject attended training sessions as part of the experiment. The aim of the training was that the subjects should learn the techniques and apply them correctly. The final part of the training consisted in applying the techniques on a small program. That is why the training could be analyzed as an experiment in itself. After the training the subjects applied the allotted technique on larger programs.

The effectiveness of all the techniques was low. None of the techniques was more than 35% effective in any of the experiments (2008 and 2009) considering both the application of the techniques on the small program and on the big ones.

The effectiveness of all the techniques was lower when they were applied on the programs of the experiment than when they were applied on the training program. This is surely due to the fact that it is easier to find defects in trivial programs than in complex ones. Conducting other experiments combining a greater number of trivial programs with more complex ones will provide us with more information to enable us to reject or accept this hypothesis.

In the Experiment 2009 we used only one program and each subject applied only one technique. This differs from the design of the Experiment 2008 in which each subject applied three techniques on three different programs and a total of four programs were used. The 2008 experiment was the first controlled experiment in Software Engineering conducted in our research team. The experiment lasted a little more than a year and it was exhausting both for the subjects and the research team. The changes in the design for the Experiment 2009 respond to a learning process in conducting controlled experiments.

5.1 Relation to Other Results

In "Reviewing 25 years of testing technique experiments" [1] and in "A look at 25 years of data" [2] the authors study the experiments conducted to evaluate testing techniques (up to the date of the articles). These two works present the existing empirical knowledge on the topic, problems found and research opportunities in the area. We shall discuss the results of our experiment in the light of these two articles.

In these two articles the authors detect that the majority of the software programs used in the experiments "suffer from at least one of two problems: they are small and unusually simple, or the researchers seeded the defects rather than looking for naturally occurring ones". In our experiments we use 4 programs each of which is developed especially for the experiments. 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. This way we try to solve the two problems mentioned.

The authors are more explicit as regards what the programs used in the experiments are like: "They [the researchers] use relatively small programs, between 150 and 350 LOC, which are generally toy programs and might not be representative of industrial software". The four programs used in our experiments have the following number of lines of code: 468, 566, 828 and 1820. We believe that our programs are suitable for conducting unit testing experiments as far as their size and complexity are concerned.

The authors also detect that "it seems that some types of faults are not well suited to some testing techniques". The trainings of the 2008 and 2009 Experiments show that the dynamic techniques have problems in finding NF defects (defects that do not cause failures). Using EP and DT the participants did not discover any of the NF defects. With MCC, AU and SC only one NF defect is found. The reason for this is that dynamic techniques are based on 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 produces the failure; in this review the tester can find other defects, including a NF defect. Furthermore, in the white box techniques the testers review the code thoroughly to develop a group of test cases that satisfy the coverage that the assigned technique requires; this is an opportunity to find defects. What this experiment shows is that the attention of the subjects concentrates only on the task they are performing. It is difficult for them to detect defects while they are reviewing the code for developing test cases using white box techniques and it is also difficult for them to find a defect that is not provoking the failure that they are trying to solve.

In the 2009 Experiment we did not find statistically valid differences between the AU technique and the SC technique. The authors state in the articles mentioned that "there does not appear to be a difference between all-uses and all-edges testing as regards effectiveness from the statistical viewpoint, as the number of programs in which one comes out on top of the other is not statistically significant". Given that all-edges is a more demanding technique than SC it is reasonable to expect that if we did not find statistically valid differences between the effectiveness of AU and SC, differences will not be found between AU and all-edges either.

The authors also mention that from a practical point of view "all-edges is easier to satisfy than AU". Our results indicate with statistical validity that the cost of using AU is higher than the cost of using SC (50% more expensive). These results are consistent because in theory all-edges is more expensive to execute than SC (however, we have not found experiments to validate this).

The authors also state that the "boundary analysis technique appears to behave differently compared with different structural testing techniques (particularly, sentence coverage and condition coverage)". In the Experiment 2008, which uses the EP technique (equivalence partition and boundary-value analysis), we do not find statistical evidence in the effectiveness between EP and the structural MCC technique (it is worth mentioning that it is the first time the MCC technique has been used in an experiment.)

However, we did find statistical difference in the effectiveness of EP and the structural LIP technique; EP being more effective than LIP. It must be borne in mind that the LIP technique was ruled out from the results analysis of the training of the Experiment 2008 due to the fact that the subjects probably used it in an incorrect way. In spite of the fact that the LIP technique was explained again to the subjects to whom it was assigned, we are uncertain as to how it was used in the 4 programs of the experiment. Knowing if LIP was used properly requires an analysis of the data that has not been done yet.

In the Experiment 2008 we used DT (the other black box technique used). In this experiment there was no statistically valid difference between the effectiveness of DT and the effectiveness of EP. However, the effectiveness averages did present differences. DT was more effective than EP, both in the training and in the experiment.

We have not found formal experiments that use DT in literature. Since we obtained differences in the effectiveness averages between DT and EP it seems interesting that the research community should conduct formal experiments that compare both techniques or even resort to using DT instead of EP in their experiments as a "representative" technique of the black box techniques.

In the two experiments we conducted we found a high variability between the subjects that applied the same technique in the same program. This is consistent with what was found in the review of 25 years of experiments: "There appears to be a dependence on the subject as regards technique application time".

It is important to point out that we have no knowledge of other experiments conducted in which the defects are corrected. The correction of defects by the researchers during the experiment simulates better the testing activity in the industry.

5.2 Contributions of the Research

Conducting formal experiments contributes to the empirical knowledge of a certain area. Our experiments make a contribution in the area of empirical software engineering and more specifically in the area of formal experiments that try to evaluate software unit testing techniques.

We also made a contribution studying three testing techniques that had not been studied before through formal experiments: MCC, LIP y DT.

The construction of 4 more complex programs than those used habitually in the formal experiments of the area and having real defects (as opposed to defects injected by the researchers) is a contribution to the material to conduct formal experiments.

The way the experiment is conducted, making the correction of the defects during the execution of the tests, is a novelty that can be incorporated by the researchers of the field. It simulates the reality of software unit testing in industry better. However, it is clear that this way of conducting experiments requires more effort from the research team.

References

[1] Natalia Juristo, Ana M. Moreno, Sira Vegas. 2004. Reviewing 25 Years of Testing Technique Experiments. Empirical Software Engeneering, 9, 1-2 (March 2004), 7-44.

[2] Natalia Juristo, Ana M. Moreno, Sira Vegas, Forrest Shull. 2009. A Look at 25 Years of Data. IEEE Software, 26, 1 (January 2009), 15-17.

Part II

Methodological Recommendations for Controlled Experiments that Evaluate Software Testing Techniques

Chapter 6

Introduction

In this Part of the thesis entitled “Methodological Recommendations for Controlled Experiments that Evaluate Software Testing Techniques” we present a series of articles that complement in different ways the main line of research of this thesis: the study of the effectiveness and the cost of different testing techniques. In this introduction we show the relation between these complementary works and the central investigation presented in Part I.

This part includes an article that presents and evaluates different taxonomies of software defects: “A framework to Evaluate Defect Taxonomies”. The evaluation is done by means of a Comparison Framework we developed as part of our research.

Several experiments that aim at knowing the effectiveness of testing techniques segment the results by type of defect. They try to find out how effective a certain technique is for each type of defect. The article presented in Part I, “Effectiveness of Five Verification Techniques” explains that ODC taxonomy and the one proposed by Beizer are used to classify the defects. However, due to the small number of observations the effectiveness data discriminated by type of defect could not be analyzed. It is convenient to know different taxonomies and to evaluate them to decide which to use in future experiments.

Different research groups in different places in the world conduct formal experiments in software engineering. In particular, many experiments aim at knowing the cost and effectiveness of different testing techniques. Normally these experiments are very difficult to compare or to add. The article “Towards a Framework to Compare Formal Experiments” included here presents a primary construction of a framework to compare, but above all to identify, the most important items of a formal experiment the intention of which is to evaluate different testing techniques.

The next article, “Construction of a Laboratory Package for a Software Engineering Experiment”, presents a packaging experience of the Experiment 2008. Packaging an experiment provides the possibility, among other things, of replicating the experiment easily by different groups of researchers (or more easily than if it was not packaged). Besides, packaging is also important since it provides with complete information of the experiment that has been conducted. It is important to clarify that in the packaging of the 2008 Experiment I only collaborated supplying information about the experiment and reviewing the article mentioned before. In other words, the main research work was carried out by the rest of the authors and not by me.

The article "Calidad de los Datos en Experimentos en Ingeniería de Software. Un Caso de Estudio" [Quality of the data in Software Engineering Experiments. A Case Study] presents a study on the quality of the data of the Experiment 2008. This article is presented in Spanish.

Chapter 7

Methodological Recommendations for Controlled Experiments that Evaluate Software Testing Techniques

The articles included in this chapter are:

A Framework to Evaluate Defect Taxonomies
Diego Vallespir, Fernanda Grazioli and Juliana Herbert
Proceedings of the Argentine Congress of Computer Science 2009, pp. 643-652, San Salvador de Jujuy, Argentina, October, 2009.

Towards a Framework to Compare Formal Experiments that Evaluate Testing Techniques
Diego Vallespir, Silvana Moreno, Carmen Bogado and Juliana Herbert
Research in Computing Science, pp. 69-80, ISSN 1870-4069, 2009.

Construction of a Laboratory Package for a Software Engineering Experiment
Cecilia Apa, Martín Solari, Diego Vallespir and Sira Vegas
Proceedings of the Ibero-American Conference on Software Engineering, pp. 101-114, Rio de Janeiro, Brazil, April 2011.

Calidad de Datos en Experimentos en Ingeniería de Software: Un Caso de Estudio
Carolina Valverde, Adriana Marotta and Diego Vallespir
Submitted to a Conference. Article in Spanish.

ARTICLE

A Framework to Evaluate Defect Taxonomies

Diego Vallespir, Fernanda Grazioli and Juliana Herbert

Proceedings of the Argentine Congress of Computer Science 2009, pp. 643-652, San Salvador de Jujuy, Argentina, October, 2009.

A Framework to Evaluate Defect Taxonomies

Diego Vallespir[1], Fernanda Grazioli[1], and Juliana Herbert[2]

[1] Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Montevideo, Uruguay.
[2] Herbert Consulting
Porto Alegre, RS, Brazil.
dvallesp@fing.edu.uy , fgrazioli@adinet.com.uy,
juliana@herbertconsulting.com

Abstract. This paper presents a framework for evaluate and compare different defect taxonomies. Six well-known taxonomies are evaluated with it and the results are showed. We found deficiencies in every taxonomy once they are evaluated with the framework.

Key words: Software engineering, Software verification and validation, Defect taxonomies

1 Introduction

There are different reasons for using defect taxonomies. In *software development* they can be used to know which defect types are normally injected, improving both the verification activity and the software development process. Defect taxonomies have been used in many ways in *empiric software engineering* (ESE).

Knowing what defect types are injected, allows to look for them in a particular way. So, the verification activity could take less time and find more defects. For example, if 90% of the defects are in the requirements specification, it is possible to thoroughly verify this specification and reduce the time used in verifying other software products. In other words, to guide the verification considering the knowledge about the defects that are injected.

Also, the classification of defects gives an idea of the phases, activities, disciplines, etc. of the development process where most of the defects are injected. With this data the software development process can be improved by reducing the quantity of defects injected during the development.

For example, in ESE the defect taxonomies have been used to study the impact that different verification techniques have on different defect types. The goal is to know if different verification techniques find different defect types. By knowing this it can be possible to optimize the combination of different verification techniques in order to maximize the number of defects found.

Not only these reasons but many others that have not been exposed here show the importance of defect taxonomies in software development and in ESE. Unfortunately, there is not an universally used taxonomy, neither in software development nor in ESE. This situation causes a lot of problems, for example, it is difficult or even impossible to compare some results between researchers.

A wide variety of taxonomies exists on the industry and in the literature. It is important to compare them from different points of view, trying to identify strengths and weaknesses of each one.

In this paper we analyze the most important taxonomies we found: Hewlett-Packard Taxonomy [1], Kaner, Falk and Nguyen's Taxonomy [2], Robert Binder's Taxonomy [3], IEEE Standard Classification for Software Anomalies [4], Orthogonal Defect Classification [5] and Beizer's Taxonomy [6]. Due to space reasons the taxonomies are not presented so we must assume they are known.

The paper is organized as follows. Section 2 presents a meta-taxonomy. A framework for the taxonomies comparison is presented in section 3. The taxonomies' comparison is presented in section 4. Section 5 presents the conclusions and the future work.

2 Meta Taxonomy

There is a lack of literature on meta taxonomies. We developed our comparison framework based on a previous work of Freimut [7]. From section 2.1 to 2.3 the Freimut proposal is presented. This proposal is divided in different aspects: attributes[3], structure types and properties.

Section 2.4 presents a taxonomy classification proposed in [8].

2.1 Attributes

A taxonomy is composed of attributes. Each attribute has a set of possible values. The values represent defect characteristics that must be registered at the classification moment.

The attributes to be considered must be those that are relevant to a future analysis. The attributes proposed by Freimut are listed below.

Location It refers to where the defect is located. The amount of detail in the specification may vary either indicating the system's high level entity on which the defect is found (Design, Code) or indicating the name of the software product it belongs to.

Timing It refers to the project's phase in which the injection, detection and/or correction of the defect is produced.

Symptom It refers to what it is observed when a failure is generated or to a description of the activity that is causing the failure.

End Result It refers to the description of the failure caused by the defect. This attribute usually contains values such as performance and usability among others.

Mechanism It refers to how the defect was injected, detected and/or corrected.

[3] In the original they are called *key elements*

Cause (Error) It refers to the human error that caused the injection of the defect (communication, lack of time and others).
Severity It refers to how serious the defect or failure is.
Cost It refers to the time and effort devoted to finding and/or correcting a defect.

Sometimes, the difference between symptom and end result is not clear. Symptom refers to what it is observed, maybe without knowing what is really happening, when a failure occurs. On the other hand, end result refers to what it really happens when the failure occurs.

For example, a simple error message could be a symptom of a failure. Once the defect (that causes the failure) is found it can be known the real extent of the effects of the failure (end result). For example, the database is corrupted and a error message is shown (the same one as in the symptom).

2.2 Structure Types

The existing relationships between the attributes in a taxonomy determine its structure type.

One of the possible structures is a hierarchy. The values of an attribute in a certain level are refined by values of attributes in the next level of the hierarchy. An example is Beizer's taxonomy [6].

A tree is another structure in which the attributes are not independent. The choice of a value in an attribute determines the possible values of the following attributes.

Another possible structure is the orthogonal in which values of each attribute are assigned independently of the other attributes. An example of this is the ODC taxonomy [5].

Another structure is the semi-orthogonal. There is no dependence among some attributes while among others the choice of a value determines the possible values of another attribute. The HP taxonomy [1] is an example of this.

2.3 Properties

If a taxonomy is not properly defined, problems may arise in order to get correct, reliable and good quality results. Therefore, the analysis of the results may be questionable. To avoid these problems, it is desired that taxonomies include the following requirements and properties.

Mutually exclusive attribute values When choosing the value of an attribute, only one value is appropriate.
Orthogonal attributes As it was previously explained, orthogonality refers to the independence between attributes.
Complete attribute values The set of values must be complete for each attribute. This means that when classifying an attribute for a given defect it is possible to choose an appropriate value for it.

Small number of attribute values The set of possible values for each attribute should not be very big. Having a small set of values not only makes the classification process simpler but also makes it less probable of mistakes. However, if the taxonomy is meant for a thorough analysis, it is possible that a bigger set of values may be needed in order to achieve an adequate precision in the results.

Description of attribute values It is important that all the possible values are defined clearly and stated with examples of defects that classified under that value.

2.4 Taxonomy Classification

In [8], it is presented a classification of what they call defect classification systems. These are divided in three categories: defect taxonomy, root cause analysis and defect classification.

The defect taxonomies are defects types categorizations. An example is the Beizer's taxonomy.

In root cause analysis not only defects are analyzed, but also their cause. The main goal is to identify the roots of these causes and eliminate them in order to avoid more defects. This approach is considered rather elaborated, so the cost/benefit relation is not clear. There are proposals for this in [9] and [10].

The defect classification uses multiple dimensions to classify defects. The goal is to reduce the costs and maintain the benefits of root cause analysis. Examples of this are ODC, HP and IEEE taxonomies.

In this paper we use the term taxonomy for any of these classification systems.

3 Comparison Framework

In this section it is presented our comparison framework. The best way to evaluate taxonomies in a correct and coherent way is to build a framework in which the characteristics, strengths and deficiencies of each taxonomy can be objectively and evenly evaluated.

Freimut also made an interesting comparison since he evaluates the attributes of various taxonomies, including HP, IEEE and ODC. However, as taxonomies can also be evaluated by their properties it is not enough to achieve our goal. Besides, this comparison does not include all the taxonomies evaluated here nor is completely coherent with its definitions.

In order to reach the objective of carrying out a complete and correct theoretical balance, we propose the development of a comparison framework. The framework will consist of two views: Attribute and Property.

The **Attribute view** takes Freimut's proposal as the main idea, which is modified in the framework by the addition of more specific attributes. The target of this view is to evaluate the capacity of a taxonomy to describe defects. The description of each attribute of the framework is as follows. The original name of those which derive from Freimut's proposal appears between brackets at the end of the description. Otherwise appears "new".

Suspected location It refers to the place where the defect is suspected to be. (Location)

Real location It refers to the place where the defect really is. (Location)

Failure's causing phase It refers to the project phase in which occurs the failure. (Timing)

Injection's phase It refers to the project phase in which the defect is injected. (Timing)

Symptom It refers to what is observed when the failure occurs. (Symptom)

Type It refers to the defect type that was detected. (New)

Failure's causing mechanism It refers to the activity that drives to the failure. (Mechanism)

Injection's mechanism It refers to how the defect was injected. (Mechanism)

Correction's mechanism It refers to the activity that corrected the defect. (Mechanism)

Failure's impact It refers to the impact the failure has on the product when the failure occurs. (End result)

Defect's impact It refers to the impact the defect has on the project. (End result)

Cause (Error) It refers to the human error that caused the defect's injection. (Cause)

Responsible source It refers to where the product (in which the defect was injected) was developed (In house, external, etc.). (New)

Re-correction It refers to whether or not the defect was introduced when fixing a previous defect. (New)

Occurrence's Probability It refers to the estimated odds in favour of the failure to occur. (New)

Severity It refers to how serious the failure is. (Severity)

Priority It refers to how fast the defect has to be corrected. (New)

Detection's cost It refers to the time consumed in the detection of the defect after the failure appeared. (Cost)

Estimated cost of correction It refers to the estimated time its correction will take. (Cost)

Real cost of correction It refers to the time consumed in the defect's correction. (Cost)

The **Property view**, is based on the desired properties proposed by Freimut including as well other properties that we believe are necessary and useful. Below are described the properties that have not been previously presented and the possible set of values for each one.

Quality of attribute value's description It refers to expressing what level of quality the descriptions of the attributes' values have. For this property there are no possible values defined, therefore the existing quality level for each taxonomy must be expressed.

Quality of attribute value's examples It refers to expressing what level of quality the examples of attributes' values have. The possible values for this property would be "Good", "Bad" or "No examples contained".

Classification It refers to the way of classifying taxonomies according to [8]. The possible values for this property would be "Defect taxonomy", "Root cause analysis" or "Defect classification".

Structure It refers to the structure of the taxonomy. The possible values for this property would be "Hierarchical", "Tree", "Orthogonal" or "Semi-Orthogonal".

Generality's level It refers to the capacity of a taxonomy to be applied in different software projects or processes. The possible values for this property would be "Specific" (can be applied only for particular software), "Intermediate" (can be applied only in a phase of software development) or "Generic" (can be applied in any phase of the development process).

Adaptability It refers to the capacity of expansion and modification of the taxonomy depending on the requirements. The possible values for this property would be "Adaptable", "Adaptable with terms" or "Non adaptable".

Learning time It refers to the time it takes to learn how to use certain taxonomy. The possible values for this property would be "High", "Medium" or "Low".

Classification time It refers to the time it takes to classify a defect in a taxonomy after knowing how to use it. The possible values for this property would be "High", "Medium" or "Low".

The properties proposed in section 2.3 and the possible set of values for each one are listed below.

Mutually exclusive attribute values "Yes" or "No" are the possible values for this property.

Orthogonal attributes The possible values for this property would be "Yes", "No" and "Does not apply". The last value applies only when the taxonomy is composed by one attribute.

Complete attribute values The possible values for this property would be "Yes", "Aparent" and "No".

4 Comparison

The way the attribute view should be used is as follows: for each attribute in the view, look for a corresponding attribute in the taxonomy under evaluation. Table 1 shows the evaluation of the six taxonomies against the framework's attribute view. It is observed that the only attribute included in all taxonomies, is the defect type.

Binders, Kaner, Beizer an HP classify the defect only when it has been detected. ODC and IEEE carry out the classification once the failure occurs and also after the defect is detected. IEEE also carries out the classification once the defect is corrected and when the tracking of the defect is finished.

Table 1. Taxonomies' comparison against the Attribute view

Attribute	**HP**	**Kaner**	**Binder**	**IEEE**	**ODC**	**Beizer**
Suspected location				Suspected Cause		
Real location	Origin			Source, Actual Cause	Target	
Failure's causing phase				Project Phase		
Injection's phase						
Symptom				Symptom, Product Status		
Type	Type	Type	Type	Type	Type	Type
Failure's causing mechanism				Project Activity	Trigger, Activity	
Injection's mechanism	Mode		Type		Qualifier	
Correction's mechanism				Corrective Action, Resolution		
Failure's impact				Customer value, Mission/Safety	Impact	
Defect's impact				Project Cost, Project Quality/Reliability, Project Risk, Societal		
Cause (Error)						
Responsible source					Source, Age	
Re-correction					Age	
Occurrence's Probability				Repeatability		
Severity				Severity		
Priority				Priority		
Detection's cost						
Estimated cost of correction				Project Schedule		
Real cost of correction						

We can observe that there are attributes of the framework that are not contemplated in any of the taxonomies. The following presents each of them and the reasons for the addition in the framework.

Injection phase It is interesting to register the phase in which the defect is injected because that information can be used to prevent the injection from similar defects in future projects.

Cause (Error) It is interesting to register which was the human error that led to the defect. By being aware of the problem, defects in future projects can be avoided. Some results of root cause analysis studies give possible values for this attribute. For example, in [9], a Cause attribute is proposed with the values Education, Communication and Tools. In [10] an attribute that captures different types of causes is proposed: Human causes, Project causes, Revision Causes.

Detection's cost It is interesting because a costs' analysis could be made depending on the different defect types. This allows to estimate changes in the project's schedule.

Real cost of correction Such as the detection's cost, its registration is interesting for a future analysis of costs.

Table 2 presents the comparison of the six taxonomies against the framework's property view.

In HP, Kaner, IEEE and ODC taxonomies the values for an attribute are mutually exclusive. However, in Binder and Beizer's taxonomies, the set of values for an attribute does not present this property, which can generate inconsistent data since a defect could be classified in different ways.

Observing the orthogonality of attributes, this property is not classified by Kaner, Binder or Beizer's taxonomies because these constituted of only one attribute. For IEEE and ODC, this property is fulfilled since the value of each attribute is completely independent of the values of the other attributes. However, for HP this does not apply because the values of the Type attribute depends on the value chosen for the Origin attribute.

The property of having a complete set of attribute values is a difficult one to demonstrate. The Beizer is the only taxonomy that fulfills this property due to a special value that means "others". This value is used when a defect characteristic does not correspond to any of the other values.

Regarding the quality of the descriptions and examples of the values of an attribute, a variation among the different taxonomies is observed. Anyhow, the HP, IEEE and ODC taxonomies can be grouped under a level of good quality since they present complete and clear descriptions, also they include concise examples. On the other hand, Kaner, Binder and Beizer's taxonomies can be grouped under a level of bad quality given their attributes are ambiguous or incomplete and they do not have descriptions or examples.

Regarding the learning time, the taxonomies that are considered easily understandable were grouped with "Low", either because of the clarity and no ambiguity of its presentation, and/or because the taxonomies have a small amount of attributes. The taxonomies presented by Binder and Beizer were tagged with a "High" value. Although they are composed by only one attribute, many possible values exist for it. This, added to vague descriptions and examples of poor quality, result in a longer learning time than the other group. The taxonomy presented by IEEE is also considered to have "High" learning time because it contains such a large amount of attributes that they have to be studied and understood.

Table 2. Taxonomies' comparisons against the Property view

Property	**HP**	**Kaner**	**Binder**	**IEEE**	**ODC**	**Beizer**
Mutually exclusive attribute values	Yes	Yes	No	Yes	Yes	No
Orthogonal attributes	No	Does not apply	Does not apply	Yes	Yes	Does not apply
Complete attribute values	Aparent	Aparent	Aparent	Aparent	Aparent	Yes
Quality of attribute value's description	Good: clear descriptions, no ambiguities	Bad: not all values are explained	Values are not described	Good: clear descriptions, no ambiguities	Good: clear descriptions, no ambiguities	Bad: superficial description, ambiguous in some cases
Quality of attribute value's examples	Good	No examples contained	No examples contained	Good	Good	Bad
Classification	Defect classification	Defect Taxonomy	Defect Taxonomy	Defect classification	Defect classification	Defect Taxonomy
Structure	Semi-Orthogonal	Does not apply	Does not apply	Orthogonal and Hierarchical	Orthogonal	Hierarchical
Generality's level	Generic	Generic	Specific	Generic	Generic	Generic
Adaptability	Adaptable	Adaptable	Adaptable	Adaptable	Adaptable	Adaptable
Learning time	Low	Low	High	High	Low	High
Classification time	Low	Low	High	High	Low	High

The proposed reasoning for the time of classification is analogue. Those taxonomies, with no ambiguous descriptions, good examples and/or few attributes to classify, are grouped under the value "Low". The taxonomies with higher amount of attributes as well as those that are not clearly defined or exemplified, take a longer time for the user to classify because of a lack of certainty among two or more values, are considered to have a "High" time of classification.

It is clear that classifying time as "High", "Medium" or "Low" is subjective, with the connotations that it implies. In the future, an average of the times of learning and classification could be calculated for each taxonomy, and then be able to formulate a better comparison, more representative of the reality.

It is observed that the amount of attributes, the quality of the descriptions' values of the attributes and the quality of the examples presented in each taxonomy; significantly influence both the learning and the classification times of the taxonomy.

5 Conclusions

We developed and presented an original comparison framework for defect taxonomies. This framework extends and improves Freimut ideas. It contains two relevant views: attribute and property. The first view is useful to evaluate those defects' characteristics that are considered for a given taxonomy. The second view is useful to evaluate desired properties each taxonomy should have.

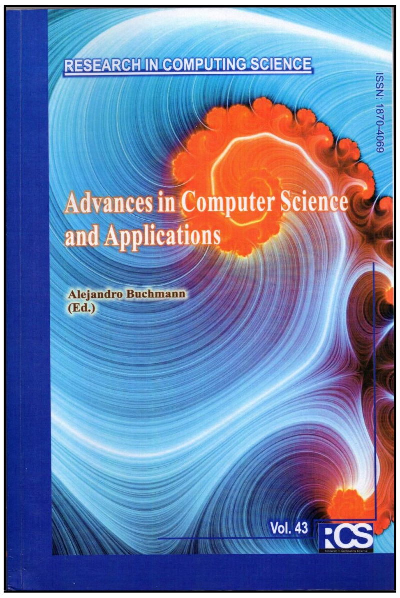
Using the framework we evaluate and compare six well-known taxonomies. The results show that each taxonomy considers different characteristics of a defect. Some of them are more complete, from an attribute point of view, than others. However, those less complete have a bigger set of values for their attributes, for example, Beizer and Binder. The analysis also shows that there are differences in the properties among the taxonomies.

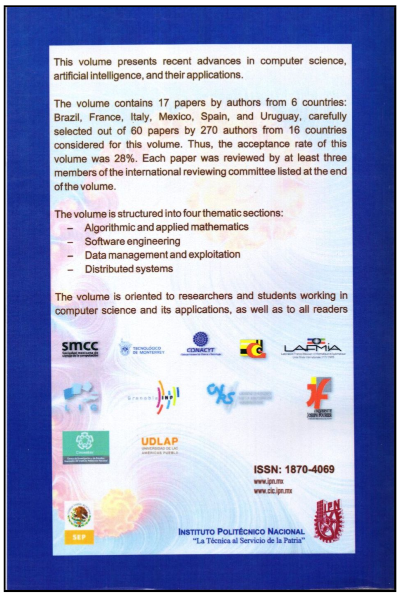
As a future work it is important to: analyze the impact of whether having or not a specific framework's attribute in a taxonomy, analyze the values proposed in each taxonomy for each attribute, analyze if different test levels (unit, integration, system) require different taxonomies and finally analyze the way taxonomies have been used in the industry. We are currently working on these last two points.

References

1. Grady, R.B.: Practical Software Metrics For Project Management and Process Improvement. Hewlett-Packard (1992)
2. Kaner, C., Falk, J., Nguyen, H.Q.: Testing Computer Software (2nd. Edition). International Thomson Computer Press (1999)
3. Binder, R.V.: Testing Object-oriented Systems Models, Patterns, and Tools. Addison-Wesley (1999)
4. IEEE: IEEE 1044-1993 Standard Classification for Software Anomalies. Institute of Electrical and Electronics Engineers (1993)
5. Chillarege, R.: Handbook of Software Reliability Engineering. IEEE Computer Society Press, McGraw-Hill Book Company (1996)
6. Beizer, B.: Software Testing Techniques, Second Edition. Van Nostrand Reinhold Co. (1990)
7. Freimut, B.: Developing and using defect classification schemes. Technical report, Fraunhofer IESE (2001)
8. Wagner, S.: Defect Classifications and Defect Types Revisited. Technische Universitt Mnchen (2008)
9. Mays, R.G., Jones, C.L., Holloway, G.J., Studinski, D.: Experiences with defect prevention. IBM SYSTEMS JOURNAL (1990)
10. Leszak, M., Perry, D.E., Stoll, D.: A case study in root cause defect analysis. Proceedings of the 22nd international conference on Software engineering (2000)

ARTICLE


RESEARCH IN COMPUTING SCIENCE
ISSN: 1870-4069
Advances in Computer Science and Applications
Alejandro Buchmann (Ed.)
Vol. 43
RCS


This volume presents recent advances in computer science, artificial intelligence, and their applications.

The volume contains 17 papers by authors from 6 countries: Brazil, France, Italy, Mexico, Spain, and Uruguay, carefully selected out of 60 papers by 270 authors from 16 countries considered for this volume. Thus, the acceptance rate of this volume was 28%. Each paper was reviewed by at least three members of the international reviewing committee listed at the end of the volume.

The volume is structured into four thematic sections:
- Algorithmic and applied mathematics
- Software engineering
- Data management and exploitation
- Distributed systems

The volume is oriented to researchers and students working in computer science and its applications, as well as to all readers

ISSN: 1870-4069
www.ipn.mx
www.cic.ipn.mx

INSTITUTO POLITÉCNICO NACIONAL
"La Técnica al Servicio de la Patria"

Towards a Framework to Compare Formal Experiments that Evaluate Testing Techniques

Diego Vallespir, Silvana Moreno, Carmen Bogado and Juliana Herbert

Research in Computing Science, pp. 69-80, ISSN 1870-4069, 2009.

Towards a Framework to Compare Formal Experiments that Evaluate Testing Techniques

Diego Vallespir[1], Silvana Moreno[1], Carmen Bogado[1], and Juliana Herbert[2]

[1] Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Montevideo, Uruguay.
[2] Herbert Consulting
Porto Alegre, RS, Brazil.
dvallesp@fing.edu.uy , {silvanamoren, cmbogado}@gmail.com,
juliana@herbertconsulting.com

Abstract. There are many formal experiments to evaluate the performance of different software testing techniques. The first we know about is from 1978 [1].The most recent one is currently under execution and some initial results have already been obtained [2]. Having a comparison framework of experiments is necessary in order to be able to formally compare them and make progress on the construction of new experiments based on previous ones. This paper presents a comparison framework and four known formal experiments are compared.

1 Introduction

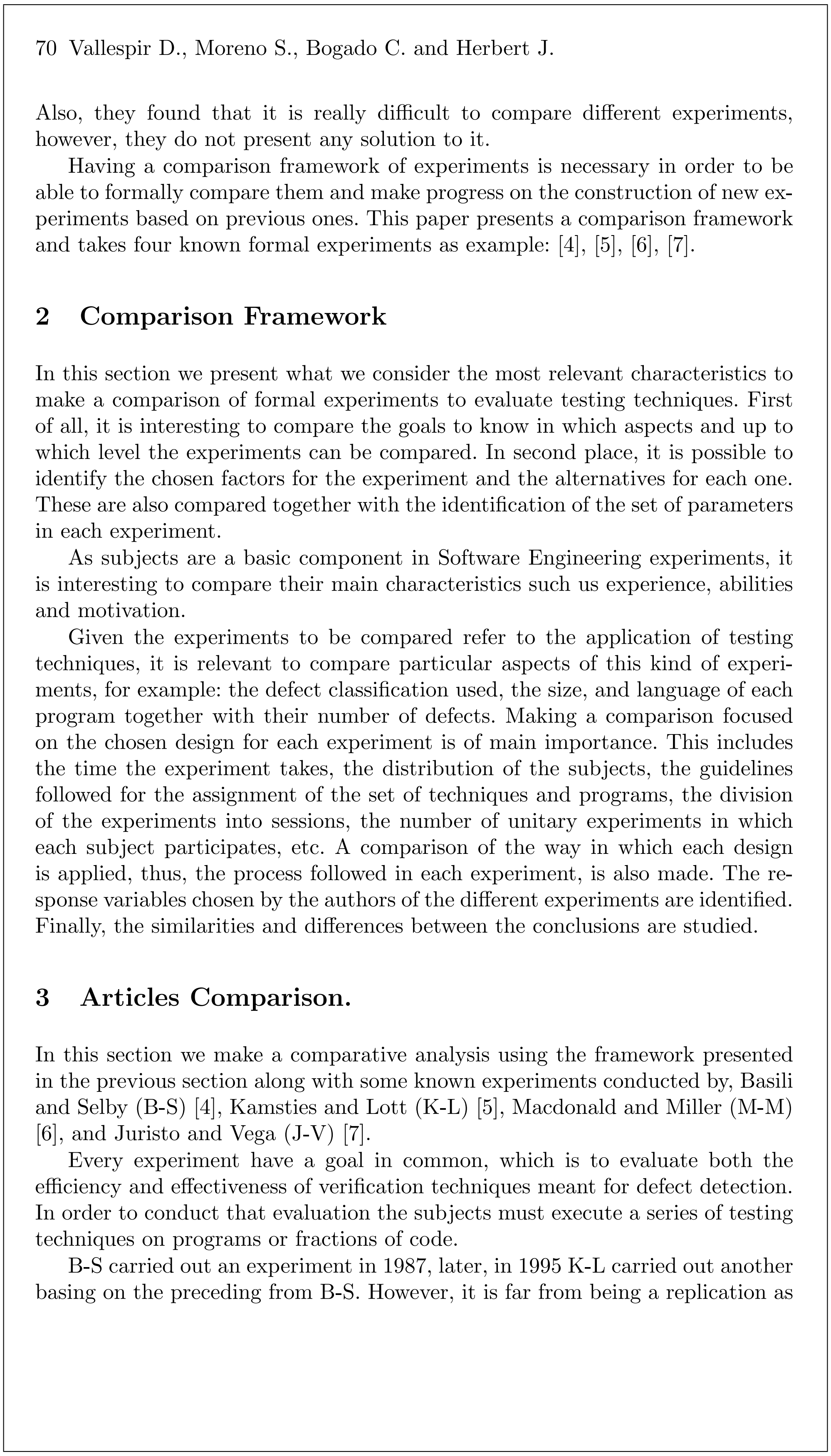
It is normal to use a hammer to hammer a nail into a wall. There are many types of hammers but it is easy to choose one and even more, a lot of hammers do the same job. It is normal to use a software testing technique to verify a software unit. Unfortunately, it is not known which one to choose nor if different techniques perform the same way for the same task.

The performance of each technique (cost, effectiveness and efficiency) has to be known at the time of choosing a testing technique. But to obtain this kind of knowledge is not easy given the variability of their performance, which depends on the subject that applies it, the programming language and the application type that is being tested (information system, robotics, etc). Some advances have been made but there is still a long way to go.

There are many formal experiments to study the performance of different software testing techniques. The first we know about is from 1978 [1]. The most recent one is currently under execution and some initial results have already been obtained [2].

Many years of empiric investigation in the subject have gone by though there are not definite results yet. In *A look at 25 years of data*, the authors have reached the same conclusion after studying various experiments on software testing [3].

©A. Buchmann (Ed.)
Advances in Computer Science and Applications.
Research in Computer Science 43, 2009, pp. 69-80



Also, they found that it is really difficult to compare different experiments, however, they do not present any solution to it.

Having a comparison framework of experiments is necessary in order to be able to formally compare them and make progress on the construction of new experiments based on previous ones. This paper presents a comparison framework and takes four known formal experiments as example: [4], [5], [6], [7].

2 Comparison Framework

In this section we present what we consider the most relevant characteristics to make a comparison of formal experiments to evaluate testing techniques. First of all, it is interesting to compare the goals to know in which aspects and up to which level the experiments can be compared. In second place, it is possible to identify the chosen factors for the experiment and the alternatives for each one. These are also compared together with the identification of the set of parameters in each experiment.

As subjects are a basic component in Software Engineering experiments, it is interesting to compare their main characteristics such us experience, abilities and motivation.

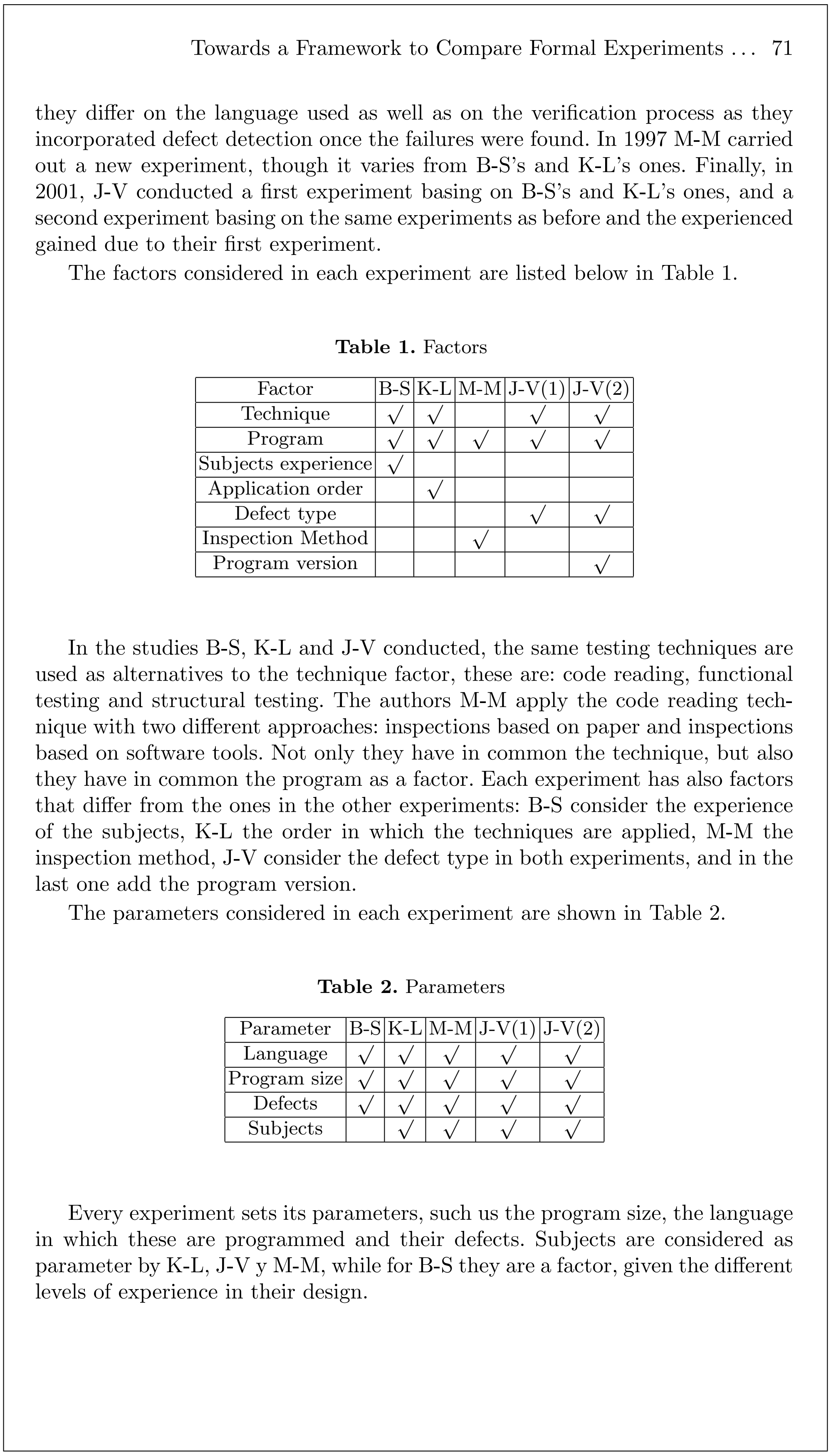
Given the experiments to be compared refer to the application of testing techniques, it is relevant to compare particular aspects of this kind of experiments, for example: the defect classification used, the size, and language of each program together with their number of defects. Making a comparison focused on the chosen design for each experiment is of main importance. This includes the time the experiment takes, the distribution of the subjects, the guidelines followed for the assignment of the set of techniques and programs, the division of the experiments into sessions, the number of unitary experiments in which each subject participates, etc. A comparison of the way in which each design is applied, thus, the process followed in each experiment, is also made. The response variables chosen by the authors of the different experiments are identified. Finally, the similarities and differences between the conclusions are studied.

3 Articles Comparison.

In this section we make a comparative analysis using the framework presented in the previous section along with some known experiments conducted by, Basili and Selby (B-S) [4], Kamsties and Lott (K-L) [5], Macdonald and Miller (M-M) [6], and Juristo and Vega (J-V) [7].

Every experiment have a goal in common, which is to evaluate both the efficiency and effectiveness of verification techniques meant for defect detection. In order to conduct that evaluation the subjects must execute a series of testing techniques on programs or fractions of code.

B-S carried out an experiment in 1987, later, in 1995 K-L carried out another basing on the preceding from B-S. However, it is far from being a replication as



they differ on the language used as well as on the verification process as they incorporated defect detection once the failures were found. In 1997 M-M carried out a new experiment, though it varies from B-S's and K-L's ones. Finally, in 2001, J-V conducted a first experiment basing on B-S's and K-L's ones, and a second experiment basing on the same experiments as before and the experienced gained due to their first experiment.

The factors considered in each experiment are listed below in Table 1.

Table 1. Factors

Factor	B-S	K-L	M-M	J-V(1)	J-V(2)
Technique	√	√		√	√
Program	√	√	√	√	√
Subjects experience	√				
Application order		√			
Defect type				√	√
Inspection Method			√		
Program version					√

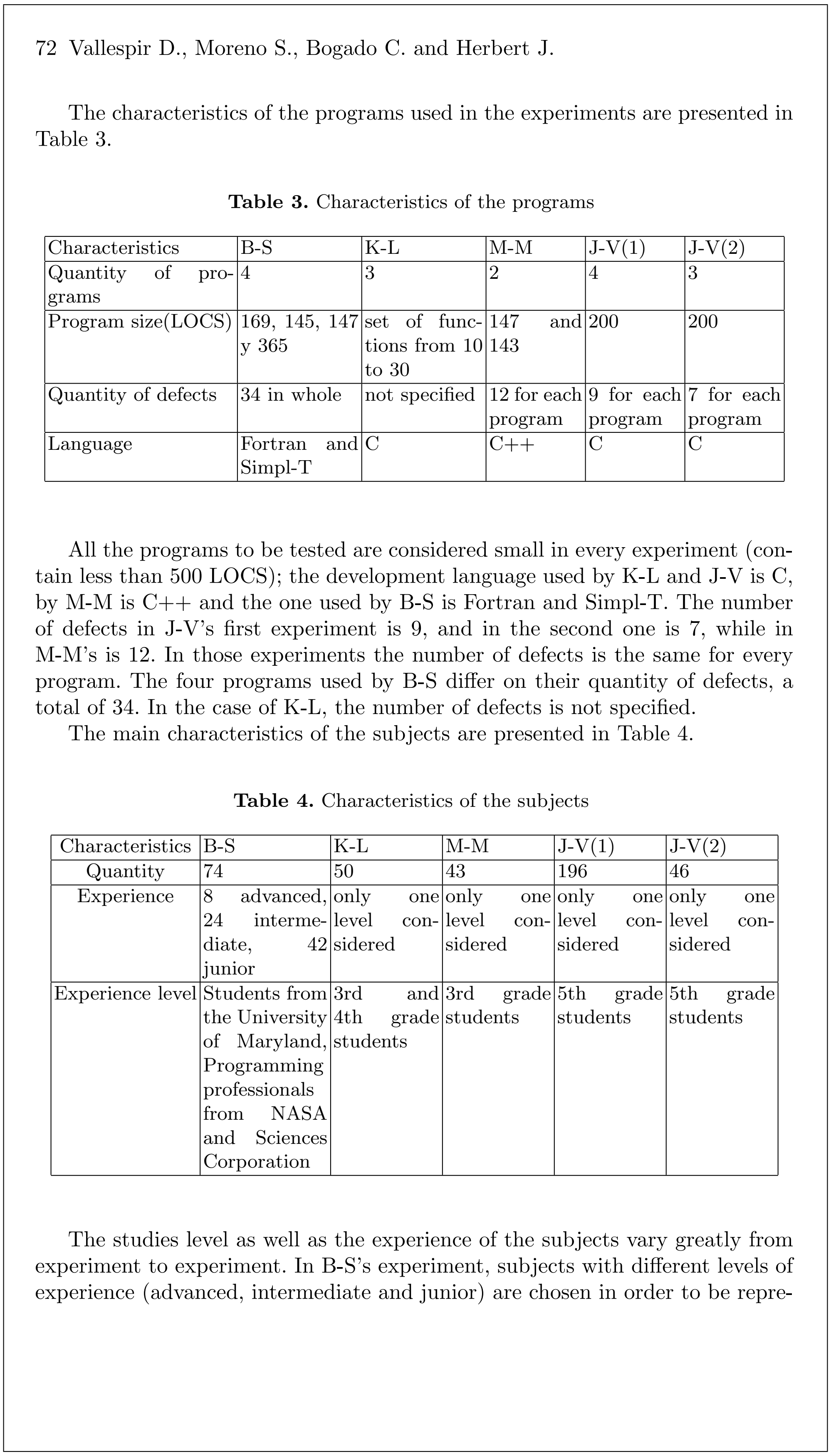
In the studies B-S, K-L and J-V conducted, the same testing techniques are used as alternatives to the technique factor, these are: code reading, functional testing and structural testing. The authors M-M apply the code reading technique with two different approaches: inspections based on paper and inspections based on software tools. Not only they have in common the technique, but also they have in common the program as a factor. Each experiment has also factors that differ from the ones in the other experiments: B-S consider the experience of the subjects, K-L the order in which the techniques are applied, M-M the inspection method, J-V consider the defect type in both experiments, and in the last one add the program version.

The parameters considered in each experiment are shown in Table 2.

Table 2. Parameters

Parameter	B-S	K-L	M-M	J-V(1)	J-V(2)
Language	√	√	√	√	√
Program size	√	√	√	√	√
Defects	√	√	√	√	√
Subjects		√	√	√	√

Every experiment sets its parameters, such us the program size, the language in which these are programmed and their defects. Subjects are considered as parameter by K-L, J-V y M-M, while for B-S they are a factor, given the different levels of experience in their design.



The characteristics of the programs used in the experiments are presented in Table 3.

Table 3. Characteristics of the programs

Characteristics	B-S	K-L	M-M	J-V(1)	J-V(2)
Quantity of programs	4	3	2	4	3
Program size(LOCS)	169, 145, 147 y 365	set of functions from 10 to 30	147 and 143	200	200
Quantity of defects	34 in whole	not specified	12 for each program	9 for each program	7 for each program
Language	Fortran and Simpl-T	C	C++	C	C

All the programs to be tested are considered small in every experiment (contain less than 500 LOCS); the development language used by K-L and J-V is C, by M-M is C++ and the one used by B-S is Fortran and Simpl-T. The number of defects in J-V's first experiment is 9, and in the second one is 7, while in M-M's is 12. In those experiments the number of defects is the same for every program. The four programs used by B-S differ on their quantity of defects, a total of 34. In the case of K-L, the number of defects is not specified.

The main characteristics of the subjects are presented in Table 4.

Table 4. Characteristics of the subjects

Characteristics	B-S	K-L	M-M	J-V(1)	J-V(2)
Quantity	74	50	43	196	46
Experience	8 advanced, 24 intermediate, 42 junior	only one level considered	only one level considered	only one level considered	only one level considered
Experience level	Students from the University of Maryland, Programming professionals from NASA and Sciences Corporation	3rd and 4th grade students	3rd grade students	5th grade students	5th grade students

The studies level as well as the experience of the subjects vary greatly from experiment to experiment. In B-S's experiment, subjects with different levels of experience (advanced, intermediate and junior) are chosen in order to be representative of different levels of knowledge of reality. While in K-L's the subjects are students in third and fourth year from the University of Kaiserslautern. They have some experience in C, but even so, they have a previous training in the usage of techniques and language. This previous training is also put into practice in both J-V's experiments, who choose inexperienced students in the fifth year from the Computer Sciences School, Polytechnic University of Madrid. The subjects involved in M-M's experiment are in the third year of the career, have a solid basis on Scheme, C++ and Eiffel programming, are highly motivated by the experience given it is associated with a major class project and therefore they are graded on their work.

Some design decisions related to each experiment are shown in Table 5.

Both B-S and K-L used the same classification scheme for defects, J-V a subclassification from that scheme while M-M did not classified any defect found. The scheme used is that proposed in Basili's article, in which defects are classified as omission and commission at a first level. The commission defects are those which appear as a result of an incorrect segment of existing code. Omission defects are those which result when the programmer omits including an entity. Besides, the classification system is divided to distinguish the defect types: initialization, calculation, control, interface, data and cosmetics. According to the designs, all the authors decide to make a training prior to the experiment in order to present the subjects with the techniques to be used. The B-S's experiment is divided in 5 sessions. The first one is training, the following three consist of the experiment itself and a follow-up session. K-L's and M-M's experiments consist of two experiment sessions after a training phase. J-V organize the experiment in five sessions: the training session and four execution ones. All through B-S's experiment four programs are used, but in each session three out of the four are tested, thus there is a combination of programs never tested. In the case of K-L's, M-M's and J-V's experiments all the programs are tested in every session. In K-L's are three programs, in M-M's two, and four and three in J-V's experiments, respectively.

Table 6 presents the characteristics of the process followed in each experiment.

The total number of subjects in B-S's experiment is 72, divided in: 8 advanced, 24 intermediate and 42 junior. At the same time they are organized as follows: 29 in the first session, 13 in the second and 32 in the third one. In the first two sessions only subjects with intermediate and junior levels participate, while in the third session advanced subjects are also involved. In K-L's fewer subjects are involved: 27 in the first session and 23 different subjects in the second one. The number of subjects in M-M's experiment is close to K-L's: 43. In this case they are organized in two sessions: 22 and 21 subjects respectively. At the same time they are divided in working groups: six groups of three subjects and one of four in the first session, and seven groups of three subjects in the second one. The total number of subjects in J-V's first experiment rises to 196, organized in 8 groups of 12 subjects (four groups apply structural techniques and the other four apply functional techniques), and four groups of 25 subjects

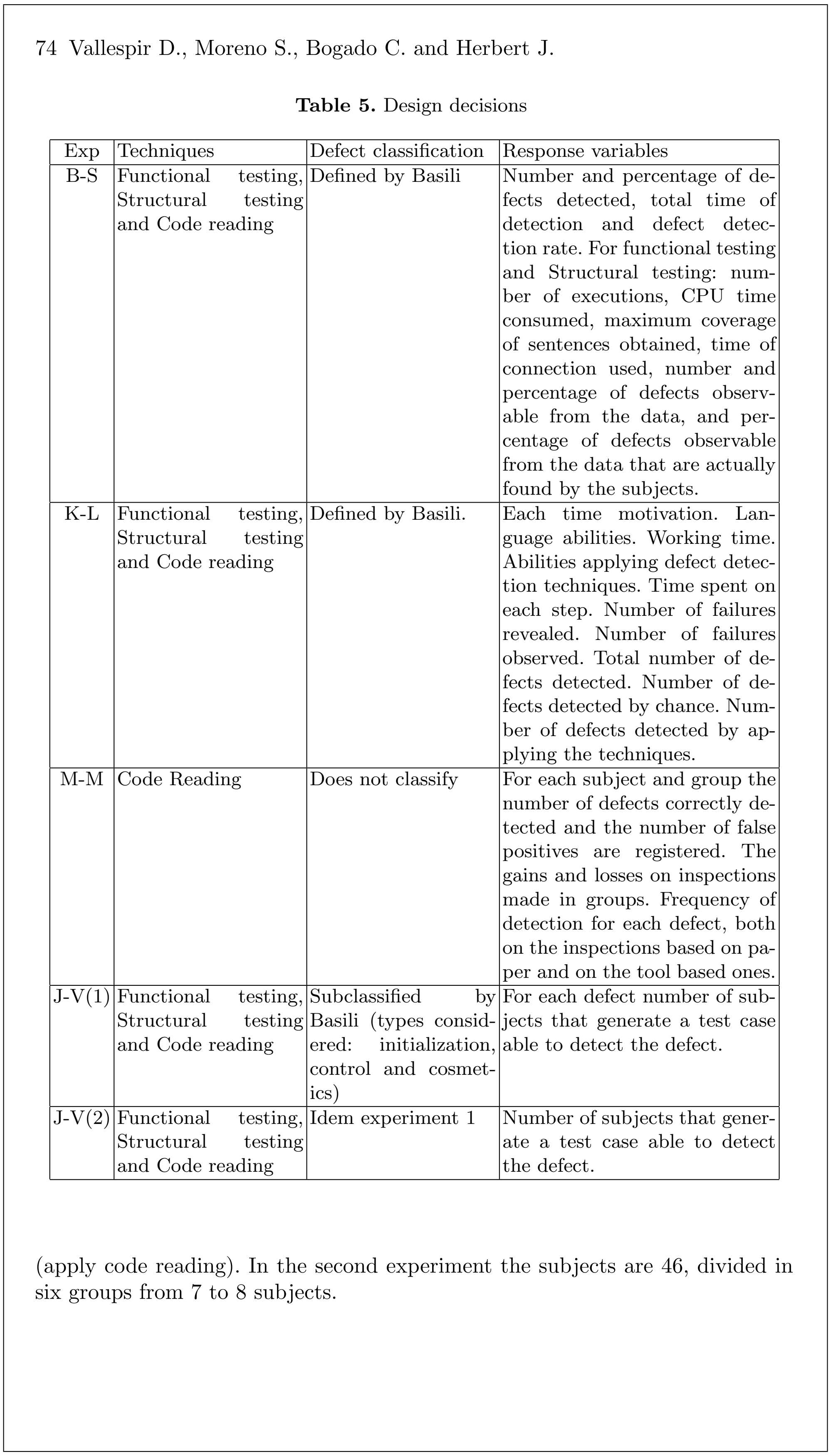


Table 5. Design decisions

Exp	Techniques	Defect classification	Response variables
B-S	Functional testing, Structural testing and Code reading	Defined by Basili	Number and percentage of defects detected, total time of detection and defect detection rate. For functional testing and Structural testing: number of executions, CPU time consumed, maximum coverage of sentences obtained, time of connection used, number and percentage of defects observable from the data, and percentage of defects observable from the data that are actually found by the subjects.
K-L	Functional testing, Structural testing and Code reading	Defined by Basili.	Each time motivation. Language abilities. Working time. Abilities applying defect detection techniques. Time spent on each step. Number of failures revealed. Number of failures observed. Total number of defects detected. Number of defects detected by chance. Number of defects detected by applying the techniques.
M-M	Code Reading	Does not classify	For each subject and group the number of defects correctly detected and the number of false positives are registered. The gains and losses on inspections made in groups. Frequency of detection for each defect, both on the inspections based on paper and on the tool based ones.
J-V(1)	Functional testing, Structural testing and Code reading	Subclassified by Basili (types considered: initialization, control and cosmetics)	For each defect number of subjects that generate a test case able to detect the defect.
J-V(2)	Functional testing, Structural testing and Code reading	Idem experiment 1	Number of subjects that generate a test case able to detect the defect.

(apply code reading). In the second experiment the subjects are 46, divided in six groups from 7 to 8 subjects.

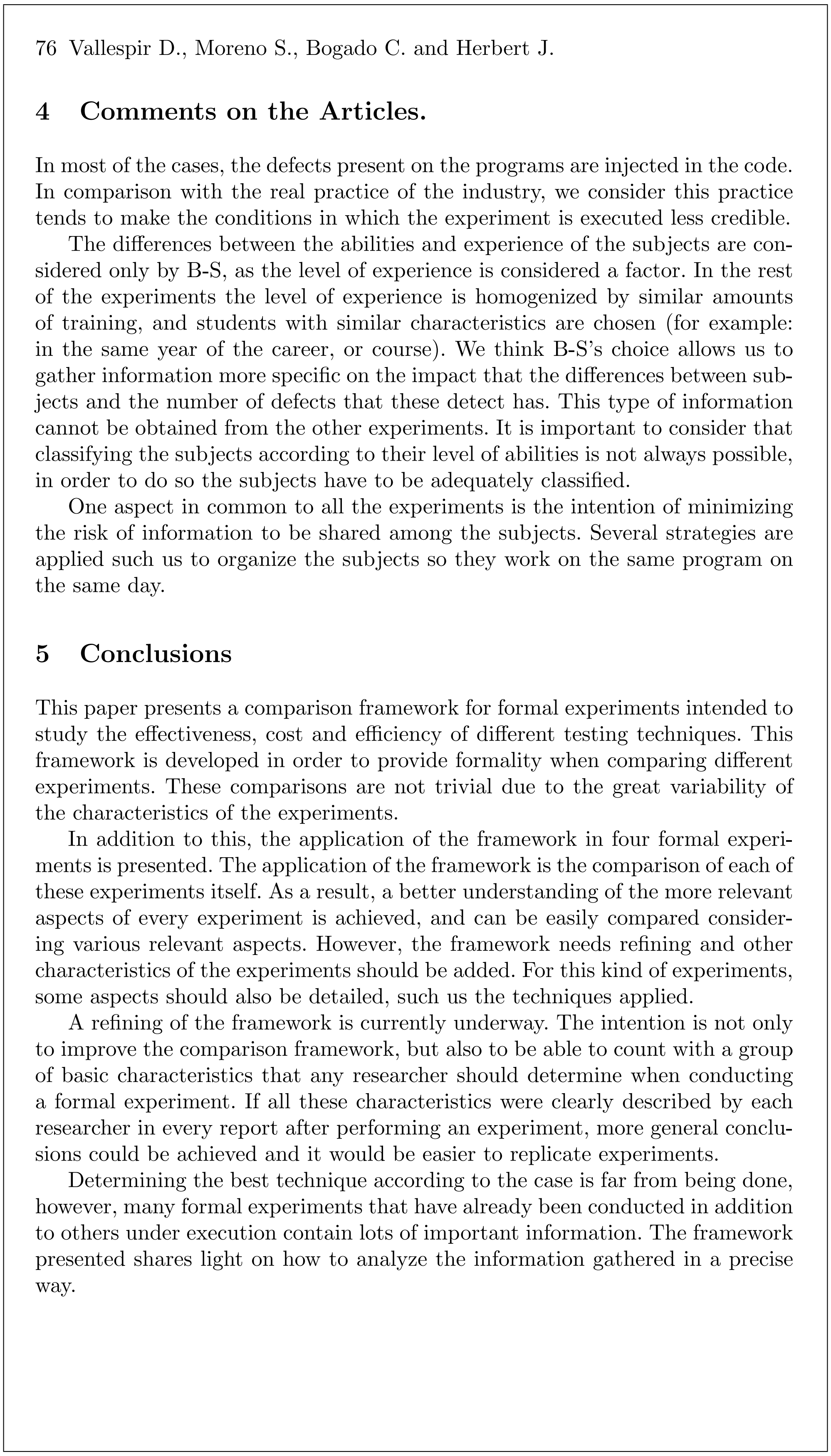
In the process followed by Basili, three out of the four programs are used in each phase, and every subject uses the three techniques and tests the three programs. In each phase every subject tests the same program on the same day. The K-L's experiment consists of two internal replications, for which three days of a week are settled. Each day a different program is tested by applying the three techniques. The first day participate 23 students, the second 19 and the third day 15. The inspection process of M-M's experiment takes two sessions. The first session is of individual detection and the second is when the consolidation is achieved after working in groups. The experiment is conducted for a period of 10 weeks. During the first six weeks the students are trained. During the remaining four the experiment is performed. The inspection consists of inspecting the source code using a check list, considering the specification of the program.

In M-M's every subject applies both inspections and works on the two programs. In J-V's first experiment a session is carried out each day using one program, and each of the three groups execute a different technique from the three possible ones. The intention of J-V's design is to eliminate the validity threat of the learning, while for K-L the intention is to minimize this threat by assigning the subjects each technique and program only once. In the second experiment of J-V the design is adapted so all the subjects apply every technique. It is organized in three days, and each day only one program and all the techniques are executed. Each of these is executed by two different groups.

The general conclusions reached in each experiment are presented in Table 7.

The complementary conclusions reached in each experiment are presented in Table 8.

Both B-S and K-L reached the conclusion that code reading technique is as effective as structural and functional testings when it comes to the number of defects detected. They also concluded that subjects were more efficient when applying functional testing. M-M concluded that there is no big difference between inspections based on paper and those based on the tool, either it is individual or working in group. With regard to J-V's first experiment, they concluded that the number of subjects that detect a defect depends not only on the program used but also on the technique applied and the defect itself. In addition to this, code reading technique is less sensitive to defect hiding than the other techniques. Functional testing behaves better than the structural testing technique in most cases, but the cases in which the two techniques behave identically, or one better than the other, occur indistinctly for each defect type. According to their second experiment, they concluded that regardless of the defect type, code reading technique is not as effective as functional and structural techniques. Another conclusion is that functional and structural techniques behave identically. The addition of the program version in this last experiment has an impact on the number of subjects that finds a defect. More subjects develop test cases that detect the defects in a version while fewer do it in the other, regardless of the program, technique and defect.



4 Comments on the Articles.

In most of the cases, the defects present on the programs are injected in the code. In comparison with the real practice of the industry, we consider this practice tends to make the conditions in which the experiment is executed less credible.

The differences between the abilities and experience of the subjects are considered only by B-S, as the level of experience is considered a factor. In the rest of the experiments the level of experience is homogenized by similar amounts of training, and students with similar characteristics are chosen (for example: in the same year of the career, or course). We think B-S's choice allows us to gather information more specific on the impact that the differences between subjects and the number of defects that these detect has. This type of information cannot be obtained from the other experiments. It is important to consider that classifying the subjects according to their level of abilities is not always possible, in order to do so the subjects have to be adequately classified.

One aspect in common to all the experiments is the intention of minimizing the risk of information to be shared among the subjects. Several strategies are applied such us to organize the subjects so they work on the same program on the same day.

5 Conclusions

This paper presents a comparison framework for formal experiments intended to study the effectiveness, cost and efficiency of different testing techniques. This framework is developed in order to provide formality when comparing different experiments. These comparisons are not trivial due to the great variability of the characteristics of the experiments.

In addition to this, the application of the framework in four formal experiments is presented. The application of the framework is the comparison of each of these experiments itself. As a result, a better understanding of the more relevant aspects of every experiment is achieved, and can be easily compared considering various relevant aspects. However, the framework needs refining and other characteristics of the experiments should be added. For this kind of experiments, some aspects should also be detailed, such us the techniques applied.

A refining of the framework is currently underway. The intention is not only to improve the comparison framework, but also to be able to count with a group of basic characteristics that any researcher should determine when conducting a formal experiment. If all these characteristics were clearly described by each researcher in every report after performing an experiment, more general conclusions could be achieved and it would be easier to replicate experiments.

Determining the best technique according to the case is far from being done, however, many formal experiments that have already been conducted in addition to others under execution contain lots of important information. The framework presented shares light on how to analyze the information gathered in a precise way.

Table 6. Characteristics of the process

Exp	Organization of the subjects	Training	Process
B-S	8 advanced, 24 intermediate and 42 junior. First session 29, Second 13 and third 32.	The first session consists of the initial training. Subjects are presented with similar types of trainings.	In each phase, three out of the four programs are used, and every subject applies the three techniques and tests the three programs. Each phase consists of 5 sessions: an initial training, three testing sessions and a follow-up session. In each phase every subject tests the same program on the same day.
K-L	First replication: 27 subjects. Second replication: 23 students the first day, 19 the second and 15 the third day	The subjects are presented with the testing techniques and trained prior to the execution	The experiment consists of two internal replications, which are conducted in three settled days out of a week. Each day a different program is tested with the three techniques. The same guidelines as in the training were used.
M-M	Session 1 consists of six groups of three subjects and one of four, while session 2 consists of seven groups of three subjects.	During the first six weeks of the experiment the subjects are trained	The process of inspection takes two sessions. The first one is of individual detection and the second consists of working in groups to reach consolidation. The experiment is conducted for 10 weeks, during the last four the inspections are executed. These consist of inspecting the source code using a check list, considering the specification of the program.
J-V(1)	Consists of 8 groups of 12 subjects (4 test with structural techniques and the other four with functional techniques), and 4 groups of 25 subjects (testing with code reading)	During the initial session the training is conducted	The experiment is organized in 5 sessions, during the last four the testing of the programs is performed. Each day a session is executed, each of the three groups execute a program with a different technique from the three available ones.
J-V(2)	Consists of six groups from seven to eight subjects		The design is changed so every group execute all the techniques. It is organized in three days, and each day only one program and all the techniques are executed. Each of these is executed by two different groups.

Table 7. General Conclusions of the authors

Exp	General conclusion
B-S	According to the number of defects detected and the associated cost, code reading technique is as effective as functional and structural testings. The efficiency, effectiveness and cost depend on the type of software under test.
K-L	When detecting defects, any technique can be as effective as the others, if time is not considered an important aspect and every subject lacks of experience in the language as well as in the three techniques under study.
M-M	There is no big difference between inspections based on paper and those based on the tool, either it is individual or working in group.
J-V(1)	The number of subjects that detect a defect depends not only on the program used but also on the technique applied and the defect itself. Some defects behave better when certain programs are used as well as defects that do so when certain techniques are applied.
J-V(2)	Code reading always behaves worse than the functional and structural techniques, indistinctly for the defect type. With regard to functional and structural techniques, they both behave identically. The program version influences on the number of subjects that detect a defect.

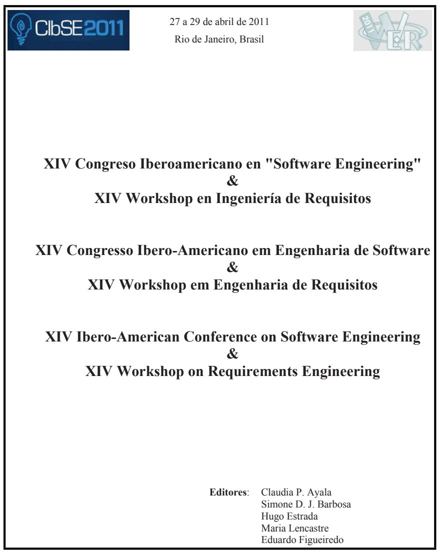
Table 8. Complementary Conclusions of the authors

Exp	Complementary conclusions
B-S	The advanced subjects detected more defects and were more efficient when applying code reading than functional and structural testing. Besides, the number of defects found with functional testing was larger than with structural testing. Intermediate and junior subjects were almost as efficient and effective when applying the three techniques. Code reading technique detected more interface defects than did either of the other techniques, while functional testing did so with control defects. When applying code reading, the subjects gave the most accurate estimates, while functional testers gave the least accurate estimates.
K-L	Subjects were more efficient when applying functional testing.
M-M	Significant differences are not found between both methods as to the number of false positives found, nor in the gained or lost cost due to the meetings.
J-V(1)	Code reading technique is less sensitive to defect hiding than the other techniques. Functional testing behaves better than the structural testing technique in most cases, but the cases in which the two techniques behave identically, or one better than the other, occur indistinctly for each defect type.
J-V(2)	The number of subjects that detect a defect by applying the reading technique does not depend on the observability of the defect. More subjects develop test cases which detect more defects with one version than with the other, regardless of the program, the technique and the defect.

References

1. Myers, G.J.: A controlled experiment in program testing and code walkthroughs/inspections. Communications of the ACM **21**(9) (September 1978) 760–768
2. Vallespir, D., Herbert, J.: Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. In: Proceedings of the Mexican International Conference in Computer Science. (2009)
3. Moreno, A., Shull, F., Juristo, N., Vegas, S.: A look at 25 years of data. IEEE Software **26**(1) (Jan.–Feb. 2009) 15–17
4. Basili, V.R., Selby, R.W.: Comparing the effectiveness of software testing strategies. IEEE Transactions on software engineering **13**(12) (1987) 1278–1296
5. Kamsties, E., Lott, C.M.: An empirical evaluation of three defect-detection techniques. In: Proceedings of the Fifth European Software Engineering Conference. (1995) 362–383
6. Macdonald, F., Miller, J.: A comparison of tool-based and paper-based software inspection. Empirical Software Engineering **3**(3) (1998) 233–253
7. Juristo, N., Vegas, S.: Functional testing, structural testing, and code reading: What fault type do they each detect? Empirical Methods and Studies in Software Engineering **2765/2003** (2003) 208–232

ARTICLE


27 a 29 de abril de 2011
Rio de Janeiro, Brasil

XIV Congreso Iberoamericano en "Software Engineering"
&
XIV Workshop en Ingeniería de Requisitos

XIV Congresso Ibero-Americano em Engenharia de Software
&
XIV Workshop em Engenharia de Requisitos

XIV Ibero-American Conference on Software Engineering
&
XIV Workshop on Requirements Engineering

Editores: Claudia P. Ayala
Simone D. J. Barbosa
Hugo Estrada
Maria Lencastre
Eduardo Figueiredo

CIbSE papers
Congresso Iberoamericano en "Software Engineering"
Congresso Ibero-Americano en Engenharia de Software
Ibero-American Conference on Software Engineering

Construction of a Laboratory Package for a Software Engineering Experiment

Cecilia Apa, Martín Solari, Diego Vallespir and Sira Vegas

Proceedings of the Ibero-American Conference on Software Engineering, pp. 101-114, Rio de Janeiro, Brazil, April 2011.

Construction of a Laboratory Package for a Software Engineering Experiment

Cecilia Apa[1], Martín Solari[2], Diego Vallespir[1], and Sira Vegas[3]

[1] Universidad de la República, Uruguay
[2] Universidad ORT, Uruguay
[3] Universidad Politécnica de Madrid, España
ceapa@fing.edu.uy,martin.solari@ort.edu.uy,
dvallesp@fing.edu.uy,svegas@fi.upm.es

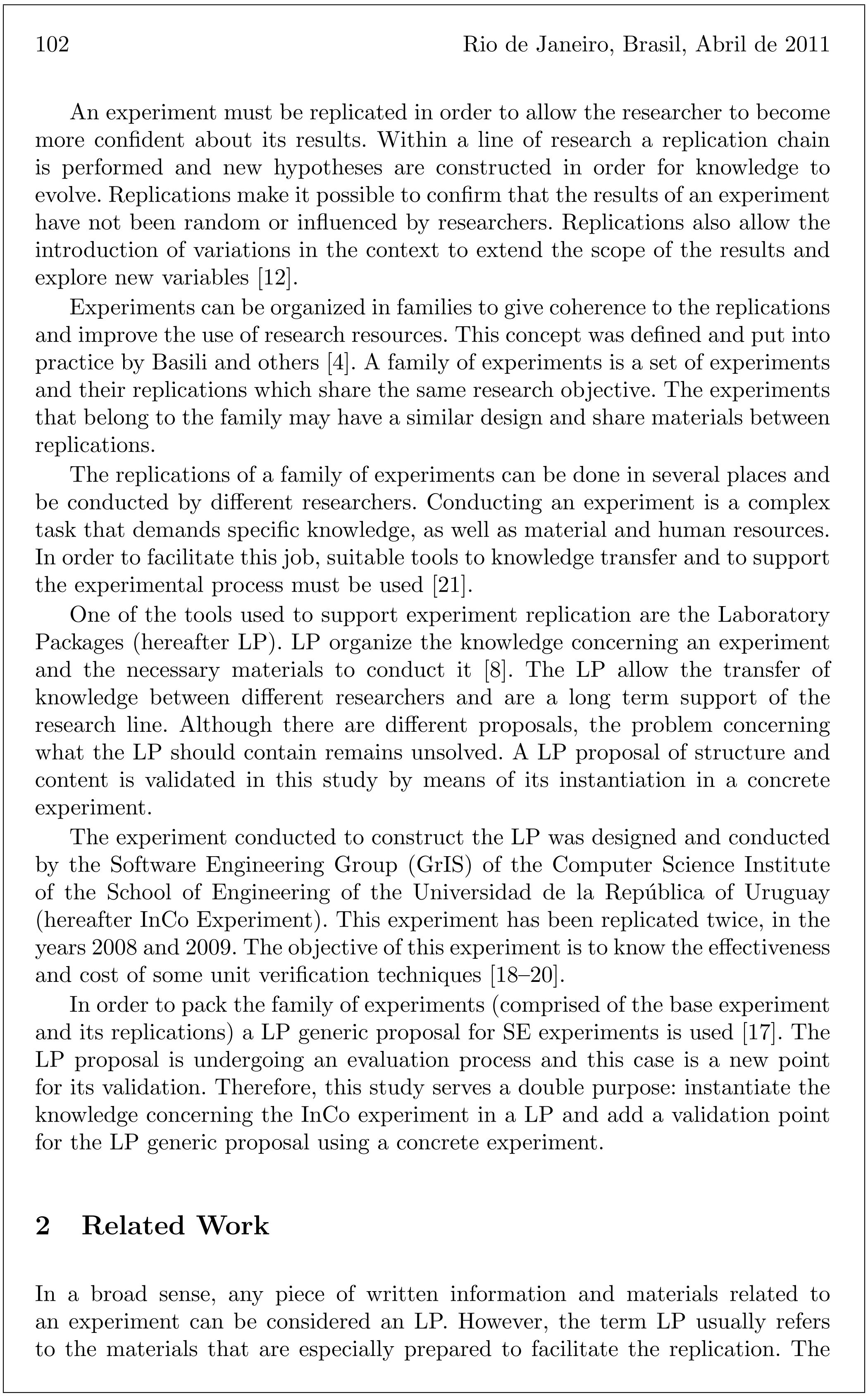
Abstract. This work presents the construction of a laboratory package for a controlled software engineering experiment. A laboratory package is the container of the knowledge and materials necessary to replicate an experiment. Its construction was based on the use of a generic proposal of laboratory packages. The study has two objectives: to obtain a package instance for a concrete experiment and to validate the generic proposal. The defined instantiation process has made it possible to gather the incidents and generate lessons learned. This case study is added as a validation point of the proposal. It is concluded that the proposal is feasible to instantiate a software engineering experiment, and complete to cover the activities of the experimental process. The reviews carried out made it possible to obtain a more adequate package for the experiment and to improve the generic proposal for future instantiations.

Keywords: Empirical Software Engineering, Case Study, Experimentation, Laboratory Package, Controlled Experiment.

1 Introducción

Experimentation is one of the means used in the frame of scientific method to obtain evidence regarding theoretical conjectures. An experiment involves the creation of a controlled environment where a phenomenon can be observed repeatedly [6]. Thus it is possible to isolate the essential mechanisms that have influence on it to identify cause-effect relationships.

In Software Engineering (hereafter SE) experimentation is a research method used to contrast in real situations the theoretical affirmations about techniques and tools [11]. Although empirical methods are used by the SE scientific community, there is still a great number of affirmations that are not supported by evidence. For example, it is common to read affirmations that indicate that a certain technique is better than another despite the fact that it has not been proved except for a particular case. This situation requires that the use of experiments in SE be increased and deepened looking for effective and efficient methods to conduct them [15].



An experiment must be replicated in order to allow the researcher to become more confident about its results. Within a line of research a replication chain is performed and new hypotheses are constructed in order for knowledge to evolve. Replications make it possible to confirm that the results of an experiment have not been random or influenced by researchers. Replications also allow the introduction of variations in the context to extend the scope of the results and explore new variables [12].

Experiments can be organized in families to give coherence to the replications and improve the use of research resources. This concept was defined and put into practice by Basili and others [4]. A family of experiments is a set of experiments and their replications which share the same research objective. The experiments that belong to the family may have a similar design and share materials between replications.

The replications of a family of experiments can be done in several places and be conducted by different researchers. Conducting an experiment is a complex task that demands specific knowledge, as well as material and human resources. In order to facilitate this job, suitable tools to knowledge transfer and to support the experimental process must be used [21].

One of the tools used to support experiment replication are the Laboratory Packages (hereafter LP). LP organize the knowledge concerning an experiment and the necessary materials to conduct it [8]. The LP allow the transfer of knowledge between different researchers and are a long term support of the research line. Although there are different proposals, the problem concerning what the LP should contain remains unsolved. A LP proposal of structure and content is validated in this study by means of its instantiation in a concrete experiment.

The experiment conducted to construct the LP was designed and conducted by the Software Engineering Group (GrIS) of the Computer Science Institute of the School of Engineering of the Universidad de la República of Uruguay (hereafter InCo Experiment). This experiment has been replicated twice, in the years 2008 and 2009. The objective of this experiment is to know the effectiveness and cost of some unit verification techniques [18–20].

In order to pack the family of experiments (comprised of the base experiment and its replications) a LP generic proposal for SE experiments is used [17]. The LP proposal is undergoing an evaluation process and this case is a new point for its validation. Therefore, this study serves a double purpose: instantiate the knowledge concerning the InCo experiment in a LP and add a validation point for the LP generic proposal using a concrete experiment.

2 Related Work

In a broad sense, any piece of written information and materials related to an experiment can be considered an LP. However, the term LP usually refers to the materials that are especially prepared to facilitate the replication. The

LP are oriented towards encouraging and supporting replications, encapsulating materials, methods and experiences related to SE experiments [4].

It is not possible to be thorough about all the experiments in SE that have used this tool. However, some relevant works related to LP are collected in this section. The works considered make specific proposals or analyze their use in replication of experiments in SE.

Brooks and other researchers promote the replication as a necessary step for the evolution of knowledge [5]. On the other hand, it must be pointed out that the construction of LP is a hard task for the researcher. Even so, they said that without LP suitably constructed and documented, cumulative and systematic empirical research would be difficult to achieve.

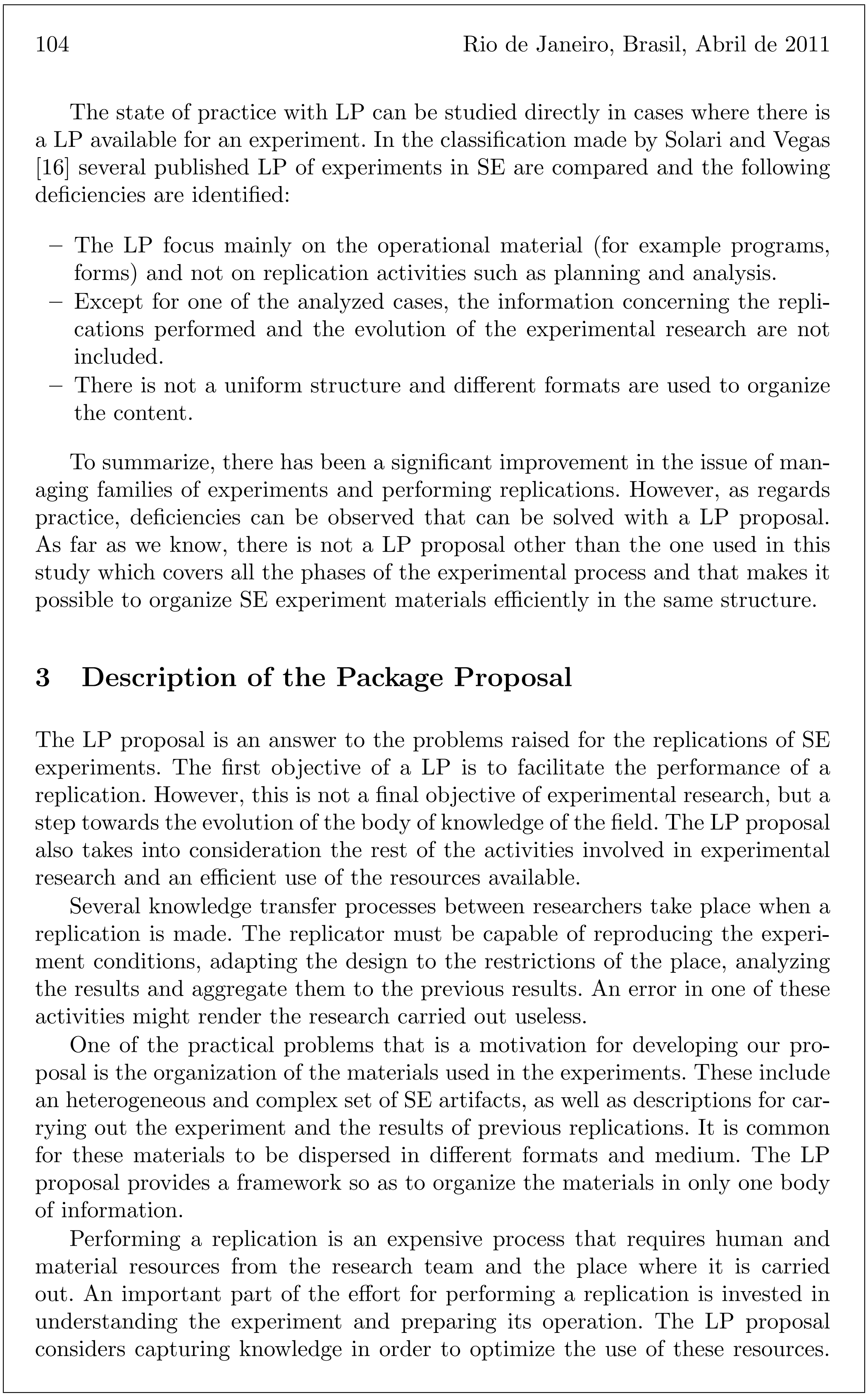
Basili and others conducted an important experience for the development of the concept of family of experiments through replicated experiments on code reading techniques [3]. Several research groups from different countries participated in the research project. An LP was constructed within this frame so that the experiment could be replicated and the techniques transmitted among the members of an experimentation net.

In the same research line the specific problem of knowledge transfer was approached. A model is proposed to capture the tacit knowledge the researchers possess of the experiment and the importance of having an LP to prevent incidents in the replication process. It is said that in order to transmit the tacit knowledge effectively it is necessary to go beyond the simple description of the operational material; the LP must include valid design alternatives and grounding of the experimental processes [14]. In that study a LP proposal, validated with a specific experiment, is made.

The problems posed by the use of communication instruments by researchers were approached by Juristo and others by comparing several replications. Different communication mechanisms and instruments were applied in each replication: meetings, subsequent consultations and different types of LP. The conclusions of the study indicate that a way of achieving successful replications is to have a detailed LP supported by a minimal process of communication between the researchers [10].

It is usual for scientific communities to propose specific guidelines for the publication of the results of controlled experiments. Empirical SE is no exception and the guidelines presented by Jedlitschka and Pfahl are a common reference for the publication of results [9]. A structured description of the experiment is promoted in that proposal so as to facilitate understanding. These guidelines must be taken into account for the construction of a LP adding other elements necessary to make a replication.

Mendonça and others use a dynamic approach to improve the LP by using observational studies [13]. They suggest studying the LP during the process of performing the replication itself. According to the authors, the experimental process must include the packaging of the experience and the conduction of a pilot study with the LP. Thus an improved LP, that facilitates the replication of the experiment in the future, is produced.



The state of practice with LP can be studied directly in cases where there is a LP available for an experiment. In the classification made by Solari and Vegas [16] several published LP of experiments in SE are compared and the following deficiencies are identified:

- The LP focus mainly on the operational material (for example programs, forms) and not on replication activities such as planning and analysis.
- Except for one of the analyzed cases, the information concerning the replications performed and the evolution of the experimental research are not included.
- There is not a uniform structure and different formats are used to organize the content.

To summarize, there has been a significant improvement in the issue of managing families of experiments and performing replications. However, as regards practice, deficiencies can be observed that can be solved with a LP proposal. As far as we know, there is not a LP proposal other than the one used in this study which covers all the phases of the experimental process and that makes it possible to organize SE experiment materials efficiently in the same structure.

3 Description of the Package Proposal

The LP proposal is an answer to the problems raised for the replications of SE experiments. The first objective of a LP is to facilitate the performance of a replication. However, this is not a final objective of experimental research, but a step towards the evolution of the body of knowledge of the field. The LP proposal also takes into consideration the rest of the activities involved in experimental research and an efficient use of the resources available.

Several knowledge transfer processes between researchers take place when a replication is made. The replicator must be capable of reproducing the experiment conditions, adapting the design to the restrictions of the place, analyzing the results and aggregate them to the previous results. An error in one of these activities might render the research carried out useless.

One of the practical problems that is a motivation for developing our proposal is the organization of the materials used in the experiments. These include an heterogeneous and complex set of SE artifacts, as well as descriptions for carrying out the experiment and the results of previous replications. It is common for these materials to be dispersed in different formats and medium. The LP proposal provides a framework so as to organize the materials in only one body of information.

Performing a replication is an expensive process that requires human and material resources from the research team and the place where it is carried out. An important part of the effort for performing a replication is invested in understanding the experiment and preparing its operation. The LP proposal considers capturing knowledge in order to optimize the use of these resources.

For example, the proposal includes a list of activities necessary to carry out a replication and the estimated effort for each one.

Given the evolutionary nature of experimental research, the LP must be a flexible instrument. It must incorporate the material from different versions of the experiment as well as the descriptions of each replication and the results that are obtained. The LP must help the researcher to be acquainted with the evolution of the research line and to adapt the replication to the place where it is going to be performed.

To summarize, the LP proposal tries to consider the following challenges:

– Transfer SE and experimental knowledge to make a replication.
– Organize the material in one body of information.
– Optimize the use of research resources.
– Facilitate the adaptation of the experiment to the context.
– Make it possible to aggregate the results and the evolution of the research.

The LP proposal consists of three elements: a generic description of the structure that must have a specific LP, a template and a check list of the content. Thus, it can be used as a guideline in the process of organizing the knowledge and the materials of a SE experiment. In [17] a detailed description of it is made.

The proposed structure divides the LP in several inter-related modules that are outlined in figure 1. Each of the modules supports an activity of the experimental research process. When structuring the content, the fact that the LP is a technical document that can be read in a non-linear way and used for different purposes has been taken into account. The module with the greatest level of

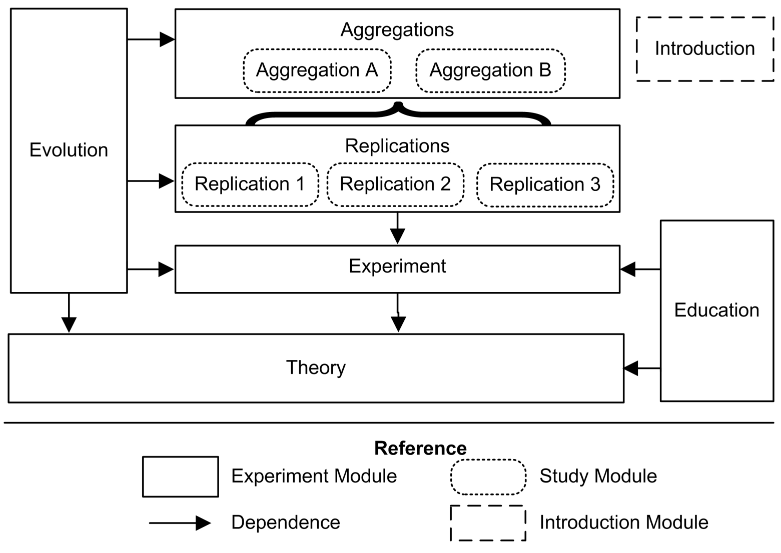


Fig. 1. Modules of the LP proposal.

abstraction is that of the **theory** of the experiment, which sets the theoretical frame for conducting the experiment and the interpretation of the results. In the module **education** the materials used for training the subjects that are going to take part in the experiment are collected. The general aspects of the experiment and the instructions for replicating it are defined in the module **experiment**. This module is organized according to the activities of the replication process: planning, experimental design, operation and analysis. Apart from the instructions for each activity, operational material and the necessary tools to replicate the experiment are included.

The structure also considers the evolutionary dynamics of experimental research. In the module **evolution** a summary of the studies carried out is done. This module serves the purpose of establishing the relationships between the versions of the experiment and its replications as well as to see the evolution of the experiment family as a whole. The modules **replications** and **aggregations** contain the information of each study of the experiment family.

Since it is a technical operational document, the LP must satisfy certain content quality criteria. In order to achieve these quality standards a check list that is used in the reviews of the LP is included. The aspects considered in the review are: orientation to the task, organization of the operational material, support of the experimental process, format and writing style.

4 Study Case: InCo Experiment

This section describes the instantiation of the LP generic proposal for the InCo experiment, indicating the process followed and the incidents registered. The discussion section presents the lessons learned about the instantiation of the LP.

The process used for the instantiation of LP, which consists of 6 activities, is shown in figure 2. The study of the LP proposal, in which its structure and each one of its elements is analyzed, is the necessary first step to be able to generate the LP. Then the available sources of information are identified and a decision is made as to which of them will be considered. The documentation implies the generation of the LP document and the required adjoining material. After a version of the LP is generated, it is necessary to verify and validate it. During the whole process consultation and monitoring activities of the progress are carried out and there are reviews of the intermediate results. The maintenance activity follows the release of the LP, where the LP must be updated as the research evolves.

Three roles participated in this process: the instantiator of the LP, the author of the LP proposal and the responsible researcher. The instantiator of the LP is in charge of generating the LP document and the additional material which may be necessary. The author of the LP proposal participates in the verification, consultation and monitoring activities, focusing on the use of the LP proposal. The responsible researcher participates in the validation of the result, from the point of view of the experiment, as refers to the completeness and adequation of the information documented.

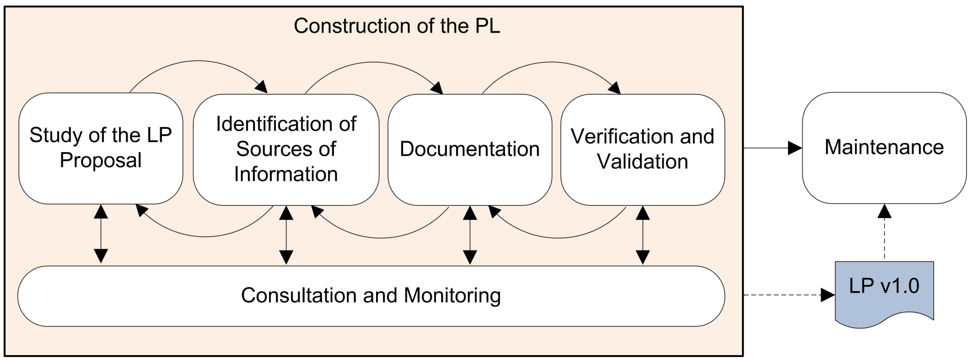

Construction of the PL
Study of the LP Proposal
Identification of Sources of Information
Documentation
Verification and Validation
Maintenance
Consultation and Monitoring
LP v1.0

Fig. 2. LP Instantiation Process

4.1 LP Instantiation

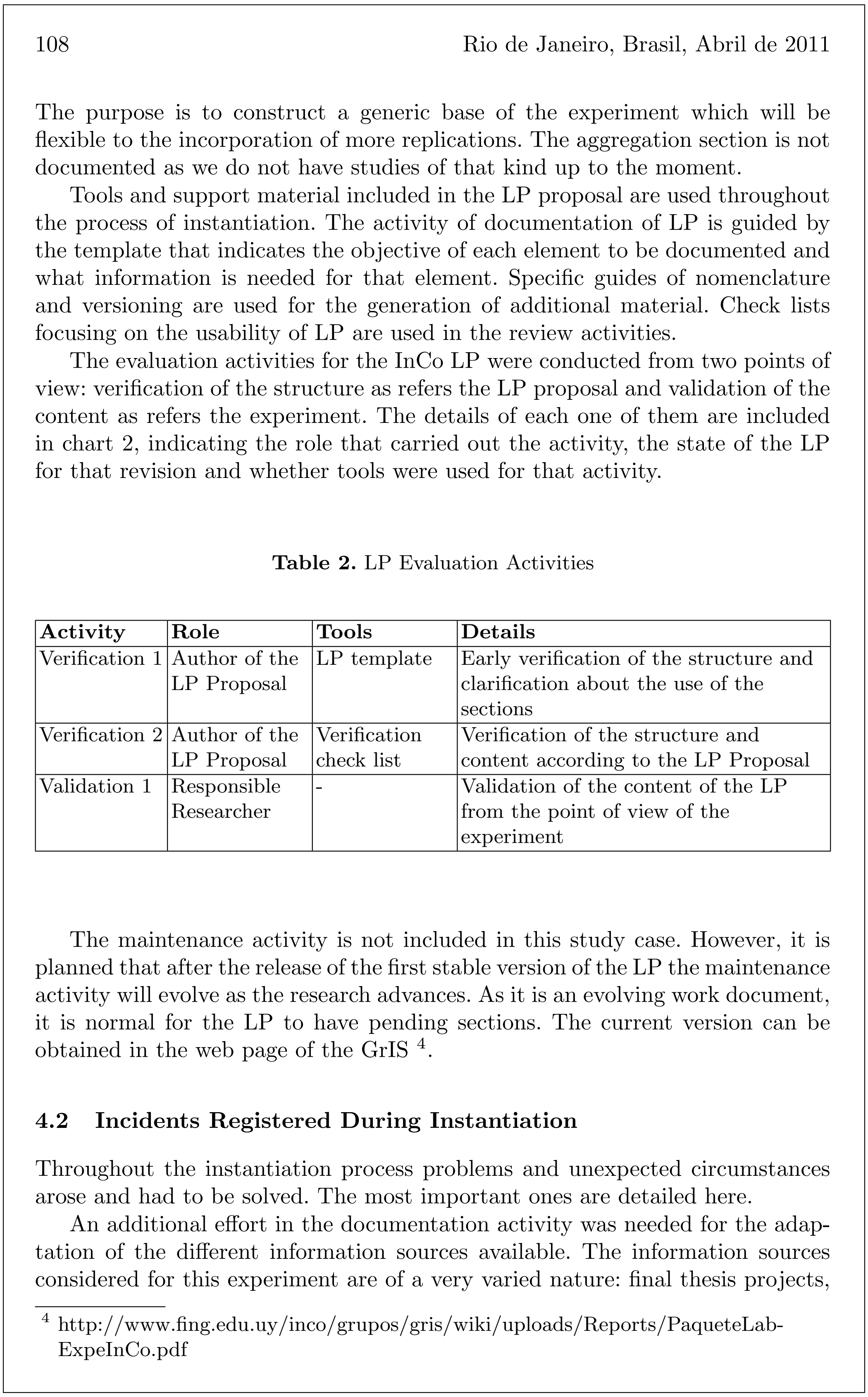
The effort required for the study of the LP proposal varies depending on the knowledge the instantiator of the LP has about the experiment and empirical SE concepts. In our case the instantiator of the LP had the necessary knowledge about empirical SE and was part of the research team which carried out the InCo experiment. However, it was still necessary to clarify doubts through meetings with the author of the LP proposal, in which there was consultation about the objective of each LP component and the level of detail necessary for each objective.

In all, there were 10 monitoring meetings during the construction of the LP, in which everything documented up to that moment was reviewed and doubts were clarified. The construction of the LP was therefore carried out through an iterative and incremental process. The summary of the meetings held during the instantiation are detailed in table 1.

Table 1. Consultation meetings during the LP instantiation

Meeting	**Phase**	**Observations**
1, 2, 3	Study of the LP proposal	Clarifying doubts about the template, the information required and the priority elements to be included.
4, 5, 6, 7	Identification of Information Sources	Review of the documented LP, specific doubts.
8, 9 y 10	Verification and Validation	Corrections derived from the Verification and external Validation of the draft versions of the LP.

At the moment of identifying the sources of information and the elements to be included, we decided to use the replication corresponding to the year 2008.



The purpose is to construct a generic base of the experiment which will be flexible to the incorporation of more replications. The aggregation section is not documented as we do not have studies of that kind up to the moment.

Tools and support material included in the LP proposal are used throughout the process of instantiation. The activity of documentation of LP is guided by the template that indicates the objective of each element to be documented and what information is needed for that element. Specific guides of nomenclature and versioning are used for the generation of additional material. Check lists focusing on the usability of LP are used in the review activities.

The evaluation activities for the InCo LP were conducted from two points of view: verification of the structure as refers the LP proposal and validation of the content as refers the experiment. The details of each one of them are included in chart 2, indicating the role that carried out the activity, the state of the LP for that revision and whether tools were used for that activity.

Table 2. LP Evaluation Activities

Activity	**Role**	**Tools**	**Details**
Verification 1	Author of the LP Proposal	LP template	Early verification of the structure and clarification about the use of the sections
Verification 2	Author of the LP Proposal	Verification check list	Verification of the structure and content according to the LP Proposal
Validation 1	Responsible Researcher	-	Validation of the content of the LP from the point of view of the experiment

The maintenance activity is not included in this study case. However, it is planned that after the release of the first stable version of the LP the maintenance activity will evolve as the research advances. As it is an evolving work document, it is normal for the LP to have pending sections. The current version can be obtained in the web page of the GrIS [4].

4.2 Incidents Registered During Instantiation

Throughout the instantiation process problems and unexpected circumstances arose and had to be solved. The most important ones are detailed here.

An additional effort in the documentation activity was needed for the adaptation of the different information sources available. The information sources considered for this experiment are of a very varied nature: final thesis projects,

[4] http://www.fing.edu.uy/inco/grupos/gris/wiki/uploads/Reports/PaqueteLab-ExpeInCo.pdf

technical reports, text documents, spreadsheets, databases, emails and the tacit knowledge the researchers have about the experiment.

In some cases the passing of information from the sources to the LP was direct and in others it was necessary to restructure, enlarge or summarize the information. In the cases in which the only source of information was the knowledge and experience of the researcher, it was necessary to generate all the documentation. The sources of information for each element of the LP are described in table 3. The percentage of reuse is calculated through an approximation of the documentation generated against the documentation from the existing one.

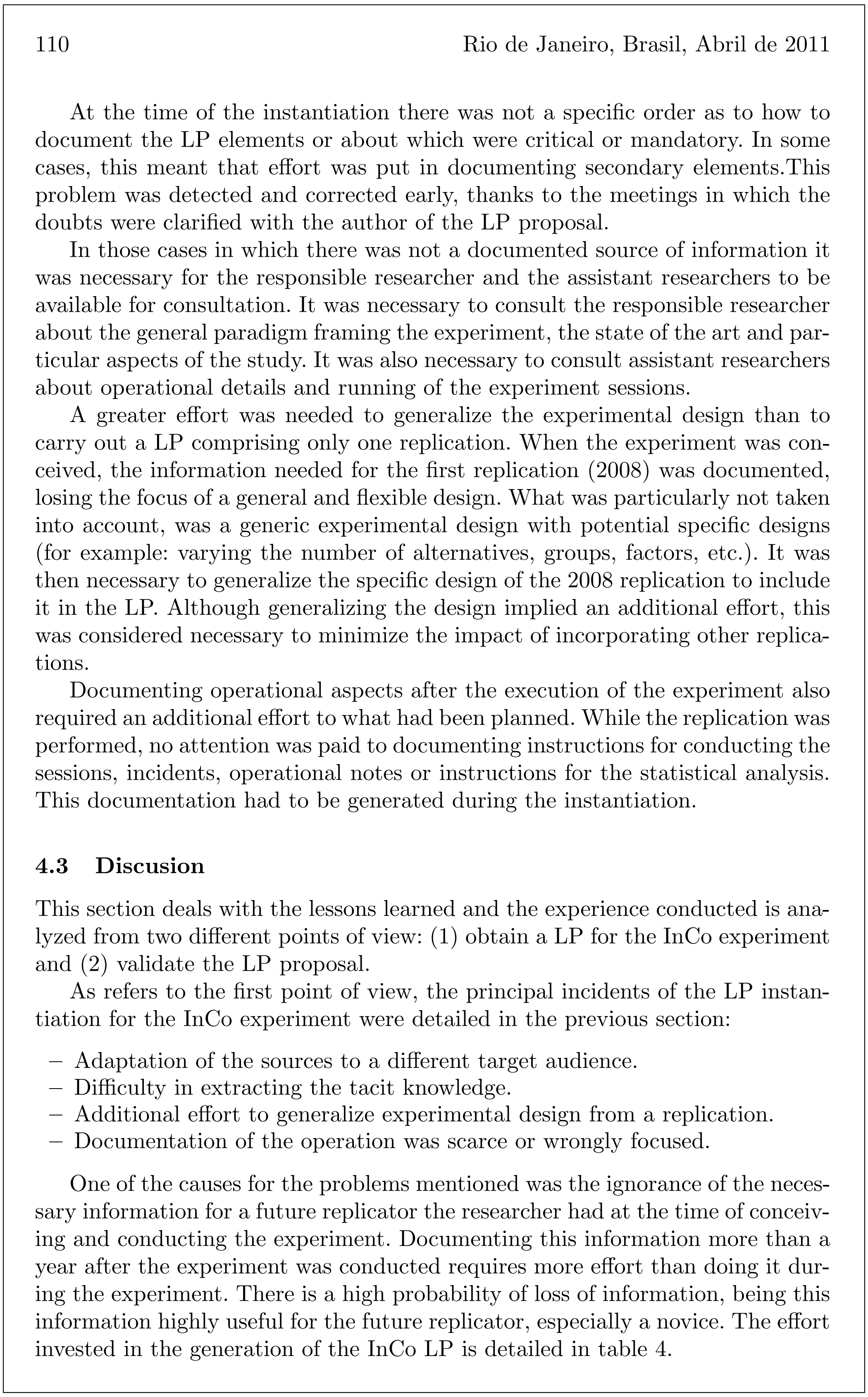
Table 3. Information Sources for the LP Elements

Element	**Information Sources**	**% Reuse**
Theory	Final thesis projects, technical reports, tacit knowledge	70%
Experiment	Final thesis projects, electronic mails, databases, other documents, tacit knowledge	60%
2008 Replication	Final thesis projects, tacit knowledge	90%
Education	Final thesis projects, theoretical presentation slides	80%

Depending on the source of information more or less restructuring of the data or generation of the documentation was needed. Most of the information was obtained reutilizing the documentation of two final thesis projects that detail the design and execution of the 2008 replication [7, 1] and a technical report which describes the general concepts of empirical SE [2]. This information required little or moderate restructuring in both cases. The biggest problems at the moment of processing these sources of information were:

– Large volumes of information: The final thesis projects have about 100 pages each and the technical report 22.
– Repeated information: Both final thesis projects describe the 2008 replication and information about empirical SE and concepts presented in the technical report are repeated.
– Different target audience: the information is not oriented to a replicator but to an academic committee who evaluates the knowledge of the students as refers to the work completed.

More effort was needed to restructure and adapt the information obtained form other documents to the objectives of the LP element to which it was destined. The biggest adaptation and restructuring effort was to change the target audience: most of the information was destined to the researchers (who had a tacit knowledge about the experiment) or to the subjects of the experiment and not to a replicator.



At the time of the instantiation there was not a specific order as to how to document the LP elements or about which were critical or mandatory. In some cases, this meant that effort was put in documenting secondary elements.This problem was detected and corrected early, thanks to the meetings in which the doubts were clarified with the author of the LP proposal.

In those cases in which there was not a documented source of information it was necessary for the responsible researcher and the assistant researchers to be available for consultation. It was necessary to consult the responsible researcher about the general paradigm framing the experiment, the state of the art and particular aspects of the study. It was also necessary to consult assistant researchers about operational details and running of the experiment sessions.

A greater effort was needed to generalize the experimental design than to carry out a LP comprising only one replication. When the experiment was conceived, the information needed for the first replication (2008) was documented, losing the focus of a general and flexible design. What was particularly not taken into account, was a generic experimental design with potential specific designs (for example: varying the number of alternatives, groups, factors, etc.). It was then necessary to generalize the specific design of the 2008 replication to include it in the LP. Although generalizing the design implied an additional effort, this was considered necessary to minimize the impact of incorporating other replications.

Documenting operational aspects after the execution of the experiment also required an additional effort to what had been planned. While the replication was performed, no attention was paid to documenting instructions for conducting the sessions, incidents, operational notes or instructions for the statistical analysis. This documentation had to be generated during the instantiation.

4.3 Discusion

This section deals with the lessons learned and the experience conducted is analyzed from two different points of view: (1) obtain a LP for the InCo experiment and (2) validate the LP proposal.

As refers to the first point of view, the principal incidents of the LP instantiation for the InCo experiment were detailed in the previous section:

- Adaptation of the sources to a different target audience.
- Difficulty in extracting the tacit knowledge.
- Additional effort to generalize experimental design from a replication.
- Documentation of the operation was scarce or wrongly focused.

One of the causes for the problems mentioned was the ignorance of the necessary information for a future replicator the researcher had at the time of conceiving and conducting the experiment. Documenting this information more than a year after the experiment was conducted requires more effort than doing it during the experiment. There is a high probability of loss of information, being this information highly useful for the future replicator, especially a novice. The effort invested in the generation of the InCo LP is detailed in table 4.

Table 4. LP construction effort

Activity	**Effort (hours)**
Study of the LP proposal	8.0
Monitoring and control meetings	11.5
Documentation	44.0
Generation of attached material	6.5
Total effort	*70.0*

Although it is strongly recommended that the LP should be generated from the early stages and throughout the conduction of the experiment, it is also beneficial to do it in later stages. It allows a post-mortem analysis of the experiment, improving the documentation generated and allowing the reflection on aspects that had not been previously considered. This generates a compendium of lessons learned that will help to improve the results in the future replications, with less effort and more precision.

The reviews that were carried out parallel to the instantiation process were useful to improve the final result. One of the objectives was to obtain a LP adequate for the experiment. Another objective was to adhere to the LP proposal. These reviews collaborated in improving the quality of the instantiated LP for the InCo experiment and provided improvement suggestions for the LP proposal.

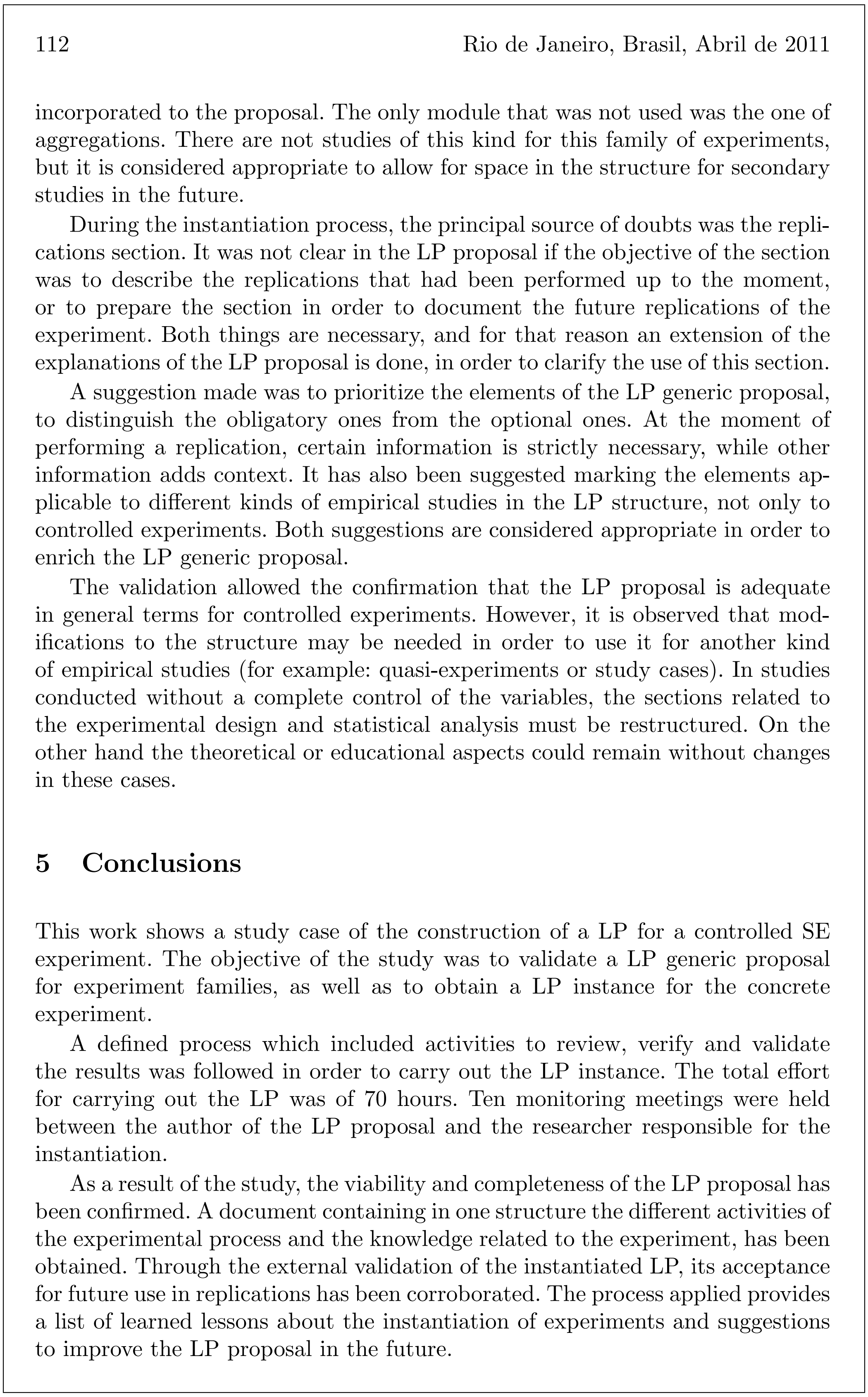
The cases that showed deviation from the proposed structure were analyzed to decide whether they corresponded to the experiment or not. Of the 35 sections and sub-sections of the proposal, 7 were not used or remained pending of completion, either because there was not information at the time, or because they were not critical sections. Four unplanned sections were added to the original structure. For this reason, it is considered that most of the proposal was adequate to the InCo experiment.

From the point of view of the LP generic proposal validation this study case has been useful to confirm the following aspects:

– Viability: it was possible to instantiate a LP for a concrete experiment.
– Completeness: the LP obtained covers all the activities of the experimental process.
– Acceptation: the result is potentially usable in replications and improvement of the previous organization of the material.

Comparing the result obtained with the original proposal it is observed that there have not been significant changes in the structure. This is also explained because the experiment used in this case is relatively similar to the ones used in the previous series of evaluations that originated the proposal. However, because it is a validation, it is desirable to identify which the potential improvements are in order to have feed back and improve the original proposal.

The most significant aggregations to the LP were two: a section about experimental theory and the description of the context parameters. Both changes were evaluated during the instantiation process and are considered appropriate to be



incorporated to the proposal. The only module that was not used was the one of aggregations. There are not studies of this kind for this family of experiments, but it is considered appropriate to allow for space in the structure for secondary studies in the future.

During the instantiation process, the principal source of doubts was the replications section. It was not clear in the LP proposal if the objective of the section was to describe the replications that had been performed up to the moment, or to prepare the section in order to document the future replications of the experiment. Both things are necessary, and for that reason an extension of the explanations of the LP proposal is done, in order to clarify the use of this section.

A suggestion made was to prioritize the elements of the LP generic proposal, to distinguish the obligatory ones from the optional ones. At the moment of performing a replication, certain information is strictly necessary, while other information adds context. It has also been suggested marking the elements applicable to different kinds of empirical studies in the LP structure, not only to controlled experiments. Both suggestions are considered appropriate in order to enrich the LP generic proposal.

The validation allowed the confirmation that the LP proposal is adequate in general terms for controlled experiments. However, it is observed that modifications to the structure may be needed in order to use it for another kind of empirical studies (for example: quasi-experiments or study cases). In studies conducted without a complete control of the variables, the sections related to the experimental design and statistical analysis must be restructured. On the other hand the theoretical or educational aspects could remain without changes in these cases.

5 Conclusions

This work shows a study case of the construction of a LP for a controlled SE experiment. The objective of the study was to validate a LP generic proposal for experiment families, as well as to obtain a LP instance for the concrete experiment.

A defined process which included activities to review, verify and validate the results was followed in order to carry out the LP instance. The total effort for carrying out the LP was of 70 hours. Ten monitoring meetings were held between the author of the LP proposal and the researcher responsible for the instantiation.

As a result of the study, the viability and completeness of the LP proposal has been confirmed. A document containing in one structure the different activities of the experimental process and the knowledge related to the experiment, has been obtained. Through the external validation of the instantiated LP, its acceptance for future use in replications has been corroborated. The process applied provides a list of learned lessons about the instantiation of experiments and suggestions to improve the LP proposal in the future.

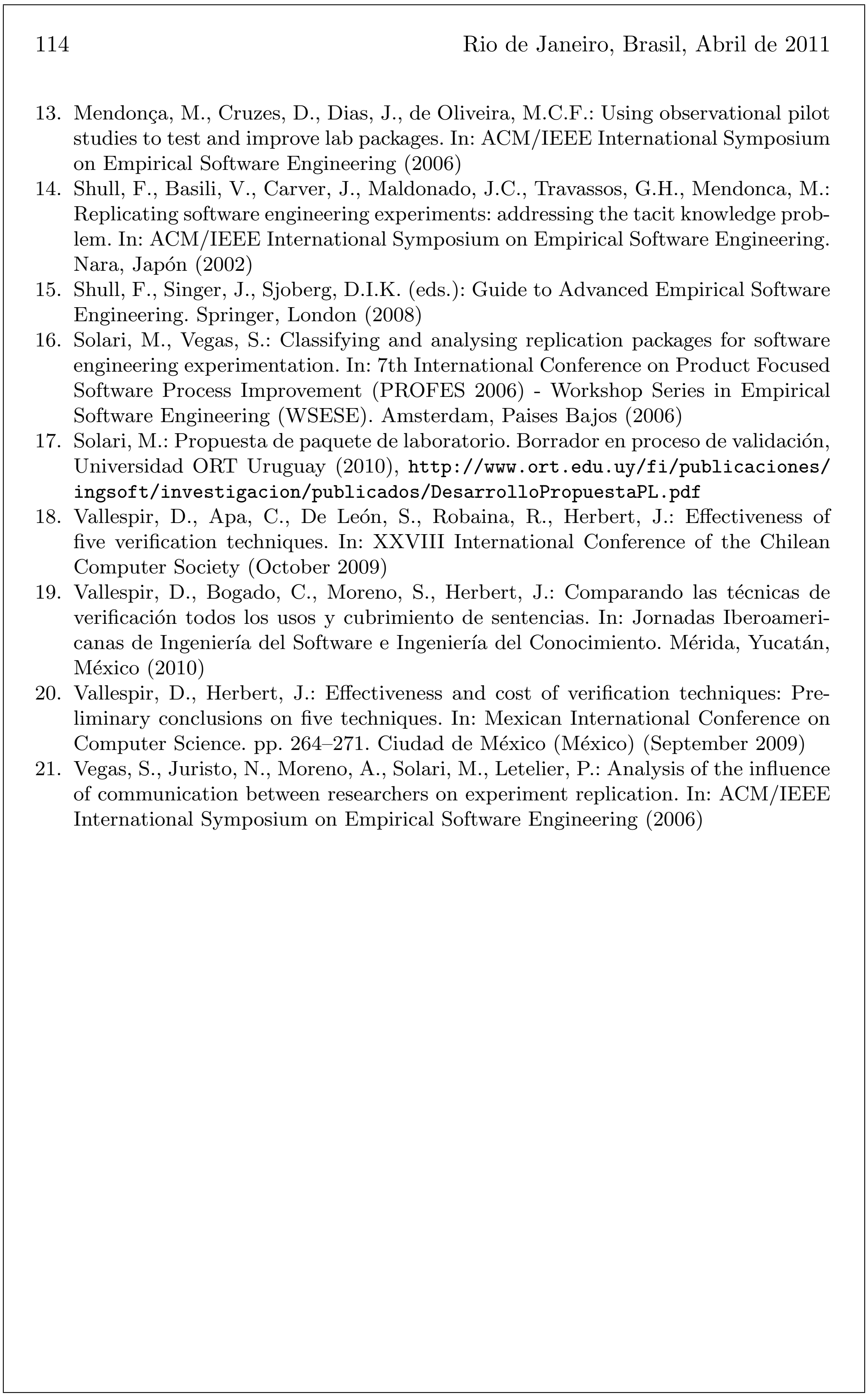
It has also been observed that it is possible to instantiate a LP sometime after the experiment was conducted. However, we strongly recommend compiling the necessary information during the process of experimentation. The structure of the LP proposal can be used as a guide to collect this knowledge while conducting the experiment.

The incorporation of the 2009 replication of the InCo experiment is pending for future work. This incorporation will allow the evaluation of the impact of the proposal at the time of aggregating this information to a LP already instantiated.

Another future line for this research is the validation of the LP for other empirical studies and for scenarios of use progressively different from the InCo. For example, the instantiation of LP for quasi-experiments and other empirical studies must be studied. The use of LP in a different context from that of the replication, for example the aggregation of results of various replications must also be considered.

References

1. Apa, C.: Diseño y Ejecución de un Experimento con 5 Técnicas de Verificación Unitaria. Tesis de fin de carrera, Facultad de Ingeniería, Universidad de la República, Uruguay (2009)
2. Apa, C., Robaina, R., De León, S., Vallespir, D.: Conceptos de ingeniería de software empírica. Reporte técnico 10-02 InCo PEDECIBA-Informática 0797–6410, Facultad de Ingeniería, Universidad de la República, Uruguay (2010)
3. Basili, V.R.: Evolving and packaging reading technologies. Journal of Systems and Software 38(1), 3–12 (1997)
4. Basili, V.R., Shull, F., Lanubile, F.: Building knowledge through families of experiments. IEEE Transactions on Software Engineering 25(4), 456–473 (1999)
5. Brooks, A., Daly, J., Miller, J., Roper, M., Wood, M.: Replication of experimental results in software engineering. Tech. rep., International Software Enginnering Research Network (ISERN) (1996)
6. Campbell, D.T., Stanley, J.: Experimental and Quasi-Experimental Designs for Research. Wadsworth Publishing (1963)
7. De León, S., Robaina, R.: Análisis de la Efectividad y el Costo de 5 Técnicas de Verificación. Tesis de fin de carrera, Facultad de Ingeniería, Universidad de la República, Uruguay (2009)
8. Green, B., Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S., Zelkowitz, M.: Packaging researcher experience to assist replication of experiments. Tech. rep., International Software Enginnering Research Network (ISERN) (1996)
9. Jedlitschka, A., Pfahl, D.: Reporting guidelines for controlled experiments in software engineering. In: ACM/IEEE International Symposium on Empirical Software Engineering (2005)
10. Juristo, N., Moreno, A., Vegas, S., Solari, M.: In search of what we experimentally know about unit testing. IEEE Software 23(6), 72–80 (2006)
11. Juristo, N., Moreno, A.M.: Basics of Software Engineering Experimentation. Springer (2001)
12. Juristo, N., Vegas, S.: Using differences among replications of software engineering experiments to gain knowledge. Empirical Software Engineering and Measurement pp. 356–366 (2009)



13. Mendonça, M., Cruzes, D., Dias, J., de Oliveira, M.C.F.: Using observational pilot studies to test and improve lab packages. In: ACM/IEEE International Symposium on Empirical Software Engineering (2006)
14. Shull, F., Basili, V., Carver, J., Maldonado, J.C., Travassos, G.H., Mendonca, M.: Replicating software engineering experiments: addressing the tacit knowledge problem. In: ACM/IEEE International Symposium on Empirical Software Engineering. Nara, Japón (2002)
15. Shull, F., Singer, J., Sjoberg, D.I.K. (eds.): Guide to Advanced Empirical Software Engineering. Springer, London (2008)
16. Solari, M., Vegas, S.: Classifying and analysing replication packages for software engineering experimentation. In: 7th International Conference on Product Focused Software Process Improvement (PROFES 2006) - Workshop Series in Empirical Software Engineering (WSESE). Amsterdam, Paises Bajos (2006)
17. Solari, M.: Propuesta de paquete de laboratorio. Borrador en proceso de validación, Universidad ORT Uruguay (2010), `http://www.ort.edu.uy/fi/publicaciones/ingsoft/investigacion/publicados/DesarrolloPropuestaPL.pdf`
18. Vallespir, D., Apa, C., De León, S., Robaina, R., Herbert, J.: Effectiveness of five verification techniques. In: XXVIII International Conference of the Chilean Computer Society (October 2009)
19. Vallespir, D., Bogado, C., Moreno, S., Herbert, J.: Comparando las técnicas de verificación todos los usos y cubrimiento de sentencias. In: Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento. Mérida, Yucatán, México (2010)
20. Vallespir, D., Herbert, J.: Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. In: Mexican International Conference on Computer Science. pp. 264–271. Ciudad de México (México) (September 2009)
21. Vegas, S., Juristo, N., Moreno, A., Solari, M., Letelier, P.: Analysis of the influence of communication between researchers on experiment replication. In: ACM/IEEE International Symposium on Empirical Software Engineering (2006)

ARTICLE

Calidad de Datos en Experimentos en Ingeniería de Software: Un Caso de Estudio

Carolina Valverde, Adriana Marotta and Diego Vallespir

Submitted to a Conference. Article in Spanish.

Calidad de Datos en Experimentos en Ingeniería de Software Un caso de estudio

Carolina Valverde, Adriana Marotta y Diego Vallespir

Facultad de Ingeniería, Universidad de la República
{mvalverde,amarotta,dvallesp}@fing.edu.uy

Resumen Este trabajo presenta un estudio de la calidad de los datos de un experimento controlado en ingeniería de software. Se presentan los tipos de errores que es posible encontrar en un experimento de estas características basándose en un caso de estudio. Para cada uno de los tipos de errores definimos errores concretos que pueden tener los datos, definimos cómo medirlos (de forma manual o automática) en la base de datos del experimento y ejecutamos las mediciones. Los resultados de este caso de estudio indican que los investigadores que realizan experimentos en ingeniería de software deben analizar la calidad de los datos del experimento antes de realizar los análisis estadísticos del mismo. De esta forma los resultados del experimento reflejarán más fielmente la realidad.

1. Introducción

Los datos que mantiene una organización (información) son fundamentales para el funcionamiento de la misma. El análisis de esos datos influye directamente en la toma de decisiones. Lamentablemente, los datos pueden contener errores. Esto provoca que el análisis brinde resultados incorrectos y entonces que las decisiones sean las equivocadas. La disciplina Calidad de Datos estudia la calidad de los datos y propone formas de medir la misma y de realizar limpiezas (correcciones) en los datos de mala calidad.

La Ingeniería de Software Empírica busca, a partir de la experimentación, conocer si ciertos supuestos sobre el desarrollo de software son reales. Durante el proceso de experimentación se genera gran cantidad de datos. Sobre estos datos se realizan análisis estadísticos y estudios comparativos. Por último, se establecen resultados y conclusiones que surgen del propio análisis realizado sobre los datos.

Se debe tener en cuenta que los datos de los experimentos pueden ser de mala calidad. Esto provoca que los resultados del mismo sean cuestionables. Incluso, si la calidad de los datos está comprometida en un alto porcentaje de los datos, o con un error "grande" en un pequeño porcentaje de los mismos, los resultados del experimento pueden ser incorrectos. En ese caso se puede incluso validar una realidad que no existe.

En la Facultad de Ingeniería, de la Universidad de la República en Uruguay, realizamos un experimento controlado para evaluar la efectividad y costo de distintas técnicas de pruebas unitarias. Distintos estudiantes de la Facultad participaron en el experimento. Estos aplicaron ciertas técnicas de pruebas sobre distintos programas. Mientras realizaban las pruebas registraban datos como el tiempo que les llevó ejecutar la técnica y cuáles fueron los defectos detectados [8].

En este artículo presentamos un estudio de la calidad de los datos recolectados por los sujetos durante dicho experimento. El objetivo es conocer la calidad de los mismos y poder realizar una limpieza de datos que luego permita que los resultados de los análisis estadísticos del experimento sean más confiables. Para ello, se identifican y miden los errores que contienen los datos bajo estudio, para luego aplicar las limpiezas correspondientes.

La Figura 1 presenta el trabajo completo que fue realizado. Luego de ejecutado el experimento se identificaron los posibles tipos de errores de los datos. Para cada tipo de error se identificaron errores específicos que fueron medidos y registrados. Luego se realizó la limpieza y migración. Este artículo presenta los tipos de errores, errores y la medición realizada en la base de datos del caso de estudio.

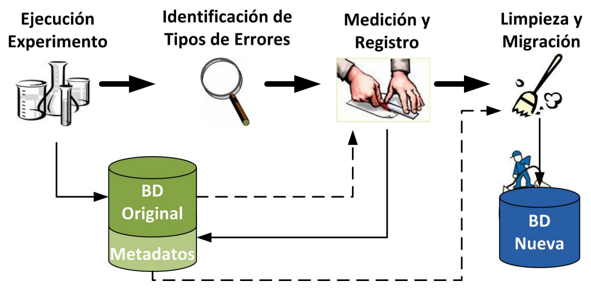


Figura 1. Etapas del estudio realizado.

El aporte de este trabajo es haber identificado y medido los tipos de errores que se pueden encontrar en experimentos en ingeniería de software. Más en particular en experimentos que estudian la efectividad y costo de las técnicas de pruebas. Mostramos que los datos de los experimentos (obviamente) contienen errores y que estos deben ser limpiados antes de realizar cualquier análisis estadístico. No encontramos en la literatura otros estudios de Calidad de Datos en experimentos en ingeniería de software, por lo que este caso de estudio aporta también como un estudio inicial en este sentido.

2. Calidad de Datos

Los datos constituyen un recurso muy valioso para las organizaciones al ser utilizados principalmente para la toma de decisiones. Estos pueden ser almacenados, comunicados o sometidos a algún proceso o transformación, siendo en todos los casos de suma importancia para garantizar la sobrevivencia y éxito de las organizaciones.

La calidad de datos es objeto de estudio de diferentes áreas, tal es el caso de la estadística, gestión o computación. A medida que su importancia se hace más evidente se incrementan las investigaciones e intenciones de mejora en la calidad de los datos que una organización utiliza.

La mala calidad de los datos influye de manera significativa y profunda en la efectividad y eficiencia de las organizaciones así como en todo el negocio, llevando en algunos casos a pérdidas multimillonarias [1]. Cada día se hace más notoria la importancia y necesidad en distintos contextos de un nivel de calidad adecuado para los datos. Por esto, es importante lograr identificar las causas por las cuales ciertos datos son de mala calidad, para eliminar, o en su defecto mejorar, la problemática de raíz.

2.1. Dimensiones y Factores de Calidad de Datos

Existen distintos aspectos que hacen a la calidad de datos. Estos aspectos son conocidos normalmente como dimensiones de calidad. En los trabajos del área de Calidad de Datos se propone gran variedad de conjuntos de dimensiones y de definiciones para las mismas [5,6,7,9]. Sin embargo existe un núcleo de dimensiones que es compartido por la mayoría de las propuestas. Este trabajo se basa fuertemente en la propuesta de Batini y Scannapieco [1], que reúne estas dimensiones consensuadas.

En este trabajo utilizamos una abstracción de la calidad de datos, presentada en [4], donde además de las dimensiones se definen otros conceptos para la clasificación y el manejo de la misma. Estos conceptos son el de factor, métrica y método de medición.

Una dimensión de calidad captura una faceta (a alto nivel) de la calidad de los datos. Por otra parte, un factor de calidad representa un aspecto particular de una dimensión de calidad. Una dimensión puede ser entendida como un agrupamiento de factores que tienen el mismo propósito de calidad.

Una métrica es un instrumento que define la forma de medir un factor de calidad. Un mismo factor de calidad puede medirse con diferentes métricas. A su vez, un método de medición es un proceso que implementa una métrica. Se pueden utilizar distintos métodos de medición para una misma métrica.

Normalmente, los datos se encuentran en algún repositorio de datos, siendo de los más usados las bases de datos relacionales. En nuestro caso de estudio los datos se encuentran en una base de datos relacional y por eso es de especial interés para este trabajo considerar las mediciones de la calidad de los datos es ese tipo de repositorio.

Las mediciones en una base de datos relacional se pueden realizar a varios niveles: celda, tupla, tabla, e incluso a nivel de la base de datos entera. En definitiva, se pueden considerar distintos niveles de granularidad para evaluar la calidad de los datos. Por esto se definen funciones de agregación, las cuales miden un conjunto de datos. Por ejemplo, es posible obtener la medida de una tupla a partir de la medida de exactitud de cada una de sus celdas.

A continuación se presentan las dimensiones y factores de calidad utilizadas en este trabajo. De la propuesta de Batini y Scannapieco [1], no se considera la dimensión frescura relacionada con el tiempo y la vigencia de los datos, ya que no tiene aplicabilidad en este caso de estudio. Esto se debe a que los datos del experimento se consideran “frescos” y no resulta de interés medir si los mismos se encuentran vigentes, el momento en que fueron recolectados es un dato conocido.

Dimensión: Exactitud

La exactitud indica que tan precisos, válidos y libre de errores están los datos. Establece si existe una correcta y precisa asociación entre los estados del sistema de información y los objetos del mundo real.

Existen tres factores de exactitud: exactitud semántica, exactitud sintáctica y precisión. La exactitud semántica se refiere a la cercanía que existe entre un valor v y un valor real v’. Interesa medir que tan bien se encuentran representados los estados del mundo real en el sistema de información.

La exactitud sintáctica se refiere a la cercanía entre un valor v y los elementos de un dominio D. Interesa saber si v corresponde a algún valor válido de D, sin importar si ese valor corresponde a uno del mundo real.

La precisión, por otra parte, se refiere al nivel de detalle de los datos.

Dimensión: Completitud

La completitud indica si el sistema de información contiene todos los datos de interés, y si los mismos cuentan con el alcance y profundidad que sea requerido. Establece la capacidad del sistema de información de representar todos los estados significativos de una realidad dada.

Existen dos factores de la completitud: cobertura y densidad. La cobertura se refiere a la porción de datos de la realidad que se encuentran contenidos en el sistema de información.

La densidad se refiere a la cantidad de información contenida y faltante acerca de las entidades del sistema de información.

En un modelo relacional la completitud se caracteriza principalmente por los valores nulos, cuyo significado a pesar de ser variado, es importante conocer. Un nulo puede indicar que dicho valor no existe, que existe pero no se conoce, o que no se sabe si existe en el mundo real.

Dimensión: Consistencia

Esta dimensión hace referencia al cumplimiento de las reglas semánticas que son definidas sobre los datos. La inconsistencia de los datos se hace presente cuando existe más de un estado del sistema de información asociado al mismo objeto de la realidad.

Las restricciones de integridad, por otra parte, definen propiedades que deben cumplirse por todas las instancias de un esquema relacional. Se distinguen tres tipos de restricciones de integridad, las cuales se corresponden con los factores de esta dimensión.

Las restricciones de dominio, se refieren a la satisfacción de reglas sobre el contenido de los atributos de una relación. Las restricciones intra-relacionales, se refieren a la satisfacción de reglas sobre uno o varios atributos de una relación.

Las restricciones inter-relacionales, se refieren a la satisfacción de reglas sobre atributos de distintas relaciones.

Dimensión: Unicidad

La unicidad indica el nivel de duplicación de los datos. La duplicación ocurre cuando un objeto del mundo real se encuentra presente más de una vez, esto es, varias tuplas representan exactamente el mismo objeto. Distinguimos entonces dos factores de la dimensión Unicidad:

- Duplicación: la misma entidad aparece repetida de manera exacta.
- Contradicción: la misma entidad aparece repetida con contradicciones.

3. Caso de Estudio: Experimento Controlado

La experimentación en ingeniería de software refiere a la correspondencia de las suposiciones, asunciones, especulaciones y creencias acerca del software con hechos de la realidad. Para establecer estas correspondencias la experimentación usa procedimientos metódicos para intentar validar, o rechazar, ciertas hipótesis.

Dentro de la experimentación se encuentran los experimentos controlados. Estos permiten comparar dos o más "conceptos". Por ejemplo, se puede realizar un experimento controlado en ingeniería de software para comparar dos o más técnicas de pruebas, dos o más procesos de software, técnicas de diseño, etc.

En estos experimentos se recolectan datos provenientes de diversas fuentes. Los datos luego son analizados con técnicas estadísticas para rechazar/aceptar las hipótesis planteadas. Por ejemplo, que cierta técnica de pruebas es más efectiva que cierta otra técnica.

En ingeniería de software gran parte de los datos que se recolectan durante un experimento es generada por humanos. Por ende, estos datos están propensos a contener errores y su calidad siempre debe estar en duda.

Durante 2008 y 2009 en la Facultad de Ingeniería, Universidad de la República en Uruguay realizamos un experimento controlado para evaluar la efectividad y costo de distintas técnicas de pruebas unitarias [8]. Este experimento lo utilizamos como caso de estudio para conocer la calidad de los datos en los experimentos controlados en ingeniería de software.

Los sujetos que participaron del experimento realizaron pruebas utilizando distintas técnicas sobre distintos programas. Todos son estudiantes avanzados de la carrera Ingeniería en Computación ya que se encontraban en cuarto o quinto año (último año) de la carrera.

Los sujetos debían probar los programas en busca de defectos. Un defecto es una anomalía en el código fuente. Cuando un sujeto encontraba un defecto debía registrar los datos del mismo.

Entro otros datos los defectos se clasificaron según dos taxonomías: ODC [3] y una propuesta por Beizer [2]. La taxonomía de ODC es una clasificación ortogonal de defectos, los defectos se clasifican según categorías que son ortogonales entre sí, mientras que Beizer es una taxonomía jerárquica. La clasificación de los defectos en las dos taxonomías se usó en el experimento para calcular la efectividad de cada técnica de pruebas respecto al tipo de defecto.

Los sujetos usan una Guía de Pruebas para realizar su trabajo en el experimento. Esta guía explica detalladamente cómo se deben diseñar y ejecutar las pruebas así como también cómo se debe realizar la recolección de datos (defectos encontrados, tiempo utilizado en las pruebas, etc.). La guía fue desarrollada por el equipo de investigación exclusivamente para este experimento. Los investigadores presentaron la guía a los sujetos durante una clase teórica previa al comienzo de la ejecución del experimento.

Cada experimento unitario (llamado Experiencia de Pruebas) consistió de un sujeto aplicando una técnica de prueba a un programa. El diseño del experimento distribuye a los 14 participantes en 40 experiencias de pruebas, cada una con un programa y una técnica de verificación a ejecutar.

De esta manera, cada sujeto aplicó distintas técnicas a distintos programas, y registró en la herramienta para registro de defectos y tiempos, los siguientes datos: fecha y hora de comienzo y finalización, tiempo de diseño de casos de prueba y de ejecución de la experiencia, y defectos encontrados.

Para cada uno de los defectos los sujetos debían registrar, entre otros, los siguientes datos: nombre de archivo y número de línea de código donde se encuentra el defecto, clasificación del defecto en ODC y Beizer, tiempo que le llevó detectar el defecto, descripción del defecto.

Para la recolección de estos datos se utilizó una herramienta disponible vía web llamada Grillo, la cual fue construida a medida para la recolección de datos del experimento. Sus principales funcionalidades son gestionar datos de las entidades: usuarios, técnicas de pruebas, experiencias y registro de defectos.

En Grillo se cargaron todos los experimentos unitarios. Dicha herramienta centraliza el registro de los defectos para todas las experiencias de verificación en una misma base de datos, y permite tener un control y seguimiento de las mismas.

Cada sujeto contaba con un usuario en la herramienta mediante el cual accedía a las experiencias que tenía asignadas, y registraba los datos requeridos. La herramienta es una aplicación web y la arquitectura está basada en un modelo cliente-servidor. Como sistema gestor de Base de Datos se utiliza HSQLDB.

En la Figura 2 se muestra el esquema de la base de datos de la herramienta Grillo, donde se almacenan los datos cuya calidad será analizada.

4. Definición de Tipos de Errores en los Datos

En esta sección se presentan los tipos de errores que se identificaron para el caso de estudio, para las dimensiones y factores de la calidad de datos. Un tipo de error define conceptualmente un error genérico en la calidad de los datos, para determinada dimensión y factor de calidad.

La Figura 3 muestra la relación que existe entre dimensiones, factores, tipos de errores y errores, y cómo se aplican estos conceptos en un caso particular. Mientras que la definición de dimensión y factor de calidad es general y resulta aplicable en cualquier contexto, los tipos de errores y errores son específicos y

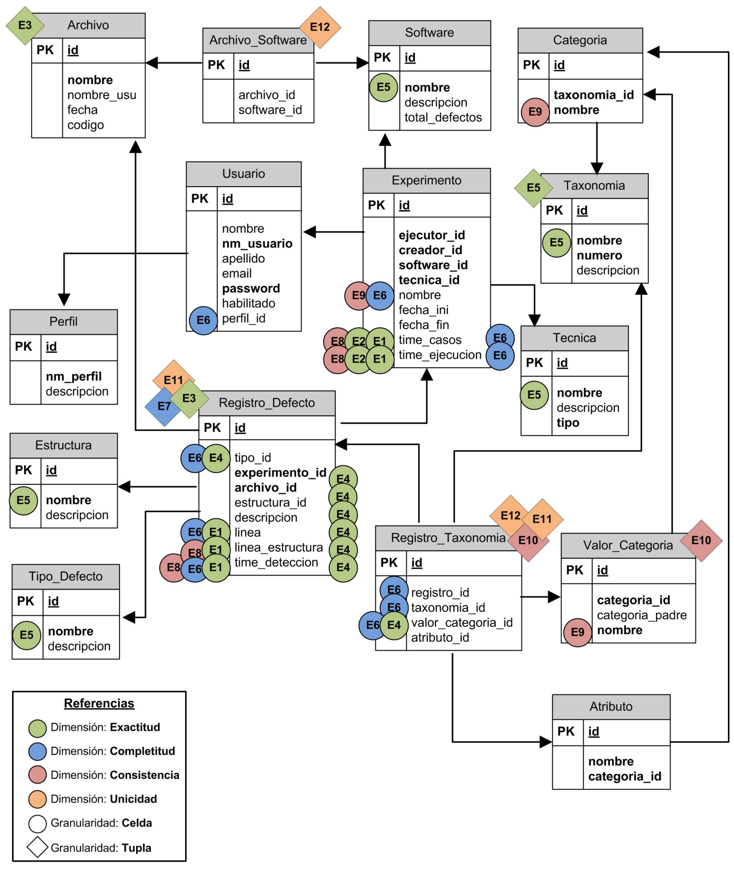


Figura 2. Esquema de la base de datos del experimento y tipos de errores

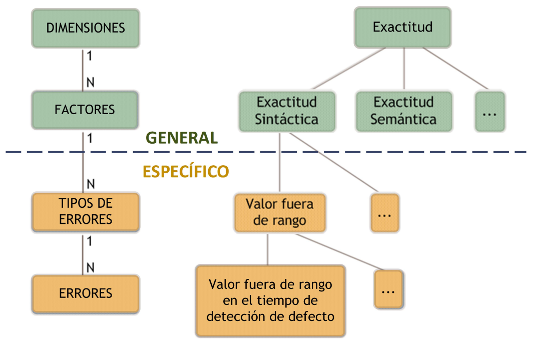


Figura 3. Relación entre Dimensiones, Factores, Tipos de Errores y Errores.

definidos para este caso de estudio, y en particular para Ingeniería de Software Empírica. Es por ello que los tipos de errores aquí presentados podrían ser reutilizados en otras experiencias empíricas.

Para identificar los tipos de errores realizamos una exploración de la herramienta Grillo y un análisis de la estructura de la base de datos que contiene los datos que registraron los sujetos.

A continuación se describen brevemente los tipos de errores identificados. Por motivos de espacio, solo presentamos un tipo de error para cada factor de calidad. Para todos los casos, la medida del error es booleana: se indica si el objeto medido contiene o no un dato erróneo. Los errores se miden, en la mayoría de los casos, mediante la definición y ejecución de consultas SQL.

Las dimensiones de calidad de datos que se miden son: Exactitud, Completitud, Consistencia y Unicidad. En la Tabla 1 se muestran, para cada dimensión y factor, los tipos de errores que identificamos para el caso de estudio.

Cuadro 1. Tipos de errores.

Dimensión	**Factor**	**Tipo de error**	**Identificador**
Exactitud	Exactitud sintáctica	Valor fuera de rango	E1
		Estandarización	E2
	Exactitud semántica	Registro inexistente	E3
		Defecto mal registrado	E4
		Valor fuera de referencial	E5
Completitud	Densidad	Valor nulo	E6
		Clasificación de defecto	E7
Consistencia	Integridad intra-relación	Reglas de integridad intra-relación	E8
		Valor único	E9
	Integridad referencial	Referencia inválida	E10
Unicidad	Duplicación	Registro duplicado	E11
	Contradicción	Registro contradictorio	E12

En la Figura 2 se presenta sobre qué objetos de la base se aplica cada tipo de error, indicado mediante su identificador. Las referencias de la figura indican a qué dimensión corresponde cada tipo de error así como su granularidad.

Valor fuera de rango

Los tiempos y líneas de código registradas deberían situarse dentro de un rango, previamente definido como válido. El valor de un tiempo que se encuentra fuera de ese rango podría ser un valor anómalo, y hacer variar incorrectamente los resultados y conclusiones obtenidas al analizar los datos del experimento (por ejemplo, cuánto se tardó en promedio en detectar un defecto de tipo X).

Se establecen criterios para determinar apropiadamente el rango a considerar para evaluar los valores, y se identifican los *outliers* mediante consultas en SQL. Es importante considerar, sin embargo, que si un valor cae fuera del rango determinado no significa necesariamente que dicho valor sea erróneo. El hecho de que existan valores anómalos pero correctos es parte de toda experiencia empírica.

El rango tiene su mínimo en 0 y a priori no se podría definir un valor máximo con certeza. Para ello se determina un intervalo considerando el valor medio y la desviación estándar de los tiempos registrados. Los valores fuera del rango [0, media + 2 x desviación estándar] se considerarán candidatos a contener errores, y por lo tanto serán analizados de manera aislada. Este rango se establece de forma arbitraria y su validez debe ser analizada en futuros trabajos. Sin embargo, sirve para tener una aproximación de cuáles son los datos con posibles problemas de calidad.

De la misma manera, un valor fuera de rango en las líneas de código no permitiría identificar, por ejemplo, cuáles son los registros de defectos que corresponden al mismo defecto de la realidad. Por otra parte, para el caso de las líneas de código se definirá un rango que tiene su mínimo en 1 y máximo igual a la cantidad de líneas que el archivo posee.

Registro inexistente

En este caso se identifican aquellos registros (tuplas) que se encuentran almacenados en la base de datos, pero que no se corresponden con ningún objeto de la realidad. Los registros inexistentes no deberían formar parte de la base ya que no reflejan la realidad.

Se analizan dos casos. Los registros de defectos, que corresponden a defectos detectados en el código y registrados en la herramienta, pero que no se identifican con ningún defecto real. Y los archivos, que corresponden al registro de archivos (.java) que no forman parte del experimento en cuestión.

Valor nulo

Interesa conocer qué información fue registrada y cuál fue omitida. Para aquella información que fue omitida, interesa conocer la causa de la omisión, y en caso que sea posible, determinar el valor que debería tomar en lugar de nulo.

Se identifican los campos que admiten nulos, pero deberían en la realidad contener algún valor distinto de vacío (el hecho de que admitan nulos es un error en el diseño de la base de datos).

El motivo de omisión de los campos podría ser que el Verificador no ingresa el valor (por omisión accidental o por no saber determinarlo), o por un error en el manejo de los datos (de la aplicación web o de la base) que ocasiona que el valor ingresado no se almacene correctamente.

Reglas de integridad intra-relación

Se definen un conjunto de reglas sobre los atributos que deben ser satisfechas en la base bajo estudio. El hecho de que alguna de estas reglas sea violada, afecta la consistencia de los datos y por lo tanto cualquier análisis que se lleve a cabo a partir de estos.

Referencia inválida

Es necesario considerar la satisfacción de reglas entre atributos de distintas tablas. Instanciando esto a los datos bajo estudio, se identifican referencias hacia determinadas tuplas que no existen en la base del experimento, y por lo tanto resultan ser referencias inválidas.

Registro duplicado

Se identifica este tipo de error cuando existen dos o más registros que aparecen repetidos de manera exacta. Existen dos situaciones:

- Cuando contienen el mismo valor en la clave y demás atributos (o en su defecto valores nulos). Este caso se contempla con controles del SGBD.
- A pesar de contener distinta clave primaria, hacen referencia al mismo objeto de la realidad y contienen los mismos datos en los campos que se definan. Para este caso se verifica que no existan registros repetidos (según el criterio definido) en la base bajo estudio.

Registro contradictorio

Se identifica este tipo de error cuando existen dos o más registros que aparecen repetidos de manera contradictoria. Esto significa que contienen distinto valor en la clave y/o demás atributos (o en su defecto valores nulos), a pesar de que hacen referencia al mismo objeto de la realidad.

El ejemplo más claro de este tipo de error es tener dos clasificaciones distintas dentro de una misma taxonomía, asociadas al mismo defecto. Es decir, un defecto debe estar clasificado de una única manera según una misma taxonomía.

5. Ejemplo de Error

A modo de ejemplo en esta sección se presenta cómo se realizó la medición de un error en particular. De esta forma se puede entender el trabajo realizado para cada uno de los errores.

Un error es la aplicación de un tipo de error (dentro de los identificados en la sección anterior) sobre un atributo, tupla y/o tabla específica, dependiendo de su granularidad. Este concepto se puede apreciar en la Figura 3. Una vez más nos referimos a un error en los datos respecto a la evaluación de su calidad.

Por otra parte, utilizamos el término instancia de error cuando a partir de la medición se detecta la presencia de algún dato erróneo sobre un objeto de la base (celda, tupla o tabla), de acuerdo a la granularidad definida.

La gran mayoría de los errores identificados fueron medidos, aunque existen algunas pocas excepciones correspondientes a mediciones manuales que no fueron ejecutadas.

Presentamos como ejemplo el error "Valor fuera de rango en el tiempo de detección de defectos".

El tipo de error Valor fuera de rango se mide sobre los tiempos y líneas de código. En particular, se presenta la medición realizada sobre el tiempo de detección de defectos.

Los sujetos deben ingresar en la herramienta Grillo el tiempo que les lleva detectar cada defecto en el código. Tanto el cálculo del tiempo como su ingreso en la herramienta se realizan de manera manual, lo que puede ocasionar que se introduzcan errores en los datos. Resulta interesante medir cuáles son los tiempos que se alejan de manera significativa del rango considerado como válido, y por lo tanto son candidatos a contener errores.

La forma de medir este error consiste en establecer un rango al que debe pertenecer el valor de cada tiempo, y verificarlo mediante la ejecución de consultas SQL.

Para obtener el rango dentro del cual deben situarse los valores de los tiempos de detección de defectos, se calcula el promedio y desviación estándar discriminando por tipo de defecto.

El rango de valores válidos para este registro es [0, promedio + 2 x desv. estándar]. El menor tiempo en el cual un sujeto puede detectar un defecto es en cero minutos; por esto es el valor mínimo del rango. El tiempo máximo del rango debe ser un valor que estadísticamente nos presente, en los registros que tengan valores por encima del mismo, registros con una alta probabilidad de contener un error. Es decir, los registros por encima del valor máximo son registros que probablemente estén mal. Este valor fue establecido en el promedio más dos veces la desviación estándar. Consideramos que cualquier registro que supere dicho valor es "sospechoso" y debe ser analizado.

Detectamos 21 registros (instancias de error) que contienen valores en el tiempo de detección de defectos fuera de dicho rango, de un total de 1009 registros de defectos. Vale destacar que se identificó la existencia de dos registros de defectos cuyo valor es casi seis veces mayor en comparación al máximo definido.

6. Resultados y Discusión

La aplicación de cada uno de los 12 tipos de errores identificados, sobre las tablas y atributos de la base, da como resultado un total de 55 errores. Para 38 de ellos, se ejecutó la medición de manera automática mediante sentencias SQL, y para un caso particular, se incluyeron algoritmos programados en Java.

La medición de 6 errores se realizó de manera manual por no ser posible su automatización. En todos los casos estas mediciones corresponden a verificaciones contra otras fuentes de datos no persistidas. Por ejemplo, consultas a responsables del experimento.

Los 11 errores restantes aún no han sido medidos. Para el tipo de error valor fuera de rango, no se cuenta con la cantidad de información suficiente para calcular promedios y desviaciones estándar que resulten significativos del total. Por otra parte, para medir la correctitud semántica sobre los registros de defectos es necesario identificar aquellos que no existen en el código, o que tienen valores en algún campo que no se corresponde con la realidad. Para ello se deben verificar manualmente los 1009 registros de defectos existentes, lo cual no se realizó aún por el esfuerzo que tiene asociada dicha verificación.

Los resultados de las mediciones indicaron que, de los 44 errores que fueron medidos, 18 contienen al menos una instancia de error (objeto de la base de datos para el cual se detecta la presencia de algún dato erróneo como resultado de las mediciones).

En total se identificaron un total de 7591 instancias con errores sobre las celdas y/o tuplas de la base. Una misma celda o tupla puede contener diferentes errores (de entre los 18 presentes), y en este caso se miden como instancias diferentes. Sin embargo, si consideramos la cantidad de tuplas de la base que contienen algún error (una tupla podría contener instancias de errores diferentes), obtenemos un total de 6906 tuplas con errores.

El Cuadro 2 muestra la cantidad de errores que fueron identificados por tipo de error

Cuadro 2. Cantidad de errores por tipo de error.

Tipo de Error	Valor fuera de rango	Estandarización	Registro inexistente	Defecto mal registrado	Valor fuera de referencial	Valor nulo
Cantidad de Errores	5	5	2	8	6	10
Tipo de Error	Clasificación de defectos	Regla de integridad intra-relación	Valor único	Referencia inválida	Registro duplicado	Registro contradictorio
Cantidad de Errores	2	5	3	3	3	3

En el Cuadro 3 se muestra cuáles son los 18 errores que contienen al menos una instancia de error, estableciendo a qué tipo de error corresponden, y sobre qué objeto de la base (tabla y atributo) se aplica. Además se indica, para cada error, el porcentaje de instancias con error de la cantidad de instancias totales que fueron medidas.

Encontramos distinta cantidad (en porcentaje) de instancias con error en los datos medidos. Los datos de clasificación de defectos en las taxonomías presentó un problema para los sujetos del experimento. Esto se desprende de que existe un alto porcentaje de instancias con error en los objetos de la base que están vinculados a los datos de clasificación de defectos.

El Cuadro 4 muestra el porcentaje de tuplas erróneas para las tablas sobre las cuales se identifica la existencia de al menos una instancia de error. Una tupla errónea puede contener más de una instancia de error diferente.

Ya mencionamos que la clasificación de defectos realizada por los sujetos contiene muchas instancias con errores. En el Cuadro 4 también se puede ver que otras tablas también tiene una cantidad importante de instancias con error; varias de ellas alrededor del 10 %. Este porcentaje indica que es neceario una limpiza de los datos para garantizar que el análisis estadístico de los datos del experimento sea sobre datos válidos. Caso contrario los resultados, y las conclusiones, del experimento pueden incluso no corresponderse con la realidad (y este es justamente el objetivo de los experimentos controlados, estudiar y conocer la realidad).

Cuadro 3. Errores con al menos una instancia errónea.

Tipo de error	**Objeto**	**% de instancias con error**
Valor fuera de rango	Registro_Defecto.time_deteccion	2,1 %
	Registro_Defecto.linea	0,8 %
	Registro_Defecto.linea_estructura	0,3 %
Registro inexistente	Archivo	2,1 %
Valor fuera de referencial	Registro_Taxonomia	38,4 %
Valor nulo	Experimento.time_ejecucion	6,8 %
	Registro_Taxonomia.taxonomia_id	13,7 %
	Registro_Taxonomia.valor_categoria_id	0,02 %
	Experimento.time_casos (técnica dinámica)	6,8 %
Clasificación de defectos	Registro_Defecto (para ODC)	0,3 %
Regla de integridad intra-relación	Registro_Defecto.time_deteccion	8,9 %
	Experimento.time_ejecucion	4,5 %
Referencia inválida	Registro_Taxonomia.registro_id	4,9 %
	Valor_Categoria.categoria_padre	10 %
Registro duplicado	Registro_Taxonomia (para ODC)	17,6 %
Registro contradictorio	Archivo_Software	8,3 %
	Registro_Taxonomia (para ODC)	1,5 %
	Registro_Taxonomia (para Beizer)	0,7 %

Cuadro 4. Tuplas con error por tabla.

Tabla	**% de tuplas con error por tabla**
Archivo	2 %
Archivo_Software	8 %
Experimento	11 %
Valor_Categoria	10 %
Registro_Defecto	12 %
Registro_Taxonomia	52 %

7. Conclusiones

En este trabajo presentamos una investigación de la calidad de los datos de un experimento controlado en ingeniería de software. Presentamos tipos de errores y errores asociados a un caso de estudio y los resultados de medirlos.

Detectamos 55 errores diferentes de los cuales pudimos medir 44. 38 fueron medidos de forma automática y 6 de forma manual. Los restantes 11 no se pudieron medir por diferentes motivos.

De los 44 errores medidos encontramos que 18 tienen al menos una instancia de error. Esto quiere decir que existen datos que consideramos erróneos en la base de datos para esos 18 errores.

El porcentaje de instancias con error sobre instancias totales medidas indica que los datos deberían limpiarse antes de realizar análisis estadísticos del experimento. Es decir, realizar análisis estadísticos sin la limpieza de estos datos puede provocar resultados y conclusiones inválidas del experimento.

Uno de los aportes de este trabajo es haber realizado un estudio novedoso: estudiar la calidad de datos de un experimento de ingeniería de software. Sobre todo, este trabajo muestra que los investigadores que realizan experimentos controlados deben tener en cuanta la calidad de los datos de sus experimentos antes de realizar los análisis estadísticos.

Desde la perspectiva del área Calidad de Datos, este trabajo muestra una aplicación de las técnicas de medición de calidad y de limpieza de datos a un dominio particular, donde fue necesario seleccionar las dimensiones de calidad adecuadas y definir las métricas y procedimientos correspondientes para obtener los resultados esperados.

Referencias

1. Batini, C., Scannapieco, M.: Data Quality: Concepts, Methodologies and Techniques. Springer-Verlag Berlin Heidelberg (2006)
2. Beizer, B.: Software Testing Techniques, Second Edition. Van Nostrand Reinhold Co. (1990)
3. Chillarege, R.: Handbook of Software Reliability Engineering. IEEE Computer Society Press, McGraw-Hill Book Company (1996)
4. Etcheverry, L., Peralta, V., Bouzeghoub, M.: Qbox-foundation: a metadata platform for quality measurement. In: 4th Data and Knowledge Quality Workshop (2008)
5. Lee, Y.W., Strong, D.M., Kahn, B.K., Wang, R.Y.: Aimq: a methodology for information quality assessment. Inf. Manage. 40, 133–146 (2002)
6. Neely, M.P.: The product approach to data quality and fitness for use: A framework for analysis. In: Proceedings of the 10th International Conference on Information Quality MIT (2005)
7. Strong, D.M., Lee, Y.W., Wang, R.Y.: Data quality in context. Commun. ACM 40, 103–110 (1997)
8. Vallespir, D., Apa, C., De León, S., Robaina, R., Herbert, J.: Effectiveness of five verification techniques. In: Proceedings of the XXVIII International Conference of the Chilean Computer Society (2009)
9. Wang, R.Y., Reddy, M.P., Kon, H.B.: Toward quality data: an attribute-based approach. Decis. Support Syst. 13, 349–372 (1995)

Chapter 8

Conclusions

The works presented in this part complement the central investigation of this thesis. The work of analysis of taxonomies presents a study and comparison of different defects taxonomies. The results of this work are interesting in themselves beyond the central research topic of this thesis. As far as the research line is concerned, we have planned to conduct experiments that study the effectiveness of the techniques segmented by type of defect both in a generic frame as in the frame of the PSP. The study of taxonomies will help determine how and with which taxonomy to classify the defects of future experiments.

The framework of comparison of experiments that evaluates testing techniques needs to be improved and extended. One of the most important conclusions of this work is that the researchers present in very different forms the experiments conducted and take into consideration different aspects of the same. This makes experimental replication very difficult. The article that presents the packaging of the Experiment 2008, makes it clear that a normalized packaging helps replication.

The last article of this Part presents the importance of data quality in software engineering experiments. Namely, it shows the importance of conducting a process of data cleansing before performing statistical analyses.

Part III

Analysis of Defect Injection and Removal in PSP

Chapter 9

Introduction

Controlled experiments that study the effectiveness and the cost of different testing techniques were presented in Part I. Those experiments study the techniques without controlling for the process of which they are part. To evaluate the influence of the overall development process on the techniques studied we need first to establish a baseline against which to compare.

In this Part the effectiveness of the PSP phases and the cost to find and fix defects having the PSP as a context are analyzed. Further research is described in the Future Work part. We used data from PSP courses taught between October 2005 and January 2010. These courses were taught by the Software Engineering Institute (SEI) of the Carnegie Mellon University or by SEI partners.

The Personal Software Process (PSP) is a software development process for an individual. This process guides the engineer from the planning of a small software program to its unit testing. It is made up of different phases: planning, detailed design, detailed design review, code, code review, compile, unit testing and postmortem. During the process the engineer records, among other things, the time employed in each phase as well as the data concerning the defects he finds and fix. The second article of this Part presents a description of the PSP in section 2.

The PSP is taught to the engineers through a theoretical and practical course. In it the engineers develop different software programs using the process. These courses are an excellent source of data of the use of the PSP.

Design reviews and code reviews are conducted using a checklist therefore the code review is similar to the desktop inspection technique presented in Part I. The checklists are created by the engineers themselves so that the items refer to the defects they normally inject during the development.

This Part is made up by two articles. The first one analyzes the injection and removal of defects injected during the detailed design phase. The second one makes a similar analysis but with the defects injected during the code phase.

Chapter 10

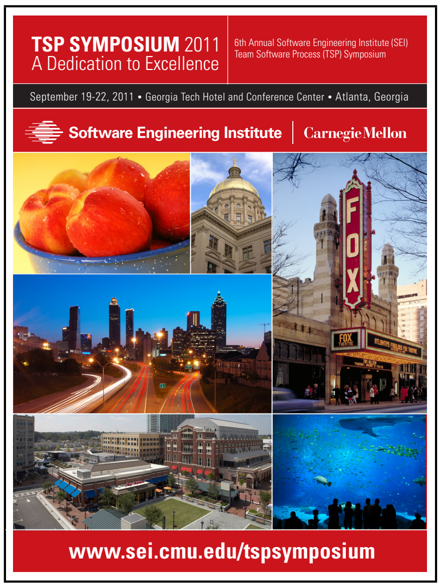
Analysis of Defect Injection and Removal in PSP

The papers included in this chapter are:

Analysis of Design Defects Injection and Removal in PSP
Diego Vallespir and William Nichols
Proceedings of the TSP Symposium 2011: A Dedication to Excellence, pp. 19-24, Atlanta, GA, United States, September, 2011

Analysis of Code Defects Injection and Removal in PSP
Diego Vallespir and William Nichols
Submitted to a Special Issue of a Software Engineering Open Journal.

ARTICLE



TSP SYMPOSIUM 2011
A Dedication to Excellence

6th Annual Software Engineering Institute (SEI) Team Software Process (TSP) Symposium

September 19-22, 2011 • Georgia Tech Hotel and Conference Center • Atlanta, Georgia

Software Engineering Institute | Carnegie Mellon

www.sei.cmu.edu/tspsymposium

Thank You

Thanks to the TSP Symposium 2011 Program Committee
Timothy Chick
Software Engineering Institute, TSP Symposium 2011 Technical Chair
Senior Member of Technical Staff, SEI Software Engineering Process Management Program

Lana Cagle, U.S. Navy
Barry Dwolatzky, Johannesburg Centre for Software Engineering
João Pascoal Faria, Faculdade de Engenharia da Universidade do Porto
Bill Nichols, Software Engineering Institute
David Saint-Amand, U.S. Navy
Rafael Salazar, Tec de Monterrey
Kathy Smith, HP Enterprise Services
David Webb, U.S. Air Force

Thanks to the TSP Symposium 2011 Planning Committee
Bob Rosenstein, SEI Conferences, Events, and Trade Shows Manager
Michele Falce, SEPM Technical Event Coordinator
Ruth Gregg, TSP Symposium Lead Event Planner
Brittney Osikowicz, Public Relations Coordinator

Thanks to the SEI Partner Network Staff
Lisa Masciantonio, SEI Partner Network Manager
Tracey Kelly
Kay Vinay

Thanks to the SEI Communication Design, Web Communication, and Customer Relations Staff
Karen Balistreri
Jeff Balmert
Deen Blash
Jeff Federoff
Maureen Fechik
David Gregg
Shane McGraw
Bill McSteen
Anna Mosesso
Melissa Neely
Daniel Pipitone
Trish Schreiber
Lizann Stelmach
Cat Zaccardi

Thank you to the Atlanta SPIN for its warm welcome and hospitality.

Be sure to bookmark www.sei.cmu.edu/tspsymposium to get this year's proceedings and for information on future TSP events hosted by the SEI.

Upcoming SEI Conferences and Events

Visit www.sei.cmu.edu/events/ to get the latest information on upcoming conferences and events.

October 24-26, 2011
CMMI Workshop 2011
Minneapolis, Minnesota

November 2-3, 2011
SEPG Latin America 2011
Lima, Peru

March 12-15, 2012
SEPG North America 2012
Reaching New Levels of Excellence
Albuquerque, New Mexico

June 5-7, 2012
SEPG Europe 2012
¡A Passion for Process!
Madrid, Spain

August 2012
SEPG Asia-Pacific 2012
Melbourne, Australia

Analysis of Design Defect Injection and Removal in PSP

Diego Vallespir and William R. Nichols

Proceedings of the TSP Symposium 2011: A Dedication to Excellence, pp. 19-24, Atlanta, GA, United States, September, 2011.

Analysis of Design Defect Injection and Removal in PSP

Diego Vallespir, Universidad de la Republic
William Nichols, Software Engineering Institute

1.1 INTRODUCTION

A primary goal of software process improvement is to make software development more effective and efficient. One way of doing that is to understand the role of defects in the process and make informed decisions about avoiding defect creation or committing the effort necessary to find and fix the defects that escape development phases. Through examination of a large amount of data generated during Personal Software Process (PSP) classes, we can show how many defects are injected during design, what types of defects are injected, and how they are detected and removed in later development phases. We can use this information to teach developers how to define and improve their own processes, and thus make the product development more effective and efficient.

"The Personal Software Process (PSP) is a self-improvement process that helps you to control, manage, and improve the way you work" [Humphrey 05]. This process includes phases that you complete while building the software: plan, detailed design, detailed design review, code, code review, compile, unit test, and post mortem. For each phase, the engineer collects data on the time spent in the development phase and data about the defects injected and removed. The defect data include the defect type, the time to find and fix the defect, the phase in which the defect was injected, and the phase in which it was removed.

During the Personal Software Process course, the engineers build programs while progressively learning PSP planning, development, and process assessment practices. For the first exercise, the engineer starts with a simple, defined process (the baseline process, called PSP0); as the class progresses, new process steps and elements are added, from estimation and planning to code reviews, to design, and design review.

In this article, we present an analysis of defects injected during the design phase of the PSP programs 6, 7, and 8 (all developed using PSP2.1). In PSP2.1, students conceptualize program design prior to coding and record the design decisions using functional, logical, operational, and state templates. Students then perform a checklist-based personal review of the design to identify and remove design defects before beginning to write code.

In this analysis, we focused on defects injected during the design phase because these data had not been specifically studied before. Previous studies did not have all the defect data, such as defect types and individual defect times; they had only summaries. Our analysis of the complete data available from individual defect logs shows not only that the defects injected during design are the most expensive to remove in test but also these are easy to remove in the review phases. The difference is striking: it costs five times more to remove a defect in test than it does to remove that same defect during review.

To show this, we observed how defects injected during design escaped into each subsequent phase of the PSP and how the cost to remove was affected by defect type and phase. We describe the different defects types injected during design and how these defect types compare with respect to the "find and fix" time. From this analysis, we show that "function" defects are the most common design phase defect, that personal design review is an effective removal activity, and that finding and fixing design defects in review is substantially less expensive than removal in test.

Some other articles study software quality improvement using PSP [Paulk 10] [Wholin 98] [Rombach 08] [Paulk 06] [Hayes 97] [Ferguson 97]. In the context of PSP, quality is measured as defect density (defects/KLOC). Our study differs from these others in that we focus on design defects, consider the defect type, and do not consider defect density. Instead, we focus on the characteristics of the defects introduced in design. Our findings resulted from analyses of the defect types injected, how they proceeded through the process until they were found and removed, and cost of removal in subsequent development phases.

1.2 THE DATA SET

We used data from the eight program version of PSP for Engineers I and II taught between October 2005 and January 2010. These courses were taught by the Software Engineering Institute (SEI) at Carnegie Mellon University or by SEI partners, including a number of different instructors in multiple countries.

This study is limited to only consider the final three programs of the 2006 version of the PSP course (programs 6, 7, and 8). In these programs, the students apply the complete PSP process, using all process elements and techniques. Specifically, these exercises include the use of design templates and design reviews. Of course, not all of these techniques are necessarily applied well because the students are in a learning process.

We began with the 133 students who completed all programming exercises. From this we made several cuts to remove errors and questionable data and to select the data most likely to have comparable design and coding characteristics.

Because of data errors, we removed data from three students. Johnson and Disney reviewed the quality of the PSP data [Johnson 1999]. Their analysis showed that 5% of the data was incorrect; however, many or most of those errors in their data were due to calculations the students made. Because our data were collected with direct entry into a MS Access tool, which then performed all calculations automatically, the lower amount (2.3%) of data removed is lower than the percentage reported by Johnson and Disney but seems reasonable.

We next reduced the data set to separate programming languages with more common design and coding characteristics. As we analyze the design defects, it seems reasonable to consider only languages with similar characteristics that might affect code size, modularity, subroutine interfacing, and module logic. The students used a number of different program languages, as shown in Figure 1.

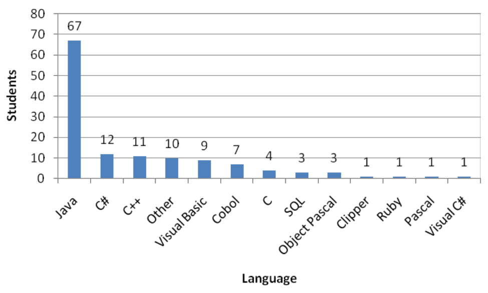


Figure 1: Quantity of students by program languages

The most common language used was Java. To increase the data set size, we decided to include the data generated by students who used Java, C#, C++, and C. This group of languages uses similar syntax, subprogram, and data constructs. For the simple programs produced in the PSP course, we judged that these were most likely to have similar modularization, interface, and data design considerations. This cut reduced our data to that generated by the programming efforts of 94 students.

Because our intent was to analyze defects injected in the design phase, we removed from consideration any data for which design defects were not recorded. From the 94 data sets remaining, two recorded no defects and 11 recorded no defects injected during design. Our data set for this analysis was, therefore, reduced to 92 engineers or 83 engineers depending upon the specific analysis performed. In the following sections we present the types of defects that were injected in the design phase, when the defects were removed, and the effort required to find and fix these defects.

1.3 WHERE THE DEFECTS ARE INJECTED

The first goal of our analysis was to better understand where defects were injected. We expected injections to be dominated by the design and code phases of course, because they are the construction phases in PSP. We began by explicitly documenting the phase injection percentages.

This analysis studied the defect injection and removal performance of individuals and the performance variation among them. We included the 92 engineers who recorded defects. We began by computing the phase data for each individual and then computed the following statistics of the distribution of individuals:

- an estimate of the mean percentage of defects injected by phase
- the 95% confidence interval for that mean (to characterize the standard error on the mean)
- the standard deviation of the distribution to characterize the spread among individuals

For each phase and for each individual, we calculated the percentage of defects injected in each PSP phase. The distribution statistics are shown in Table 1.

	DLD	DLDR	Code	CR	Comp	UT
Mean	46.4	0.4	52.4	0.3	0.03	0.5
Lower	40.8	0.2	46.7	0.0	0.0	0.2
Upper	52.0	0.7	58.1	0.7	0.09	0.9
Std. dev.	27.2	1.7	27.4	1.8	0.3	1.8

Table 1: Mean lower, upper confidence interval values and std. dev. of the % of defects injected by phase

The design and code phases have similar injection percentages both on average and in the spread. Their mean of the percentage of defects injected is near 50% with lower and upper confidence interval bounds between 40% and 58%. Both standard deviations are around 27% with. So, in the average of this population, roughly half of the defects were injected in the design phase and the other half of the defects were injected in the code phase. On average, the defect potential of these phases appears to be very similar. The standard deviation shows, however, that the variability between individuals is substantial. Nonetheless, as we expected, in the average almost 99% of the defects were injected in the design and code phases with only around 1% of the defects injected in the other phases.

The design review, code review, compile, and unit test phases also have similar average defect potentials. The average in all these cases is less than 0.5% and their standard deviations are small, the largest being 1.8% in code review and unit testing. This shows that during verification activities in PSP the percentage of defects injected is low but not zero. From time to time, developers inject defects while correcting other defects. We will study these secondary injections in a later study.

The variability between individuals and the similarity between the code and design phase is also presented in Figure 2. Note that the range in both phases is from 0% to 100% (all possible values). The 25th percentile is 26.34 for design and 35.78 for code, the median is 45.80 for design and 52.08 for code, and the 75th percentile is 64.22 for design and 71.56 for code.

Despite a high variability between individuals, this analysis shows that the great majority of defects are injected in the design and code phases. Slightly more defects are injected during code than during design, but the difference is not statistically significant. We could, therefore, focus on the defects injected in the design and code phases. In this article, we discuss only the defects injected in the design phase.

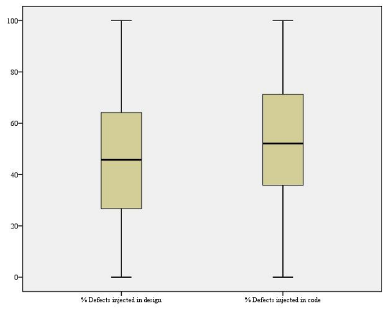


Figure 2: Percentage of defects injected by phase (box and whisker chart)

1.4 ANALYSIS OF DESIGN DEFECTS

From the 94 engineers in our data set there were 11 who recorded no injected defects during design. Our data set for analysis of the design defects was, therefore, reduced to 83 engineers. In the following sections we discuss the types of defects that are injected in the design phase, when those defects are removed, and the effort required to find and fix the defects.

1.4.1 Defect types Injected during Design

To improve the detection of design defects we first wanted to know which types of defects were injected during the design phase. Table 2 shows the mean of the percentage of the different defect types injected. It also presents the lower and upper bound of the 95% confidence interval for the mean (a measure of the standard error) and the standard deviation of the distribution.

We divided these defect types into three categories. The first is "almost not defects of this type." In this category we found system and build/package defects. It is clear that during the PSP course these types of defects were almost never injected. This may be due to the PSP course exercises rather than the PSP. Because the exercises are small, taking only a few hours, and contain few components, and make few external library references, build packages are usually quite simple. We expect to find more defects of these types in the Team Software Process in industrial scale projects. A second category is "few defects;" most of the other defect types (all except Function type) are in this category. The percentage of defects in this category ranged from 3.1% to 12.6%.

The last category, "many defects," includes only one type of defect: function. The great majority of defects injected during design were of the Function type. This type of defect was almost half of all the defects injected during Design. This is an observation familiar to PSP instructors, but not previously reported for a sizable data set.

The lower, upper, and standard deviation data show again the high variability between individuals. This can also be observed in Figure 3; the box and whisker chart shows many observations as outliers. Also, it can be seen that the function defect type goes from 0% to 100% and that the 25 percentile is 21% and the 75 percentile is 70%.

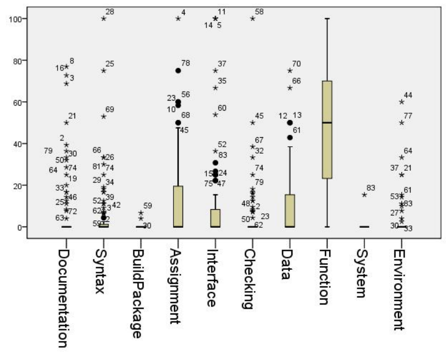


Figure 3: Box and whisker of the percentage of defects injected during design

1.4.2 When Are the Defects Injected During Design Removed?

Our analysis indicated the subsequent phases during which the design defects were removed. While our data set was larger than any previously studied, it remained too small for us to examine the removals based on defect type. Still, for each engineer who injected design defects, we identified the phases in which the engineers found the defects, then, for every phase, we determined the percentage of the defects that were found in that phase.

	Docs.	Syn.	Build	Assign.	Inter.	Check	Data	Func.	Syst.	Env.
Mean	6.9	6.0	0.1	12.6	10.0	4.6	9.8	46.6	0.2	3.1
Lower	3.3	2.5	0.0	8.2	5.1	1.6	6.3	39.7	0.0	0.9
Upper	10.5	9.5	0.3	17.0	15.0	7.6	13.3	53.5	0.6	5.3
Std. dev.	16.6	16.0	0.8	20.2	22.5	13.8	16.0	31.5	1.7	10.1

Table 2: Percentage of defect types injected during design

Table 3 shows the mean (with 95% confidence interval) and standard deviation for the different phases. The 95% confidence interval is bounded by 45.8% and 61.0%. As previously shown, the standard deviation was high, indicating the high variability between individuals. From this we learned that approximately 50% of the defects injected during Design were found in the detailed level design review (DLDR) phase.

	DLDR	Code	CR	Comp	UT
Mean	53.4	9.6	8.9	2.5	25.7
Lower	45.8	5.7	5.2	0.0	19.3
Upper	61.0	13.4	12.5	5.2	32.0
Std. dev.	34.8	17.5	16.7	12.3	29.2

Table 3: Phases where are founded the design defects (percentage)

Figure 4 shows the box and whisker charts displaying the percentage of defects found in the different phases and the histogram for the defects founds in DLDR. Figure 4 also shows the high variability between individuals in the percentage of defects found during DLDR and UT.

Code and code review have a similar percentage of defects that were injected during design. Approximately 10% of the defects were found and removed in each of those phases. Approximately 2.5% of the design defects were found during the compile phase; it is likely that the defects found in compile were pseudo-code defects. And finally, around 25% of the defects were found in unit test (UT). This means that in the PSP accounting, one of every four defects injected during design escapes all phases prior to UT. We know, of course, that not all the defects that escape into UT are found in UT. UT will not have a 100% yield; therefore the percentage of defects found in each of these phases is smaller than reported while the actual percentage of escapes into UT is a lower limit. An estimate or measurement of the UT yield will be necessary to revise these phase estimates.

1.4.3 Cost to Remove the Defects Injected in Design

What is the cost and variation in cost (in minutes) to find and fix the defects that are injected during design? First, we analyze the differences in cost segmented by the removal phase. Second, we study the differences in cost segmented by defect type.

It would also be interesting to segment and analyze both the removal phase and the defect type jointly. Unfortunately, because of limited sample size after a two dimensional segmentation, we cannot perform that analysis with statistical significance.

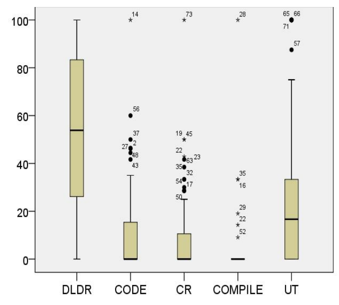
1.4.3.1 Phase Removal Cost

What is the cost, in each removal phase, to find and fix a defect injected in design? Design defects can be removed in the detailed level design review (DLDR), code, code review (CR), compile, and unit test (UT) phases. For each engineer, we calculated the average task time of removing a design defect in each of the different phases. Because some engineers did not remove design defects in one or more phases, our sample size varied by phase. We had data from 67 engineers for DLDR, 29 each for code and CR, six for compile, and 55 for UT. We excluded the cost of finding design defects in the Comp phase because we had insufficient data for that phase.

Table 4 shows the mean, lower and upper 95% confidence interval, and the standard deviation for the find and fix time (in minutes) for design defects in each of the studied phases. We might have expected an increased cost in each phase but this is not what the data showed.

	DLDR	CODE	CR	UT
Mean	5.3	5.1	4.2	23.0
Lower	3.7	2.5	2.6	11.6
Upper	6.9	7.6	5.7	34.3
Std. dev.	6.6	6.7	4.1	42.0

Table 4: Cost of find and fix defects injected in design segmented by phase removed



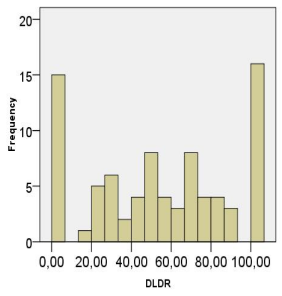


Figure 4: In which phase are the design defects found? – Variability between individuals

Rather, **the "find and fix" cost remained almost constant during DLDR, code, and CR**; in fact, the cost decreased a little in these phases, though the differences were not statistically significant. Further analysis will be needed to determine which of the defect finds in code and code review escaped through design and an effective (as opposed to ineffective) design review. Regardless, any defect discovered in these phases is essentially found by an inspection process where the "fix" time is short because the root cause has been identified.

We are not stating here that the cost of finding and fixing a design defect during DLDR, code, and CR is necessarily the same. We are stating that using PSP, the design defects that are removed during DLDR cost approximately the same as removing the ones that escape from design into code and those that escape from design into CR.

As we expected**, the average cost of finding a design defect during UT is much higher than in the other phases** by almost a factor of 5.

We also found a high variability among individual engineers. This variability can be seen in the box and whisker chart in Figure 5. We tested for normal distribution after log transformation of find and fix times for DLDR, code, CR, and UT and all are consistent (p> 0.05 using a Kolmorov-Smirnov and Shapiro-Wilk test) with a log-normal distribution. This test is primarily useful to verify that we can apply regression to the transformed data; however, understanding the distribution also helped to characterize the asymmetry and long tailed nature of the variation. That is, the log-normality affirmed our intuition that some defects found in test required far more than the average effort to fix, making test time highly variable. We also observed that the both the mean and variation of rework cost in the DLDR, code, and CR phases were significantly lower than UT in both the statistical sense and the practical sense.

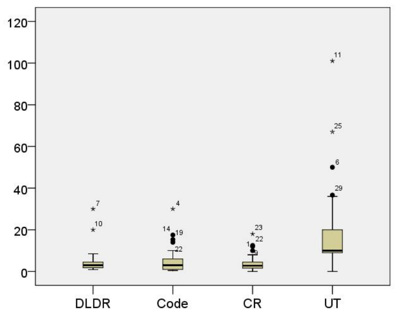


Figure 5: Box and whisker of the cost of find and fix a design defect segmented by phase removed

1.4.3.2 Defect Removal by Type

What is the find and fix cost, per defect type, of defects injected during detailed design? As we mentioned before, we had few build/package and system defects injected during design. As a result, we didn't have enough data for a statistically significant analysis of the cost of removing these two types of defects. However, we were able to analyze the remaining defect types.

Table 5 presents the mean, lower, and upper 95% confidence interval and the standard deviation for the find and fix cost of design defects, segmented by type. The cost, in minutes, for "find and fix" fell into three groups:

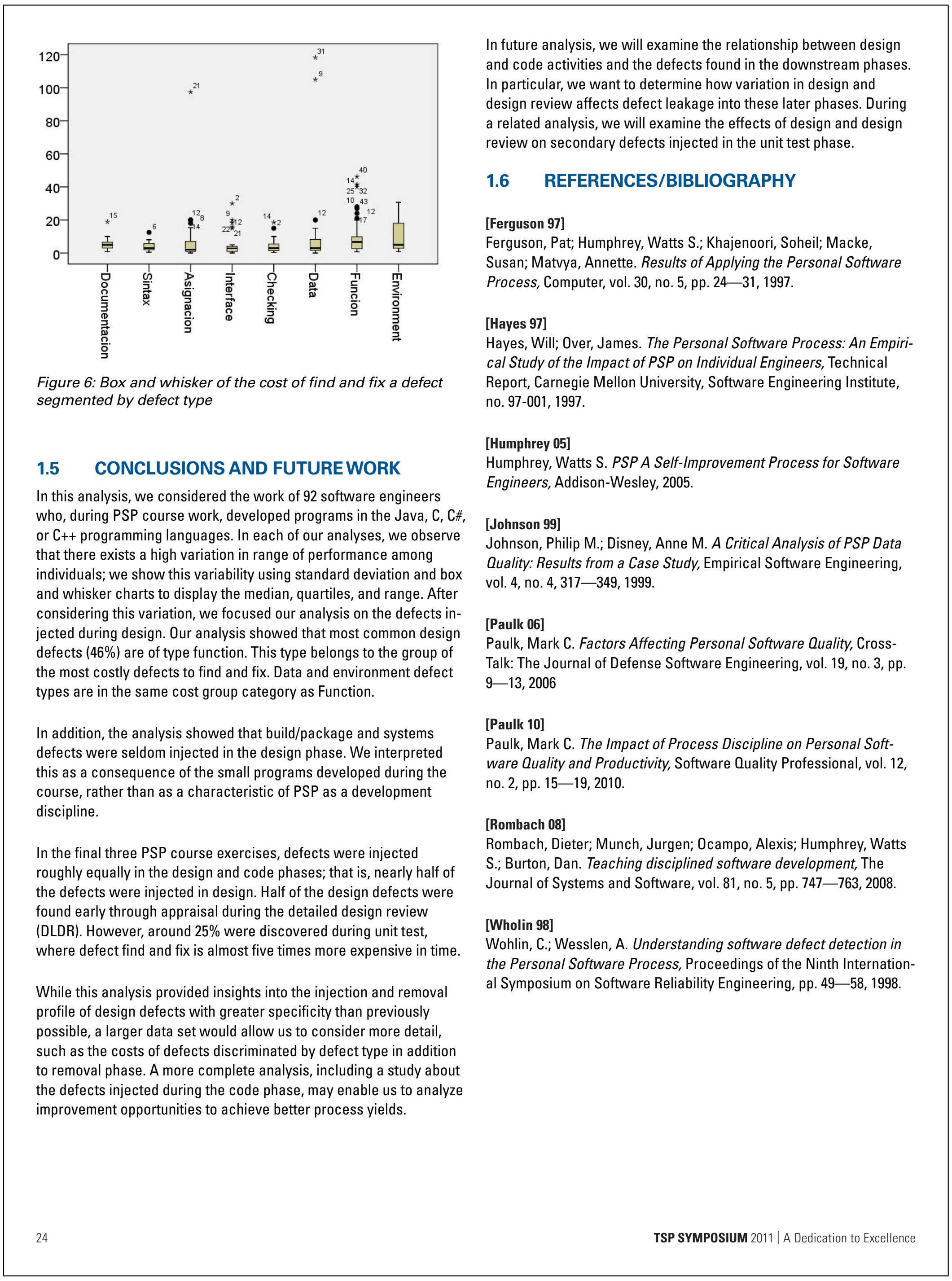
- a group that has a mean near 5 minutes: Documentation, Syntax, Interface, Checking
- a group, composed only of Assignment defects, that has a mean near 7 minutes
- a group that has a mean near 10 minutes: Data, Functions, Environment

The confidence interval of the second group is sufficiently wide that it is not clearly distinct from either of the other two groups, falling more or less in between. More data would help to clarify this grouping.
We want to emphasize that the third group, including data, functions, and environment, takes twice the time to find and fix the defects than the documentation, syntax, interface and checking types of the first group.

	Docs.	Syn.	Assign.	Inter.	Check	Data	Func.	Env.
Mean	5.6	4.3	7.3	5.4	4.9	11.0	9.3	10.5
Lower	3.6	1.8	1.9	2.5	2.2	2.2	6.9	3.0
Upper	7.6	6.7	12.7	8.2	7.5	19.8	11.7	17.9
Std. dev.	4.1	3.7	16.3	7.3	5.2	25.6	10.1	11.7

Table 5: Cost of find and fix defects injected in design discriminated by defect type

As in the other cases, the variation among individual developers was high. This can be seen using the standard deviation, as well as the box and whisker chart that is presented in Figure 6.



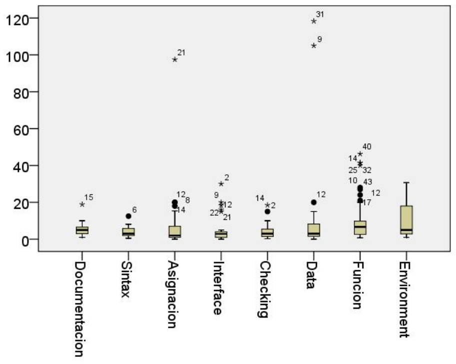


Figure 6: Box and whisker of the cost of find and fix a defect segmented by defect type

1.5 CONCLUSIONS AND FUTURE WORK

In this analysis, we considered the work of 92 software engineers who, during PSP course work, developed programs in the Java, C, C#, or C++ programming languages. In each of our analyses, we observe that there exists a high variation in range of performance among individuals; we show this variability using standard deviation and box and whisker charts to display the median, quartiles, and range. After considering this variation, we focused our analysis on the defects injected during design. Our analysis showed that most common design defects (46%) are of type function. This type belongs to the group of the most costly defects to find and fix. Data and environment defect types are in the same cost group category as Function.

In addition, the analysis showed that build/package and systems defects were seldom injected in the design phase. We interpreted this as a consequence of the small programs developed during the course, rather than as a characteristic of PSP as a development discipline.

In the final three PSP course exercises, defects were injected roughly equally in the design and code phases; that is, nearly half of the defects were injected in design. Half of the design defects were found early through appraisal during the detailed design review (DLDR). However, around 25% were discovered during unit test, where defect find and fix is almost five times more expensive in time.

While this analysis provided insights into the injection and removal profile of design defects with greater specificity than previously possible, a larger data set would allow us to consider more detail, such as the costs of defects discriminated by defect type in addition to removal phase. A more complete analysis, including a study about the defects injected during the code phase, may enable us to analyze improvement opportunities to achieve better process yields.

In future analysis, we will examine the relationship between design and code activities and the defects found in the downstream phases. In particular, we want to determine how variation in design and design review affects defect leakage into these later phases. During a related analysis, we will examine the effects of design and design review on secondary defects injected in the unit test phase.

1.6 REFERENCES/BIBLIOGRAPHY

[Ferguson 97]
Ferguson, Pat; Humphrey, Watts S.; Khajenoori, Soheil; Macke, Susan; Matvya, Annette. *Results of Applying the Personal Software Process,* Computer, vol. 30, no. 5, pp. 24—31, 1997.

[Hayes 97]
Hayes, Will; Over, James. *The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers,* Technical Report, Carnegie Mellon University, Software Engineering Institute, no. 97-001, 1997.

[Humphrey 05]
Humphrey, Watts S. *PSP A Self-Improvement Process for Software Engineers,* Addison-Wesley, 2005.

[Johnson 99]
Johnson, Philip M.; Disney, Anne M. *A Critical Analysis of PSP Data Quality: Results from a Case Study,* Empirical Software Engineering, vol. 4, no. 4, 317—349, 1999.

[Paulk 06]
Paulk, Mark C. *Factors Affecting Personal Software Quality,* Cross-Talk: The Journal of Defense Software Engineering, vol. 19, no. 3, pp. 9—13, 2006

[Paulk 10]
Paulk, Mark C. *The Impact of Process Discipline on Personal Software Quality and Productivity,* Software Quality Professional, vol. 12, no. 2, pp. 15—19, 2010.

[Rombach 08]
Rombach, Dieter; Munch, Jurgen; Ocampo, Alexis; Humphrey, Watts S.; Burton, Dan. *Teaching disciplined software development,* The Journal of Systems and Software, vol. 81, no. 5, pp. 747—763, 2008.

[Wholin 98]
Wohlin, C.; Wesslen, A. *Understanding software defect detection in the Personal Software Process,* Proceedings of the Ninth International Symposium on Software Reliability Engineering, pp. 49—58, 1998.

ARTICLE

Analysis of Code Defect Injection and Removal in PSP

Diego Vallespir and William Nichols

Submitted to a Special Issue of a Software Engineering Open Journal. The format is not the Journal one.

SUBMITTED TO AN OPEN JOURNAL IN SOFT. ENG., VOL. XX, NO. XX, 2012

Analysis of Code Defect Injection and Removal in PSP

Diego Vallespir, William Richard Nichols

Abstract—Defects are the primary source of rework during software development. Late rework often consumes 40 percent or more of project duration and resources. By examining all defects injected and removed during development projects, we may learn how to prevent or remove defects at less cost. The Personal Software Process (PSP) course provides a rich source of data describing developer effort and defects in a rigorously measured environment. In this report, the authors examine defects injected in the coding phase and later removed by developers using the full PSP2.1 process. The authors report the frequency of defect injections by type, the removal cost by type, and the phases in which defects of a given type are removed. Because the PSP process begins at detailed design and ends at unit test, some results may not generalize to a production development environment. Nonetheless, the authors report the significant findings that defects found in unit test are seven times more expensive to remove than those found earlier in the development, that approximately a quarter of all injections escape into unit test, and that escapes are dominated by defects that may have resulted from designing during the coding phase.

1 Introduction

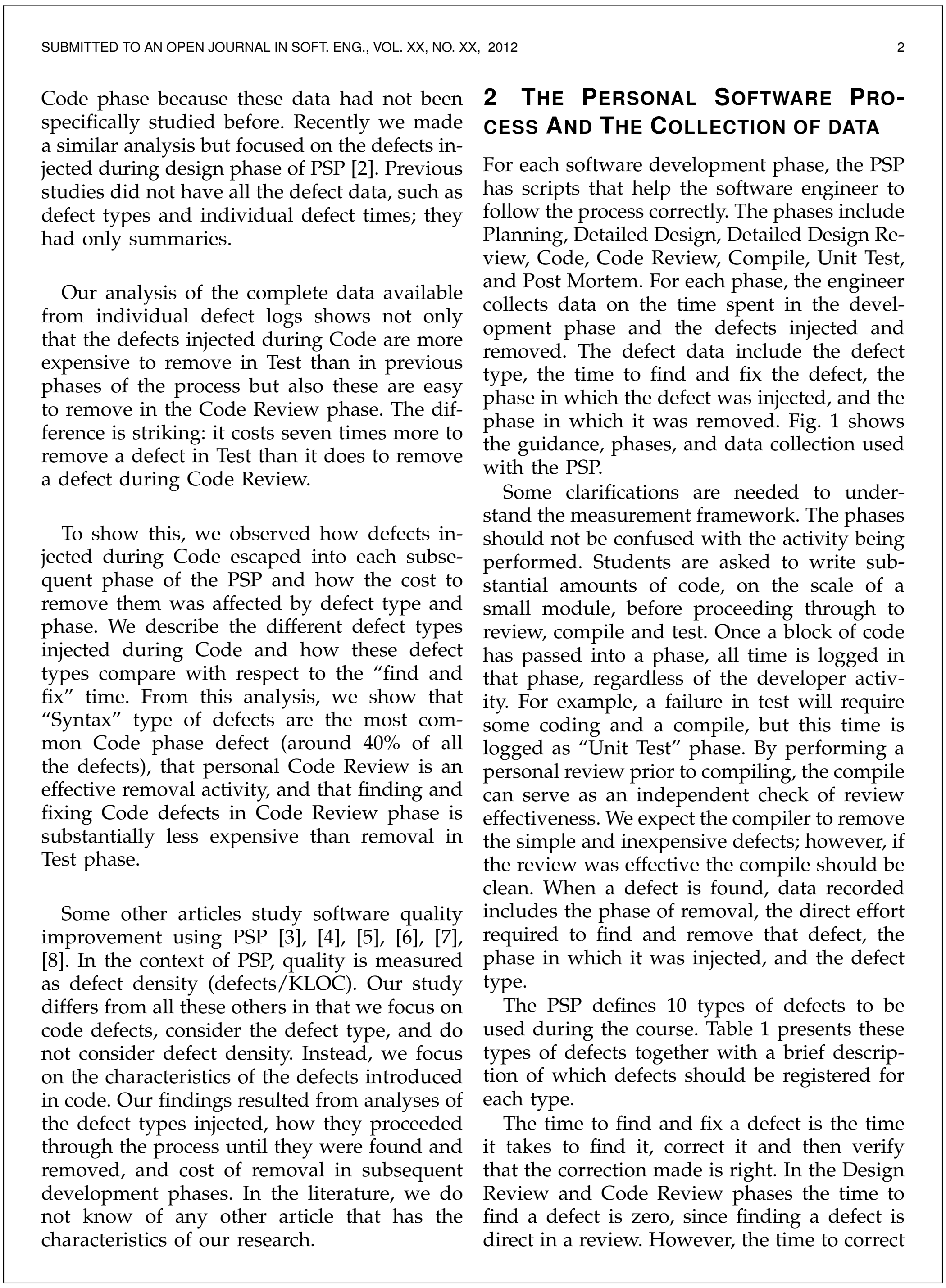
A primary goal of software process improvement is to make software development more effective and efficient. Because defects require rework, one path to performance improvement is to quantitatively understand the role of defects in the process. We can then make informed decisions about preventing defect injection or committing the effort necessary to remove the injected defects. The Personal Software Process (PSP) establishes a highly instrumented development process that includes a rigorous measurement framework for effort and defects. After examining a large amount of data generated during PSP instruction classes, we can describe how many defects are injected during PSP Code phase, the types of defects injected, when they are detected, and the effort required to remove them. We have found that even using a rigorous PSP development process, nearly a quarter of all defects injected will escape into unit test. Moreover, finding and removing defects in unit test required seven times as much effort as removal in earlier phases. The purpose of this study is not to measure the effectiveness of PSP training, but rather to characterize the defects developers inject and must subsequently remove. By examining the characteristics of defect injections and escapes, we might teach developers how to define and improve their own processes, and thus make the product development more effective and efficient.

"The Personal Software Process (PSP) is a self-improvement process that helps you to control, manage, and improve the way you work" [1]. This process includes phases that you complete while building the software. For each phase, the engineer collects data on the time spent in the development phase and data about the defects injected and removed.

During the Personal Software Process course, the engineers build programs while they progressively learn the PSP. The number of exercises of the PSP course version we analyze is eight. In this article, we present an analysis of defects injected during the Code phase of the last three PSP programs (6, 7, and 8); when building these programs the engineers used the complete PSP.

We focused on defects injected during the

- *D. Vallespir is with the Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay.*
  E-mail: dvallesp@fing.edu.uy
- *W. R. Nichols is with the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, United Stetes.*
  E-mail: wrn@sei.cmu.edu



Code phase because these data had not been specifically studied before. Recently we made a similar analysis but focused on the defects injected during design phase of PSP [2]. Previous studies did not have all the defect data, such as defect types and individual defect times; they had only summaries.

Our analysis of the complete data available from individual defect logs shows not only that the defects injected during Code are more expensive to remove in Test than in previous phases of the process but also these are easy to remove in the Code Review phase. The difference is striking: it costs seven times more to remove a defect in Test than it does to remove a defect during Code Review.

To show this, we observed how defects injected during Code escaped into each subsequent phase of the PSP and how the cost to remove them was affected by defect type and phase. We describe the different defect types injected during Code and how these defect types compare with respect to the "find and fix" time. From this analysis, we show that "Syntax" type of defects are the most common Code phase defect (around 40% of all the defects), that personal Code Review is an effective removal activity, and that finding and fixing Code defects in Code Review phase is substantially less expensive than removal in Test phase.

Some other articles study software quality improvement using PSP [3], [4], [5], [6], [7], [8]. In the context of PSP, quality is measured as defect density (defects/KLOC). Our study differs from all these others in that we focus on code defects, consider the defect type, and do not consider defect density. Instead, we focus on the characteristics of the defects introduced in code. Our findings resulted from analyses of the defect types injected, how they proceeded through the process until they were found and removed, and cost of removal in subsequent development phases. In the literature, we do not know of any other article that has the characteristics of our research.

2 The Personal Software Process And The Collection of data

For each software development phase, the PSP has scripts that help the software engineer to follow the process correctly. The phases include Planning, Detailed Design, Detailed Design Review, Code, Code Review, Compile, Unit Test, and Post Mortem. For each phase, the engineer collects data on the time spent in the development phase and the defects injected and removed. The defect data include the defect type, the time to find and fix the defect, the phase in which the defect was injected, and the phase in which it was removed. Fig. 1 shows the guidance, phases, and data collection used with the PSP.

Some clarifications are needed to understand the measurement framework. The phases should not be confused with the activity being performed. Students are asked to write substantial amounts of code, on the scale of a small module, before proceeding through to review, compile and test. Once a block of code has passed into a phase, all time is logged in that phase, regardless of the developer activity. For example, a failure in test will require some coding and a compile, but this time is logged as "Unit Test" phase. By performing a personal review prior to compiling, the compile can serve as an independent check of review effectiveness. We expect the compiler to remove the simple and inexpensive defects; however, if the review was effective the compile should be clean. When a defect is found, data recorded includes the phase of removal, the direct effort required to find and remove that defect, the phase in which it was injected, and the defect type.

The PSP defines 10 types of defects to be used during the course. Table 1 presents these types of defects together with a brief description of which defects should be registered for each type.

The time to find and fix a defect is the time it takes to find it, correct it and then verify that the correction made is right. In the Design Review and Code Review phases the time to find a defect is zero, since finding a defect is direct in a review. However, the time to correct

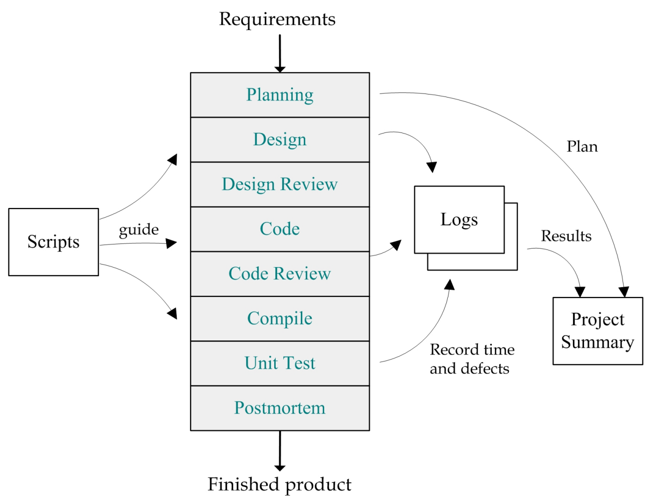


Fig. 1. The PSP phases, scripts, logs and project summary

TABLE 1
Defect types in PSP

Defect Type	Possible Defects for the Type
Documentation	Comments, messages
Syntax	Spelling, punctuation, typos, instruction formats
Build/Package	Change management, library, version control
Assignment	Declaration, duplicate names, scope, limits
Interface	Procedure calls and references, I/O, user formats
Checking	Error messages, inadequate checks
Data	Structure, content
Function	Logic, pointers, loops, recursion, computation, function defects
System	Configuration, timing, memory
Environment	Design, compile, test, or other support system problems

it and check that the correction is right depends on how complex the correction is.

On the other hand, both in the Compile and the Unit Test phases, finding a defect is an indirect activity. First, there will be a compilation error or a test case that fails. Taking that failure as a starting point (compilation or test) what causes it (the defect) must be found in order to make the correction and verify if it is right.

During the Personal Software Process course, the engineers build programs while progressively learning PSP planning, development, and process assessment practices. For the first exercise, the engineer starts with a simple, defined process (the baseline process, called PSP 0); as the class progresses, new process steps and elements are added, from Estimation and Planning to Code Reviews, to Design, and Design Review. As these elements are added, the process changes. The name of each process and which elements are added in each one are presented in Fig. 2. The PSP 2.1 is the complete PSP process.

In this article, we present an analysis of defects injected during the Code phase of the PSP, in programs 6, 7, and 8 (all developed using PSP 2.1, the complete PSP). In PSP 2.1, students conceptualize program design prior to coding and record the design decisions using functional, logical, operational, and state

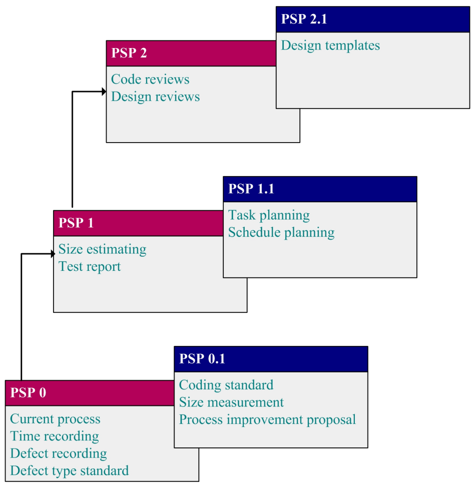


Fig. 2. PSP process level introduction during course

templates. Students then perform a checklist-based personal review of the design to identify and remove design defects before beginning to write code. After coding students perform a check-list based personal review of the code. After the review they compile the code and finally they make unit testing.

3 The Data Set

We used data from the eight program version of the PSP course (PSP for Engineers I and II) taught between October 2005 and January 2010. These courses were taught by the Software Engineering Institute (SEI) at Carnegie Mellon University or by SEI partners, including a number of different instructors in multiple countries.

This study is limited to only consider the final three programs of the PSP course (programs 6, 7, and 8). In these programs, the students apply the complete PSP process, using all process elements and techniques. Of course, not all of these techniques are necessarily applied well because the students are in a learning process.

This constitutes a threat to the validity of this study in the sense that different (and possibly better) results can be expected when the engineer continues using PSP in his working environment after taking the course. This is due to the fact that the engineer continues to assimilate the techniques and the elements of the process after having finished learning the PSP.

We began with the 133 students who completed all programming exercises of the mentioned courses. From this we made several cuts to remove errors and questionable data and to select the data most likely to have comparable design and coding characteristics.

Because of data errors, we removed data from three students. Johnson and Disney reviewed the quality of the PSP data [9]. Their analysis showed that 5% of the data was incorrect; however, many or most of those errors in their data were due to process calculations the students made. Process calculations are calculations made to obtain the values of certain derived measures the process uses to make estimates for the next program, such as the defects injected per hour in a certain phase or calculating the alpha and beta parameters for a linear regression that relates the estimated size to the real size of the program.

Because our data were collected with direct entry into a Microsoft Access tool, which then performed all process calculations automatically, the lower amount (2.3%) of data removed is lower than the percentage reported by Johnson and Disney but seems reasonable.

We next reduced the data set to separate programming languages with more common design and coding characteristics. As we analyze the code defects, it seems reasonable to consider only languages with similar characteristics that might affect code size, modularity, subroutine interfacing, and module logic. The students used a number of different program languages, as shown in Fig. 3.

The most common language used was Java. To increase the data set size, we decided to include the data generated by students who used Java, C#, C++, and C. This group of languages uses similar syntax, subprogram, and data constructs. For the simple programs produced in the PSP course, we judged that these were most likely to have similar modulariza-

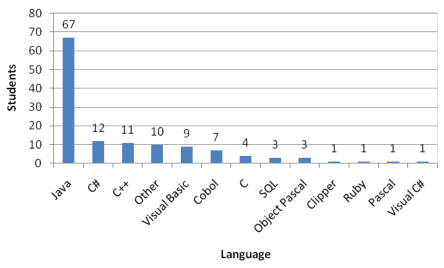


Fig. 3. Quantity of students by program languages

tion, interface, and data design considerations. This cut reduced our data to 94 subjects.

Because our intent was to analyze defects, we removed from consideration any data for which defects were not recorded. From the 94 engineers remaining, two recorded no defects at all in the three programs considered. Our data set for this analysis was, therefore, reduced to 92 engineers. In the following sections we present the types of defects that were injected in the code phase, when the defects were removed, and the effort required to find and fix these defects.

All our analysis studied the defect injection and removal performance of individuals and the performance range of variation among them. It should be clear that this is different from analyzing the behavior of a team. That is to say, we wanted to characterize the work of individual programmers, which is why we calculated individual performance for each one of the subjects. After obtaining the performance of each subject, we calculated an estimate of the mean percentage, the 95% confidence interval for that mean (to characterize the standard error on the mean), and the standard deviation of the distribution to characterize the spread among individuals. For these calculations, as it has already been mentioned, only programs 6, 7 and 8 of the PSP course were used in order to consider the complete PSP.

For this study we included the 92 engineers who recorded defects. In several cases, the number of engineers included varies, and in each of these cases the motive for varying is documented.

TABLE 2
Mean lower, upper confidence interval values and standard deviation of the % of defects injected by phase

	DLD	DLDR	Code	CR	Comp	UT
Mean	46.4	0.4	52.4	0.3	0.03	0.5
Lower	40.8	0.2	46.7	0.0	0.00	0.2
Upper	52.0	0.7	58.1	0.7	0.09	0.9
Std. dev.	27.2	1.7	27.4	1.8	0.30	1.8

4 Where the Defects are Injected

The first goal of our analysis was to better understand where defects were injected. We expected injections to be dominated by the Design and Code phases of course, because they are the construction phases in PSP. We began by explicitly documenting the phase injection percentages that occur during the PSP course.

For each PSP phase and for each individual, we calculated the percentage of defects injected. The distribution statistics are shown in Table 2.

The Design and Code phases have similar injection percentages both on average and in the spread. Their mean of the percentage of defects injected is near 50% with lower and upper CI bounds between 40% and 58%. Both standard deviations are around 27% with. So, in the average of this population, roughly half of the defects were injected in the Design phase and the other half of the defects were injected in the Code phase. On average, the defect potential of these phases appears to be very similar. The standard deviation shows, however, that the variability between individuals is substantial. Nonetheless, as we expected, in the average almost 99% of the defects were injected in the Design and Code phases with only around 1% of the defects injected in the other phases.

The Design Review, Code Review, Compile, and Unit Test phases also have similar average defect potentials. The average in all these cases is less than 0.5% and their standard deviations

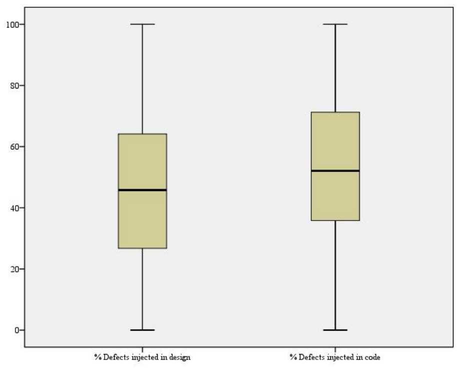


Fig. 4. Percentage of defects injected by phase (box and whisker chart)

are small, the largest being 1.8% in Code Review and Unit Testing. This shows that during verification activities in PSP the percentage of defects injected is low but not zero. From time to time, developers inject defects while correcting other defects. We will study these secondary injections in a later study.

The variability between individuals and the similarity between the Code and Design phase is also presented in Fig. 4. Note that the range in both phases is from 0% to 100% (all possible values). The 25th percentile is 26.34 for Design and 35.78 for Code, the median is 45.80 for Design and 52.08 for Code, and the 75th percentile is 64.22 for Design and 71.56 for Code.

Despite a high variability between individuals, this analysis shows that the great majority of defects are injected in the Design and Code phases. Slightly more defects are injected during Code than during Design, but the difference is not statistically significant. We could, therefore, focus on the defects injected in the Design and Code phases. In this article, we discuss only the defects injected in the Code phase.

5 Analysis of Code Defects

From the 92 engineers in our data set there were four whose records of injected defects (injected during Code) were uncertain regarding their correctness and therefore were dismissed for this analysis. Also, eight engineers did not record defects in the Code phase, so they were dismissed, as well.

Our data set for analysis of the code defects was, therefore, reduced to 80 engineers. In the following sections we discuss the types of defects that are injected in the Code phase, when those defects are removed, and the effort required to find and fix the defects.

5.1 Defect Types Injected during the Code Phase

To improve the detection of code defects we first wanted to know which types of defects were injected during the Code phase. Table 3 shows the mean of the percentage of the different defect types injected. It also presents the lower and upper bound of the 95% confidence interval for the mean (a measure of the standard error) and the standard deviation of the distribution.

None of the engineers registered System type defects injected during the Code phase. The Mean D. line presents what was found in our previous work of analysis of the defects injected during the Design phase, so these results could be comparable.

When seeking improvement opportunities a Pareto sort can be used identify the most frequent or most expensive types. These types can then be the focus for future improvement efforts. For the following analysis we sorted defects by frequency, and then segmented these defect types into three categories of increasing frequency. The first grouping is “very few” defects of this type. In the “very few” category we found System, Build/Package and Environment type of defects. In our previous work, in which the defects injected in the design phase were studied, we found within this category the System and Build/Package defect types but not the Environment type of defect. Considering this work and the previous work, it is clear that, during the PSP course, the build/package and system types of defects were seldom injected. This may be due to the PSP course exercises rather than the PSP. Because the exercises are small, taking only a few

TABLE 3
Percentage of defect types injected during code

	Docs.	Syn.	Build	Assign.	Inter.	Check	Data	Func.	Syst.	Env.
Mean	3.8	40.3	0.6	14.0	5.5	2.7	5.8	26.4	0	0.9
Mean D.	6.9	6.0	0.1	12.6	10.0	4.6	9.8	46.6	0.2	3.1
Lower	1.5	33.7	0.0	9.9	3.1	1.0	3.1	19.9	0	0.0
Upper	6.0	46.9	1.1	18.1	8.0	4.4	8.6	32.8	0	1.7
Std. dev.	10.1	29.5	2.5	18.4	11.1	7.4	12.4	29.1	0	3.9

hours, contain few components, and make few external library references, build packages are usually quite simple. We expect to find more defects of these types in the Team Software Process [10], [11] in industrial scale projects.

A second grouping is "few defects;" most of the other defect types (all except Syntax and Function types) are in this category. The percentage of defects in this category ranged from 2.7% to 14.0%. In our previous work both Syntax and Environment defect types were in this category. It is reasonable that, when analyzing the design defects mentioned, we find few Syntax defects and that the percentage of these defects in relation to the rest of the other types of defects increases when the defects injected during the Code phase are analyzed. It is natural that when coding, more Syntax defects are made than when designing, even if the design contains pseudocode, as in the case of the PSP.

The third and final grouping, "many defects," includes the Syntax and Function types of defects. The Syntax defects injected during Code are around 40% of the total defects and the Function defects are around 26%. Approximately two out of three injected defects during the Code phase are of one of these two types.

As mentioned earlier, one out of four (26.4%) defects injected during the Code phase is a Function type of defect. This is an opportunity for improvement for the PSP, since this type of defect should be injected (and as far as possible removed) before reaching Code phase. The fact that there is such an important injection in percentage of this type of defect indicates problems in the Design and Design Review phases. PSP incorporates a detailed pseudocode in the Design phase using the Logic Template. Due to this, it is in this phase where the Func-

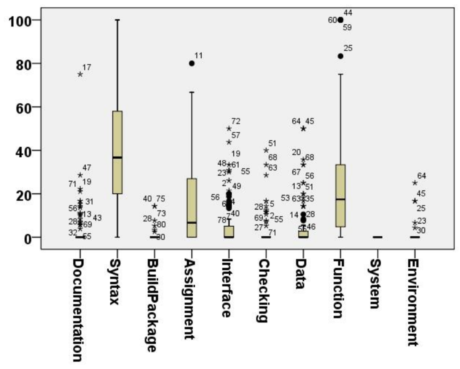


Fig. 5. Box and whisker of the percentage of defects injected during code)

tion type defects should be injected, and then they should be removed in the Design Review phase.

The lower, upper, and standard deviation data show again the high variability between individuals. This can also be observed in Fig. 5; the box and whisker chart shows many observations as outliers. The high variability among individuals is repeated in every analysis conducted, both in this work and in the previous work, which studied the defects injected in the Design phase. A detailed analysis of developer variability is beyond the scope of this report.

5.2 When Are the Defects Injected During Code Removed?

Our analysis indicated the subsequent phases during which the Code defects were removed. While our data set was large, it remained too small for us to examine the removals based on defect type. Still, for each engineer who injected

TABLE 4
Phases where the code defects are found (percentage)

	CODE DEFECTS		
	CR	Comp	UT
Mean	62.0	16.6	21.4
Lower	55.0	11.7	15.4
Upper	69.0	21.6	27.3
Std. dev.	31.3	22.4	26.9

defects in the Code phase, we identified the phases in which the engineer found the defects; then, for every phase, we determined the percentage of the defects that were found in that phase.

Table 4 shows the mean (with 95% confidence interval) and standard deviation for the different phases. As previously shown, the standard deviation was high, indicating the high variability between individuals. From this we learned that 62% of the defects injected during Code were found in the Code Review phase in average.

In our previous analysis, we found that around 50% of the defects injected during the Design phase were detected in the Design Review phase. This indicates that for the defects injected in both the Design and Code phases, the following Review phases are highly effective.

On the other hand, in the previous study we also found that around 25% of the defects injected in the Design phase are detected only in Unit Test. This happens in 21.4% of the cases, based on our analysis of defects injected in the Code phase. This indicates that a relatively high percentage of defects manage to escape from the different detection phases and reach Unit Test.

We also know, of course, that not all the defects that escape into Unit Test are found in Unit Test. This phase will not have a 100% yield. (That is, we will not find all the defects that are in a given unit when it arrives at Unit Test). Therefore the percentage of defects found in each of these phases is smaller than reported, while the actual percentage of escapes into Unit Test is a lower limit. An estimate or measurement of the Unit Test yield will be necessary to revise these phase estimates.

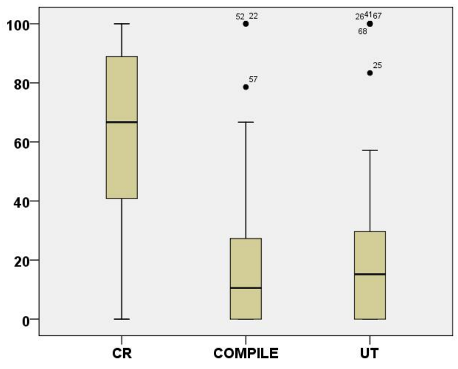


Fig. 6. which phase are the code defects found? - Variability between individuals

Fig. 6 shows the box and whisker charts displaying the percentage of defects found in the different phases. Figure 6 also shows the high variability between individuals in the percentage of defects found during Code Review, Compile, and Unit Test phases. This variability among individuals was also found in the previous study.

5.3 Cost to Remove the Defects Injected in Code

What is the cost and variation in cost (in minutes) to find and fix the defects that are injected during Code? First, we analyze the differences in cost, segmented by the removal phase. Second, we study the differences in cost segmented by defect type.

It would also be interesting to segment and analyze both the removal phase and the defect type jointly. Unfortunately, because of limited sample size after a two dimensional segmentation, we cannot perform that analysis with statistical significance.

5.3.1 Phase Removal Cost

What is the cost, in each removal phase, to find and fix a defect injected in Code? Code defects can be removed in PSP in the Code Review (CR), Compile, and Unit Test phases.

TABLE 5
Cost of find and fix defects injected in design segmented by phase removed

	CODE DEFECTS		
	CR	Comp	UT
Mean	1.9	1.5	14.4
Lower	1.5	1.1	9.8
Upper	2.3	1.9	19.0
Std. dev.	1.9	1.3	16.4

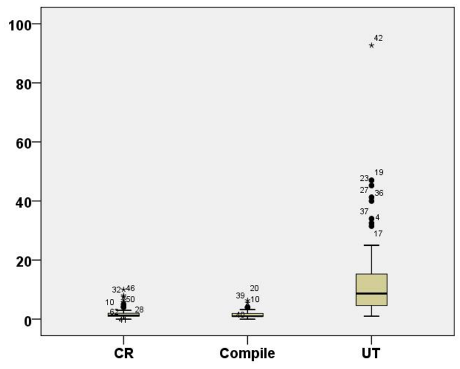


Fig. 7. Box and whisker of the cost of find and fix a code defect segmented by phase removed

For each engineer, we calculated the average task time of removing a design defect in each of the different phases. Because some engineers did not remove code defects in one or more phases, our sample size varied by phase. We had data from 72 engineers for Code Review, 44 for Compile, and 51 for Unit Test.

Table 5 shows the mean, lower, and upper 95% confidence intervals, and the standard deviation for the find and fix time (in minutes) for Code defects in each of the studied phases.

As we expected, the average cost of finding Code defects during Unit Test is much higher than in the other phases by a factor of 7. We are not stating here that the cost of finding and fixing a particular Code defect during Unit Test is 7 times higher that finding and fixing the same particular Code defect in Code Review or Compile. We are stating that using PSP, the code defects that are removed during Unit Test cost 7 times more than the ones that were removed in Code Review and Compile (these are different defects).

In our previous study we also found that the Design injection defects "find and fix" times in Design Review and Code Review are a factor of five smaller than the "find and fix" time in Unit Test .We also found that the defects injected in Design and removed in Unit Test have an average "find and fix" time of 23 minutes. Considering the two analyses, these are the defects that are most costly to remove. Defects injected in Code and removed in Unit Test follow with an average of 14.4 minutes. Testing, even at the unit test level, is consistently more a more expensive defect removal activity than alternative verification activities.

We also found high variability among individual engineers. This variability can be seen in the box and whisker chart in Fig. 7. We tested for normal distribution after log transformation of "find and fix" times for Code Review, Compile, and Unit Test. Only Unit Test is consistent (p ¿ 0.05 using Shapiro-Wilk test) with a log-normal distribution. The test for normality is primarily useful to verify that we can apply regression analysis to the transformed data; however, understanding the distribution also helped to characterize the asymmetry and long tailed nature of the variation. The log-normality confirmed our intuition that some defects found in Unit Test required far more than the average effort to fix, making Unit Test time highly variable. We observe that both the mean and variation of rework cost in the Code Review and Compile phases were significantly lower than Unit Test in both the statistical sense and the practical sense.

5.3.2 Defect Removal by Type

What is the "find and fix" cost, per defect type, of defects injected during code? We did not have enough data for a statistically significant analysis of the cost of removing the Build/Package, Checking, System, and Environment types of defects. However, we were able to analyze the remaining defect types.

Table 6 presents the mean, lower, and upper 95% confidence interval and the standard

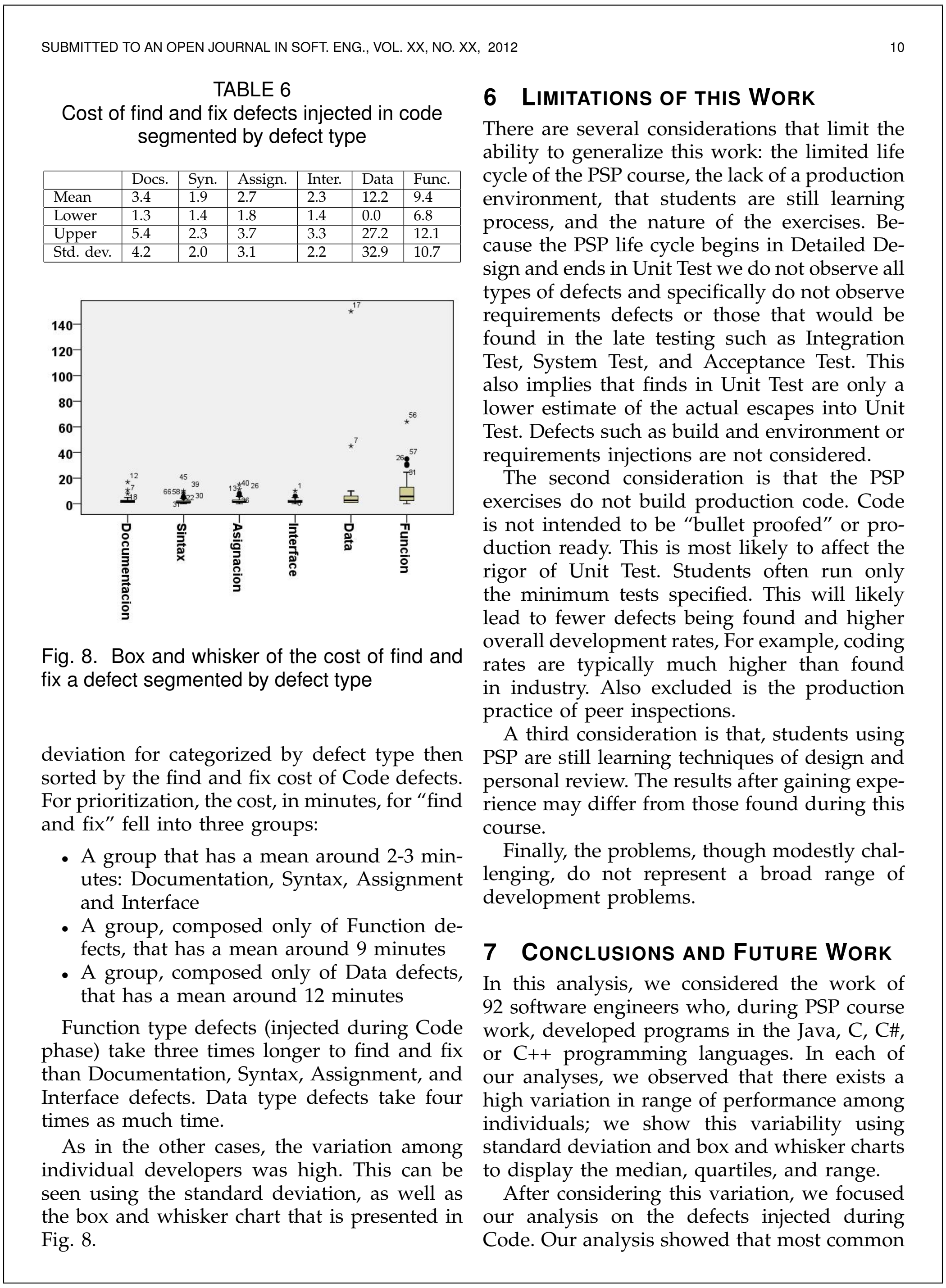


TABLE 6
Cost of find and fix defects injected in code segmented by defect type

	Docs.	Syn.	Assign.	Inter.	Data	Func.
Mean	3.4	1.9	2.7	2.3	12.2	9.4
Lower	1.3	1.4	1.8	1.4	0.0	6.8
Upper	5.4	2.3	3.7	3.3	27.2	12.1
Std. dev.	4.2	2.0	3.1	2.2	32.9	10.7

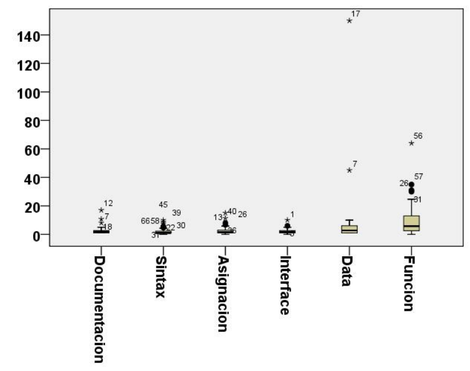


Fig. 8. Box and whisker of the cost of find and fix a defect segmented by defect type

deviation for categorized by defect type then sorted by the find and fix cost of Code defects. For prioritization, the cost, in minutes, for "find and fix" fell into three groups:

- A group that has a mean around 2-3 minutes: Documentation, Syntax, Assignment and Interface
- A group, composed only of Function defects, that has a mean around 9 minutes
- A group, composed only of Data defects, that has a mean around 12 minutes

Function type defects (injected during Code phase) take three times longer to find and fix than Documentation, Syntax, Assignment, and Interface defects. Data type defects take four times as much time.

As in the other cases, the variation among individual developers was high. This can be seen using the standard deviation, as well as the box and whisker chart that is presented in Fig. 8.

6 Limitations of this Work

There are several considerations that limit the ability to generalize this work: the limited life cycle of the PSP course, the lack of a production environment, that students are still learning process, and the nature of the exercises. Because the PSP life cycle begins in Detailed Design and ends in Unit Test we do not observe all types of defects and specifically do not observe requirements defects or those that would be found in the late testing such as Integration Test, System Test, and Acceptance Test. This also implies that finds in Unit Test are only a lower estimate of the actual escapes into Unit Test. Defects such as build and environment or requirements injections are not considered.

The second consideration is that the PSP exercises do not build production code. Code is not intended to be "bullet proofed" or production ready. This is most likely to affect the rigor of Unit Test. Students often run only the minimum tests specified. This will likely lead to fewer defects being found and higher overall development rates, For example, coding rates are typically much higher than found in industry. Also excluded is the production practice of peer inspections.

A third consideration is that, students using PSP are still learning techniques of design and personal review. The results after gaining experience may differ from those found during this course.

Finally, the problems, though modestly challenging, do not represent a broad range of development problems.

7 Conclusions and Future Work

In this analysis, we considered the work of 92 software engineers who, during PSP course work, developed programs in the Java, C, C#, or C++ programming languages. In each of our analyses, we observed that there exists a high variation in range of performance among individuals; we show this variability using standard deviation and box and whisker charts to display the median, quartiles, and range.

After considering this variation, we focused our analysis on the defects injected during Code. Our analysis showed that most common

Code defects (40%) are of type Syntax. This type of defect is the cheapest to find and fix (1.9 minutes). The types of defects injected in Code that are most expensive to correct are the Data (12.2 minutes) and Function (9.4 minutes).

In addition, the analysis showed that Build-Package, Systems and Environment defects were seldom injected in the Code phase. We interpreted this as a consequence of the small programs developed during the course, rather than as a characteristic of PSP as a development discipline.

We found that defects were injected roughly equally in the Design and Code phases; that is, around half of the defects were injected in Code. 62% of the Code defects were found early through appraisal during the Code Review phase. However, around 21% were discovered during Unit Test, where defect find and fix is almost seven times more expensive in time.

While this analysis provided insights into the injection and removal profile of Code defects with greater specificity than previously possible, a larger data set would allow us to consider more detail, such as the costs of defects discriminated by defect type in addition to removal phase. A more complete analysis may enable us to analyze improvement opportunities to achieve better process yields. In future analysis, we will examine the relationship between Design and Code activities and the defects found in the downstream phases.

REFERENCES

[1] W. S. Humphrey, *PSP A Self-Improvement Process for Software Engineers*. Addison-Wesley, 2005.

[2] D. Vallespir and W. R. Nichols, "Analysis of design defects injection and removal in psp," in *Proceedings of the TSP Symposium 2011: A dedication to excellence*, 2011, pp. 19–25.

[3] M. C. Paulk, "The impact of process discipline on personal software quality and productivity," *Software Quality Professional*, vol. 12, no. 2, pp. 15–19, 2010.

[4] C. Wohlin and A. Wesslen, "Understanding software defect detection in the personal software process," in *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, 1998, pp. 49–58.

[5] D. Rombach, J. Munch, A. Ocampo, W. S. Humphrey, and D. Burton, "Teaching disciplined software development," *The Journal of Systems and Software*, vol. 81, no. 5, pp. 747–763, 2008.

[6] M. C. Paulk, "Factors affecting personal software quality," *CrossTalk: The Journal of Defense Software Engineering*, vol. 19, no. 3, pp. 9–13, 2006.

[7] W. Hayes and J. Over, "The personal software process: An empirical study of the impact of psp on individual engineers," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. 97-001, 1997.

[8] P. Ferguson, W. S. Humphrey, S. Khajenoori, S. Macke, and A. Matvya, "Results of applying the personal software process," *Computer*, vol. 30, no. 5, pp. 24–31, 1997.

[9] M. P. Johnson and A. M. Disney, "A critical analysis of psp data quality: Results from a case study," *Empirical Software Engineering*, vol. 4, no. 4, pp. 317–349, 1999.

[10] W. S. Humphrey, "The team software process," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. 2000-023, 2000.

[11] ——, *TSP: Coaching Development Teams*. Addison-Wesley, 2006.

Chapter 11

Conclusions

The effectiveness of the different PSP phases was analyzed in this part. The phases considered are: design review, code review, compile and unit testing. The reviews use a checklist and unit testing has not defined which technique to use; which technique to use is a decision of the engineer.

The design review has a high effectiveness for the removal of the defects injected during the detailed design phase (more than 50%). On the other hand, the review of the code has a high effectiveness for the removal of the defects injected during the code phase (more than 60%).

The result obtained in the code review in PSP differs from the one obtained in the Experiment 2008 with the desktop inspection technique. The desktop inspection technique is comparable to the technique used in the code review phase of the PSP since both techniques review the code using a checklist. The results of the desktop inspection were close to 30% when a small program was used and close to 20% when the large program was used. This represents half and one third of the effectiveness of what the code review in PSP presented.

There may be many reasons why these differences existed. Knowing what they are implies that more experiments must be conducted especially in order to find these reasons. Anyway, below we present some of the reasons for this, in our opinion:

- *Practice makes perfect*; in the Experiment 2008 the subjects used the technique only twice (training and execution) while in the PSP course it is used several times. During the PSP course the code review technique is taught and explained very carefully. Besides, this technique is improved as the engineer developed the exercises of the course and also from the comments and corrections made by the instructor about the use of the technique.

- Knowledge of the code or even that it is the code developed by the reviewer himself, may have an impact on the effectiveness of a review.

The program reviewed in the Experiment 2008 was not developed by the person who reviews it. Besides, the moment in which the review is performed is the first time he has contact with the program. On the other hand, the person who reviews the code in PSP is the developer himself.

- The checklists adapted to the defects the person who developed the program usually injects is normally a success factor in the review. The checklist used during the Experiment 2008 is a generic checklist that was given to all the subjects. On the other hand, the checklist used in the PSP is created by the engineer himself from the defects he himself injected in the first exercises of the course.

These possible reasons can help design experiments that try to reject them or accept them.

We also found that the cost of finding and fixing defects in unit testing is several times more expensive than finding and fixing defects in the previous phases. For the defects injected during the design, the cost is five times higher and for the defects injected in code, the cost is 7 times higher. Besides, on average, the defects that are most expensive to remove are the ones injected in design and removed in unit testing.

To show how expensive the "escape" of defects is from where they are injected to the unit test phase we give an example. Some data will be used in an approximate form and a simplification of the problem will be done.

Example: A developer using the PSP injects 100 defects in total (100 detectable defects in the use of the PSP, that is to say, defects that are found in integration tests or in system test are not considered). Using the statistical data we found, 47 will be injected during the Design phase and 53 during the Code phase.

The 47 defects of the Design phase will be found and fixed in the following phases and in the following quantities: 25 in Design Review, 5 in Code, 4 in Code Review, 1 in Compile, and 12 in Unit Test. For each one of these phases we have an average cost to find and fix these defects. Multiplying the number of defects that are found in each phase by the average time you get the time (expressed in minutes) that is employed in each phase to remove the defects injected in design: 133 in Design Review, 26 in Code, 17 in Code Review, 1 in Compile[1], 276 in Unit Test.

Making the same type of calculation with the 53 defects injected in Code the following is obtained: 33 are found in Code Review, 9 in Compile and 11 in Unit Test. The times in minutes used in each phase to find and fix them is: 63 in Code Review, 14 in Compile and 158 in Unit Test.

[1]Considering one minute the removal time in Compile which is an approximate value since there were not enough data to obtain it with statistical validity

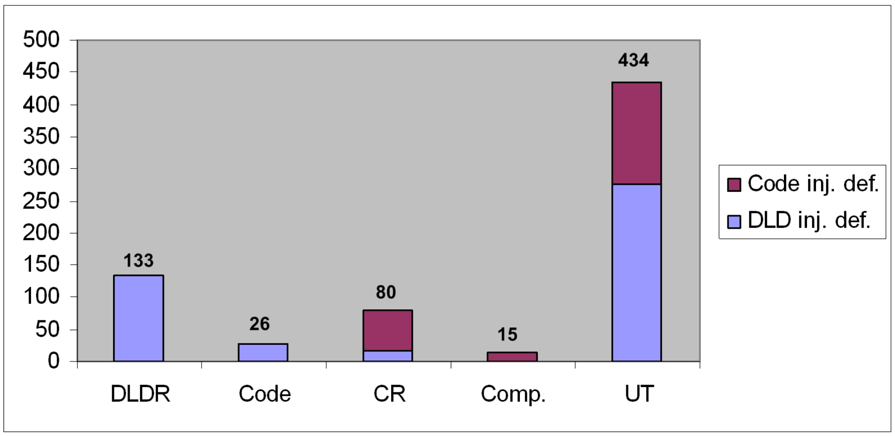


Figure 11.1: Time used per phase to detect and correct 100 defects.

Figure 11.1 shows the total cost (in minutes) of finding and fixing 100 defects in PSP per phase in which they are detected. The find and fix cost of the defects that get to Unit Test represent 63% of the total cost of finding and fixing all these defects. Looking for ways of finding a larger number of defects before Unit Test may imply a substantial reduction in the time employed in the detection and correction of defects. It is clear that waiting to reach Unit Testing or basing on it as the method of defect detection can be a bad idea.

It is important to clarify that the calculations of the costs of the techniques in Part I are different from the calculations in the articles of this Part. The costs in the first Part are related to the use of the technique. They measure, in minutes, how long it takes to execute the testing technique used by a subject. The cost analyzed in the context of the PSP is the cost of finding and fixing defects; independently from the total cost of the technique that was used.

Finally it is worth mentioning that the number of subjects of the experiments of the first Part and of this Part is very different. The PSP courses have been taught for a long time in several parts of the world. Thanks to this there is an important amount of data available ready to be analyzed.

Part IV

Conclusions and Future Work

Chapter 12

Conclusions and Future Work

This chapter includes three sections: conclusions, contributions of the research and future work.

12.1 Conclusions

The development of software is an intellectual and creative activity performed by human beings. During this activity, as it is natural because of our human condition, mistakes are made. These become defects in different software products that may later cause failures during the execution of the software by the users.

In this context software testing is extremely important. Testing aims at detecting defects before the product is used by the users. Unfortunately, knowledge about the effectiveness and cost of the testing techniques is not enough. The fact that we do not have this information makes it difficult to choose the best testing strategy that combines different techniques (of different types) at different moments of the software development process.

The general goal of this thesis is to contribute to the knowledge of different testing techniques. Our principal goal is:

Goal 1 - Investigate empirically 7 particular testing techniques: desktop inspection (DI), equivalence partitioning and boundary-value analysis (EP), decision table (DT), linearly independent path (LIP), multiple condition coverage (MCC), statement coverage (SC) and all uses (AU).

Research question 1: Which is the effectiveness of each one of those techniques?

Research question 2: Which is the cost of each one of those techniques?

In this work we study empirically these testing techniques in an isolated form. These techniques combine static and dynamic techniques, white and black box techniques, as well as techniques based on control flow and data flow.

In order to study them, two controlled experiments were conducted: the 2008 and 2009 Experiments. These experiments were conducted in the frame of courses at the Computer Science Institute of the Universidad de la República designed especially to execute them. The subjects of the experiments were all Computer Science students.

In “Reviewing 25 years of testing technique experiments” [1] and in “A look at 25 years of data” [2] the authors detect that the majority of the software programs used in the experiments “suffer from at least one of two problems: they are small and unusually simple, or the researchers seeded the defects rather than looking for naturally occurring ones”. In our experiments we use 4 programs each of which is developed especially for the experiments. 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. This way we try to solve the two problems mentioned.

The authors are more explicit as regards what the programs used in the experiments are like: “They [the researchers] use relatively small programs, between 150 and 350 LOC, which are generally toy programs and might not be representative of industrial software”. The four programs used in our experiments have the following number of lines of code: 468, 566, 828 and 1820. We believe that our programs are suitable for conducting unit testing experiments as far as their size, complexity and similarity with the software development industry are concerned.

The subject attended training sessions as part of the experiment. The aim of the training was that the subjects should learn the techniques and apply them correctly. The final part of the training consisted in applying the techniques on a small program. That is why the training could be analyzed as an experiment in itself. After the training the subjects applied the allotted technique on larger programs.

In the analysis of the results we found that the effectiveness of all the testing techniques was low. None of the techniques was more than 35% effective. This indicates that quality has to be built during the construction phase and not during the testing phase.

Besides, the effectiveness of the techniques decreased when they were used in the experiment programs in relation to when they were used in the training. This is certainly due to the fact that it is easier to find defects in trivial programs than it is to find them in complex ones. Conducting other experiments combining a greater number of trivial programs with more complex ones will provide us with more information to enable us to reject or accept this hypothesis.

The training of the Experiment 2008 shows that DI detects a greater variety of defects than the other techniques. Among the 3 testers applying the inspection 9 out of the 13 defects of the software program used in the training are found. This analysis was not repeated in the formal Experiment 2008.

We also found that dynamic techniques have problems in finding NF defects; defects that do not provoke a failure during execution. Using EP and DT the participants did not discover any of the NF defects. With MCC, SC and AU only one NF defect is found. The reason for this is that dynamic techniques are based on the execution of the program. So defects that do not produce a failure are never sought directly. In the two articles mentioned the authors find that "it seems that some types of faults are not well suited to some testing techniques".

The conclusion that dynamic techniques have problems with NF defects is quite expected. However, in our experiment the subjects review the code twice when using white box techniques and once when using black box techniques. First they review the code thoroughly to generate a group of test cases that satisfy the coverage that the assigned technique requires (the subjects that have allotted a white box technique). Then they review the code for each JUnit failure in order to find the corresponding defect. What this experiment shows is that the attention of the subjects concentrates only on the task they are performing. It is difficult for them to detect defects while they are generating test cases. It is also difficult to detect a defect they are not looking for.

In "Reviewing 25 years of testing technique experiments" and in "A look at 25 years of data" the authors study the experiments conducted to evaluate testing techniques (up to the date of the articles). These two works present the existing empirical knowledge on the topic, problems found and research opportunities in the area. We shall discuss the results of our experiment in the light of these two articles.

In the 2009 Experiment we did not find statistically valid differences between the AU technique and the SC technique. The authors state in the articles mentioned that "there does not appear to be a difference between all-uses and all-edges testing as regards effectiveness from the statistical viewpoint, as the number of programs in which one comes out on top of the other is not statistically significant". Given that all-edges is a more demanding technique than SC it is reasonable to expect that if we did not find statistically valid differences between the effectiveness of AU and SC, differences will not be found between AU and all-edges either.

The authors also mention that from a practical point of view "all-edges is easier to satisfy than AU". Our results indicate with statistical validity that the cost of using AU is higher than the cost of using SC (50% more expensive). These results are consistent because in theory all-edges is more expensive to execute than SC (however, we have not found experiments to validate this).

The authors also state that the "boundary analysis technique appears to behave differently compared with different structural testing techniques (particularly, sentence coverage and condition coverage)". In the Experiment 2008, which uses the EP technique (equivalence partition and boundary-value analysis), we do not find statistical evidence in the effectiveness between EP and the structural MCC technique (it is worth mentioning that it is the first time the MCC technique has been used in an experiment.)

However, we did find statistical difference in the effectiveness of EP and the structural LIP technique; EP being more effective than LIP. It must be borne in mind that the LIP technique was ruled out from the results analysis of the training of the Experiment 2008 due to the fact that the subjects probably used it in an incorrect way. In spite of the fact that the LIP technique was explained again to the subjects to whom it was assigned, we are uncertain as to how it was used in the 4 programs of the experiment. Knowing if LIP was used properly requires an analysis of the data that has not been done yet.

In the Experiment 2008 we used DT (the other black box technique used). In this experiment there was no statistically valid difference between the effectiveness of DT and the effectiveness of EP. However, the effectiveness averages did present differences. DT was more effective than EP, both in the training and in the experiment.

We have not found formal experiments that use DT in literature. Since we obtained differences in the effectiveness averages between DT and EP it seems interesting that the research community should conduct formal experiments that compare both techniques or even resort to using DT instead of EP in their experiments as a "representative" technique of the black box techniques.

In the two experiments we conducted we found a high variability between the subjects that applied the same technique in the same program. This is consistent with what was found in the review of 25 years of experiments: "There appears to be a dependence on the subject as regards technique application time".

It is important to point out that we have no knowledge of other experiments conducted in which the defects are corrected. The correction of defects by the researchers during the experiment simulates better the testing activity in the industry.

We also did research work that complements the central research line. These works have been very diverse and were presented in Part II. All of them aim at contributing in some lateral way to the central line of investigation.

The work of analysis of taxonomies presents a study and comparison of different defects taxonomies. The results of this work are interesting in themselves beyond the central research topic of this thesis. As far as the research line is concerned, we have planned to conduct experiments that study the effectiveness of the techniques segmented by type of defect both in a generic frame as in the frame of the PSP. The study of taxonomies will help determine how and with which taxonomy to classify the defects of future experiments.

The framework of comparison of experiments that evaluates testing techniques needs to be improved and extended. One of the most important conclusions of this work is that the researchers present the experiments conducted in very different forms and take into consideration different aspects of the same. This makes experimental replication very difficult. The article that presents the packaging of the Experiment 2008, makes it clear that a normalized packaging helps replication.

The last article of Part II presents the importance of data quality in software engineering experiments. Namely, it shows the importance of conducting a process of data cleansing before performing statistical analyses.

The second goal of this thesis is to make an initial study of the injection and removal of defects in the PSP. These studies are incipient and represent a research line that we are starting. They will serve as a basis to later make studies of the effectiveness and cost of the different testing techniques used in the PSP unit phase test.

Goal 2 - Investigate empirically the PSP defect injection and removal.

Research question 3: Which is the effectiveness of the different PSP's phases?

Research question 4: Which is the cost of finding and fixing a defect in the PSP?

In order to answer these research questions we analyzed data of the PSP courses to know the effectiveness of the phases of the process: design review, code review, compile and unit testing. In order to make this analysis we used data from the PSP courses taught between October 2005 and January 2010. These courses were taught by the Software Engineering Institute (SEI) of the Carnegie Mellon University or by SEI partners.

We found that the design review has a high effectiveness to remove the defects injected during the detailed design phase (more than 50%). On the other hand, the code review has high effectiveness to remove the defects injected during the code phase (more than 60%). This contrasts with what was found in the Experiment 2008 with the desktop inspection technique. It would be interesting to conduct experiments in order to find the reasons for these differences. Some possible causes were presented in Chapter 11.

We also found that the cost of finding and fixing defects in unit testing is several times more expensive than finding and fixing defects in the previous phases. For the defects injected during the design, the cost is five times higher and for the defects injected in code, the cost is 7 times higher. Besides, on average, the defects that are most expensive to remove are the ones injected in design and removed in unit testing. We also show that the find and fix cost of the defects that get to Unit Test represent 63% of the total cost of finding and fixing all the defects.

The controlled experiments were the first experiments in software engineering in the Universidad de la República. Thus an important milestone is established for the Software Engineering Group of the University. Our future research work takes place within the frame of empirical software engineering. In fact, this thesis marks the line of research of the Software Engineering Group of the Computer Science Institute of the Universidad de la República.

During this thesis we have stressed its use as a frame for the creation of the Software Engineering Group as a research group. This has involved an important effort, but it has been gratifying at the same time since we believe that we have been able to advance in the consolidation of this group. We have formed a group of people who want to carry out research in software engineering in Uruguay, but also with international contacts in Brazil and the United States. All of them appear as co-authors in the articles included in the thesis.

12.2 Contributions of the Research

Contributions to the main research line

- We give insight into the effectiveness and cost of 7 different testing techniques, varying from dynamics to statics and from white box to black box techniques. Also we use data flow and control flow based techniques in the white box ones.

- The construction of 4 more complex programs than those used habitually in the formal experiments of the area and having real defects (as opposed to defects injected by the researchers) is a contribution to the material to conduct formal experiments. Also, when the programs are tested by the subjects, it is also the first time they are tested (like in "real life"). The 4 programs are available for the research community.

- In the experiments we simulate the correction of defects, activity normally conducted in the software industry during unit testing. This is a novelty in the experiments conducted in the area; therefore it contributes to the way of executing them.

- We use 3 techniques that we do not find were used in experiments before: DT, LIP and MCC. So, our experiments are the first ones using these techniques.

- We present a comparison framework for comparing defect taxonomies. We made a deep analysis of different defect taxonomies that are proposed in the literature and we evaluate them using the proposed framework.

- We show that it is not easy to compare formal experiments and we present a first draft version of a framework in order to make this comparison easier.

- We study the data quality of one of the experiments. We did not find in the literature other studies like this one, at least not using a formal approach using the Data Quality discipline. We believe that every experiment must use this kind of analysis and data cleansing before making the statistical analysis.

Contributions related to PSP data analysis

- We present a new analysis of the PSP courses data. This new analysis considers the defects types and the behavior of the different phases of the PSP regarding the effectiveness divided by defect type. This was not studied before.

- We gain knowledge about the cost of finding and fixing defects that were injected during design and code in the PSP.

- A minor contribution is that we made a software product that loads the individual data bases of PSP students into a single data base. This enables us to manage the data of the different PSP courses better and makes it possible to make statistical analysis in an easier way.

12.3 Future Work

"*A team is a group of people who share a common goal. They must all be committed to this goal and must all have a common framework to guide them as they work to achieve the goal. (...)*

Teams derive their power from the way the members cooperatively work to achieve their goals. While it is essential to have trained and capable individuals on the team, the team's true power comes from its composite performance. This does not mean that the skill and ability of the members is not important, but that teams are most effective when their members cooperatively use all of their talents and skills to do their work. (...)

The self-directed team is the most powerfull tool humankind has devised for addressing challenging tasks."

– W. S. Humphrey, 2005.

In this section we present specifically what research work we are conducting at the moment and which is planned to be done.

We do not have plans to replicate the experiments of Part I in the way they were performed. In other words, building a particular and ad hoc course for conducting the experiment. However, we are building new courses for the new post-graduate degree of the School of Engineering "Specialization in Software Engineering". In some of these courses the students do laboratory works. These could be used as controlled experiments. The course "Software Inspections" that we will teach with Fernanda Grazioli could serve the purpose of conducting controlled experiments with desktop inspection technique and also with software Inspections. Lucía Camilloni, Cecilia Apa and I are preparing a software unit testing course where the students will work with black and white box techniques developing test cases in JUnit. This laboratory can be used also as a controlled experiment, both for the analysis of the effectiveness and cost and for the analysis of satisfaction of the techniques.

In a research work we are finishing now we have detected that the subjects manage to satisfy the SC criterion but not the AU criterion. This is almost finished and an article written together with Natalia Juristo, Carmen Bogado and Silvana Moreno is in a draft version: "Do the Testers Satisfy the Technique Prescription? An Empirical Study". Using the same data we want to know whether when the subjects use the SC testing technique they also satisfy other coverage criteria. The research questions should be the following: When the testers use the SC technique, do they also manage to satisfy decision coverage, loops coverage, etc? We are conducting this work together with Carmen Bogado.

We have planned, together with William Nichols, to study from the practical point of view of software engineering, the results presented in Part III. It will present both for the professional who uses the PSP and for the practitioner in general, the enormous differences we found between the reviews and the unit tests.

William Nichols and I have also planned to replicate the study conducted having a larger number of PSP courses. Different PSP courses have had different support tools for the collection of the data of the engineers: paper, electronic spreadsheets and Microsoft Access. The analyses we performed were restricted to courses that used Microsoft Access. In order to make the analysis, a prototype was built that reads each student's Microsoft Access files automatically and uploads them to a unified database. At present Rosana Robaina is building a tool that will enable us to read both the records of the electronic spreadsheets and the Microsoft Access to later save them in the unified database. This would make it possible to replicate the study with a much larger number of courses.

Carolina Valverde, Fernanda Grazioli, Adriana Marotta and myself are carrying out a study of the data quality of the PSP courses. The same concepts and steps applied in the study of the data quality of the Experiment 2008 are being applied for this study.

Silvana Moreno is making an adaptation of the PSP so that it can be used together with formal methods, in her master thesis whose tutor is Alvaro Tasistro. In particular, the study is being conducted considering Java programming language and the JML modeling language. Together with the two of them we are finishing an article for the journal Milveinticuatro that presents the JML using the *merge* example of the unit testing articles published in that same journal.

It has been planned for a more distant future to add white box unit tests to the PSP and study the impacts of effectiveness and cost in the PSP process.

In a more distant future we intend to try to estimate the number of remaining defects after the Unit Test phase in the PSP. Thus it would be possible to estimate the effectiveness of each one of the phases of the PSP.

References

[1] Natalia Juristo, Ana M. Moreno, Sira Vegas. 2004. Reviewing 25 Years of Testing Technique Experiments. Empirical Software Engeneering, 9, 1-2 (March 2004), 7-44.

[2] Natalia Juristo, Ana M. Moreno, Sira Vegas, Forrest Shull. 2009. A Look at 25 Years of Data. IEEE Software, 26, 1 (January 2009), 15-17.

Part V

Appendices

Appendix A

Unit Testing in Java

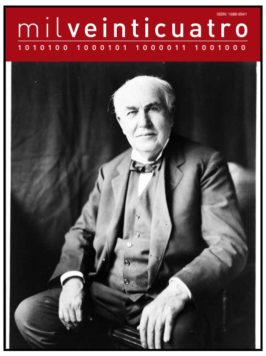
This appendix deal with software tools for unit tests. The main topic is code coverage and the support through tools to find that coverage.

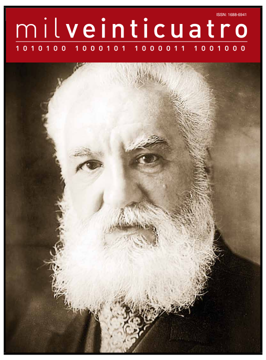
The experiments presented in Part I include white box testing techniques: Multiple Condition Coverage, Linearly Independent Path, Sentence Coverage and All Uses. During these experiments no code coverage tools were used. It is interesting to replicate the experiments conducted adding the use of code coverage tools when white box techniques are used. This was the main reason to start studying the existing tools for Java.

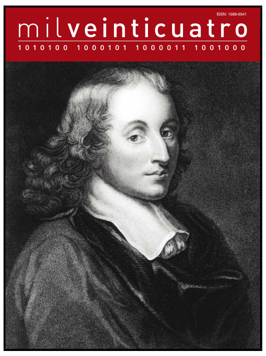
The PSP has a Unit Testing phase. The process does not define which unit tests to use; this is decided by the engineer that is using the PSP. It is in our interest to conduct experiments that add white box unit tests to the PSP. In this context the engineer would use tools to know the achieved coverage.

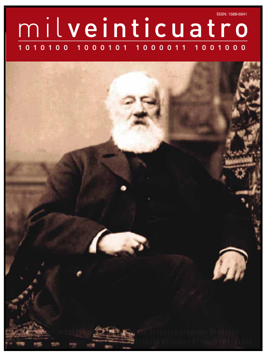
The articles on the tools for unit testing in Java were all published in different editions of the Milveinticuatro Journal, journal for divulging information on different topics and mainly in information technology in Uruguay. Publishing in this journal was seen as an opportunity to approach the software industry by the Software Engineering Group spreading the work we did.

ARTICLES









Unit Testing in Java:

- **JUnit and TestNG**
  Adriana Ávila, Lucía Camilloni, Fernando Marotta, Diego Vallespir and Cecilia Apa
  Journal Milveinticuatro, Thomas Alva Edison Edition, pp 46-48, ISSN: 1688-6941, 2010.

- **Code Coverage with Clover**
  Adriana Ávila, Lucía Camilloni, Diego Vallespir and Cecilia Apa
  Journal Milveinticuatro, Alexander Graham Bell Edition, pp 50-53, ISSN: 1688-6941, 2011.

- **Code Coverage with CodeCover**
  Carmen Bogado and Diego Vallespir
  Journal Milveinticuatro, Blaise Pascal Edition, pp 32-34, ISSN: 1688-6941, 2011.

- **Code Coverage with CoView**
  Diego Vallespir and Fernando Marotta
  Journal Milveinticuatro, Blaise Pascal Edition, pp 60-62, ISSN: 1688-6941, 2011.

- **Code Coverage with CodeCover and CoView**
  Diego Vallespir, Fernando Marotta and Carmen Bogado
  Journal Milveinticuatro, Antonio Meucci Edition, pp 32-34, ISSN: 1688-6941, 2011.

- **Data Driven with TestNG**
  Adriana Ávila, Lucía Camilloni and Diego Vallespir
  Journal Milveinticuatro, Antonio Meucci Edition, pp 60-61, ISSN: 1688-6941, 2011.

Unit Testing in Java: JUnit and TestNG

They are very important since they are used as a frame for many other software tools.

In the last few years unit tests have become very important in the development of software. These tests aim at finding out the way software units work at an early stage. One of the reasons why these tests are better known and more used now is the incorporation of agile methodologies such as Scrum and XP.
Several aspects must be considered when unit testing is incorporated to the development of software: The following are some of them:

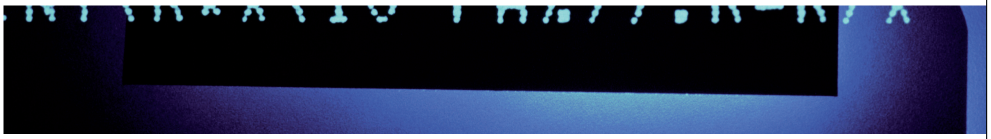
-What a software unit is
-Who designs and conducts the tests, how and when this is done
-When the unit can be released (release criterion)
-Which measurements (metrical) must be used
-What tools to use

The introduction of unit testing in an organization can be a failure if these and other aspects are not taken into consideration. Probably, the greatest risk run with unit testing is "falling" in a code-fix cycle. This type of cycle takes place normally when you depend only on unit testing to construct quality units. All that is achieved in such cases is low quality "mended" units. This discussion was presented in the article "TSP/PSP in Uruguay" published in this journal in its Isaac Newton volume.
Some of the factors to be considered when introducing unit testing in an organization will be presented in this article and a brief presentation of two drivers (tools to run the test) for Java language, JUnit and TestNG will be made.

Who, how and when

Although there are different approaches and some development processes propose variations, usually unit tests are conducted by the same person who builds the unit. Knowledge of the unit as a consequence of having built it makes it easier to find the defects quickly after the failure of a test. However, there are different variants. For example, the design of the tests is made by a different person from the one who developed the unit: but the execution and correction of the defects are carried out by the developer.

When introducing unit tests in an organization it is also important to consider how they are going to be conducted. This entails the definition of the strategy and the unit tests techniques to be used. An interesting approach would be to combine black box and white box techniques.
Black box techniques are tests developed from the specification of the unit to test. In these, the important thing is to test the functionality that the unit must have.
White box techniques is based on the code of the unit. The most important thing about these tests is being able to cover (execute) certain statements and conditions of the code.
There are different black box and white box techniques. The introduction of unit testing in an organization must be done with a clear definition of which techniques are going to be used. Without this definition different units built by different developers will have significant differences in the quality of the released unit.



One of the most important things to define is when the tests are going to be designed and conducted. It is now when it is defined whether the organization is going to do code-fix or not. It is a bad idea to depend on unit testing; several empirical studies indicate that its effectiveness is approximately 50%. This means that if 4 defects are detected during these tests (and are corrected), the unit will have approximately four more defects when it is released.

Luckily, it is very easy not to depend on these tests and take the most advantage of them: review before testing. The moment to conduct a unit test is always after being sure that what has been constructed is a good quality unit. In order to do this, it is necessary to design the unit, review the design, codify the unit and review the code in a static way. Then, unit tests will be a validation of the fact that what has been constructed is a quality unit, detecting and correcting very few defects while testing. Unfortunately, dependence on unit testing is a common mistake in the software development industry today.

Driver

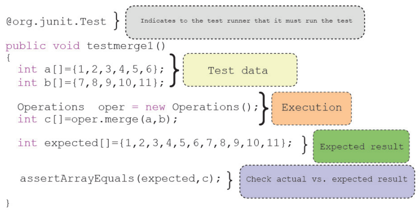
A driver is a software part created to simulate the invocation to the unit which you wish to test. It is what provides the data of the test cases to the software unit being tested and checks the actual versus expected results.
JUnit and TestNG are two of the most used drivers in Java language.

JUnit and TestNG

JUnit is a framework created by Erich Gamma and Kent Beck used to automate unit testing in Java language. Testing in JUnit is conducted through test cases that are written in Java classes. One of the most useful aspects of JUnit is that the test cases are defined to be conducted at any time; therefore it is simple to check, after making modifications to the program, whether those changes have not introduced new defects. This is known as regression testing.
TestNG is a framework created by Cedric Beust also to test software written in Java language. Its aim is to cover a wide variety of needs existing in the different types of tests, from unit tests to integration tests.

This tool was created based on JUnit and NUnit (for.NET), but introducing new features that make it more powerful and easy to use. Among the features of this tool are : annotations, JDK 5, flexible configuration of tests, software for testing using the data-driven approach, support for parameter passage, and allows to define parallelism at testing level.

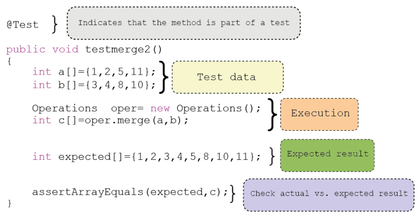
Figure 1 and figure 2 show how test cases in JUnit and TestNG respectively are generated; you will see the immense similarity in the way of codifying the test. The method under test is the merge method. This method receives two arrays of integers (each one ordered from smaller to larger) and returns an array that is the ordered union of the elements of both arrays. The method signature is the following:



```
@org.junit.Test

public void testmerge1()
{
  int a[]={1,2,3,4,5,6};
  int b[]={7,8,9,10,11};

  Operations  oper = new Operations();
  int c[]=oper.merge(a,b);

  int expected[]={1,2,3,4,5,6,7,8,9,10,11};

   assertArrayEquals(expected,c);

}
```

Indicates to the test runner that it must run the test
Test data
Execution
Expected result
Check actual vs. expected result

Figure 1 – Example in JUnit



```
@Test

public void testmerge2()
{
    int a[]={1,2,5,11};
    int b[]={3,4,8,10};

    Operations  oper= new Operations();
    int c[]=oper.merge(a,b);

    int expected[]={1,2,3,4,5,8,10,11};

    assertArrayEquals(expected,c);
}
```

Indicates that the method is part of a test
Test data
Execution
Expected result
Check actual vs. expected result

Figure 2 - Example in TestNG

Both JUnit and TestNG have great importance since they are used as frameworks for many other software tools for unit testing; for example coverage, mutant and data-driven tools. Although both perform the function of a driver, TestNG adds at least three features that JUnit does not have. They are:

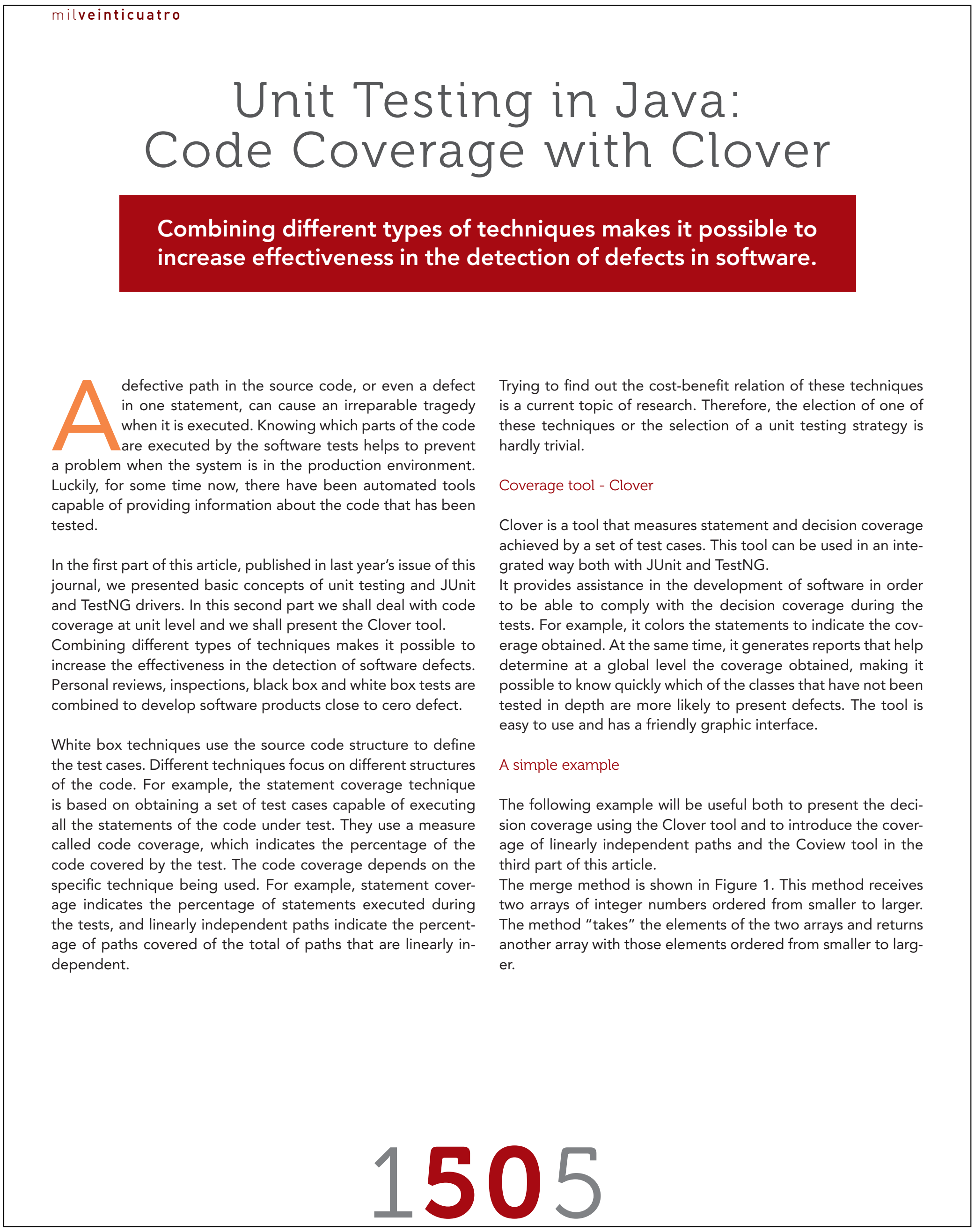
-It makes it possible to pass Java Objects as parameters of test cases through the support to the tests that use the data-driven approach.

-It makes it possible to define test groups, which provides great flexibility when conducting the tests.
-It makes it possible to define dependence between tests. It is possible to indicate which tests it does not make any sense to conduct when others fail.

We shall present in subsequent articles code coverage and data-driven tools.

Adriana Ávila
Lucía Camilloni
Fernando Marotta
Diego Vallespir
Cecilia Apa

Software Engineering Group, UdelaR
gris@fing.edu.uy



Unit Testing in Java: Code Coverage with Clover

Combining different types of techniques makes it possible to increase effectiveness in the detection of defects in software.

A defective path in the source code, or even a defect in one statement, can cause an irreparable tragedy when it is executed. Knowing which parts of the code are executed by the software tests helps to prevent a problem when the system is in the production environment. Luckily, for some time now, there have been automated tools capable of providing information about the code that has been tested.

In the first part of this article, published in last year's issue of this journal, we presented basic concepts of unit testing and JUnit and TestNG drivers. In this second part we shall deal with code coverage at unit level and we shall present the Clover tool.
Combining different types of techniques makes it possible to increase the effectiveness in the detection of software defects. Personal reviews, inspections, black box and white box tests are combined to develop software products close to cero defect.

White box techniques use the source code structure to define the test cases. Different techniques focus on different structures of the code. For example, the statement coverage technique is based on obtaining a set of test cases capable of executing all the statements of the code under test. They use a measure called code coverage, which indicates the percentage of the code covered by the test. The code coverage depends on the specific technique being used. For example, statement coverage indicates the percentage of statements executed during the tests, and linearly independent paths indicate the percentage of paths covered of the total of paths that are linearly independent.

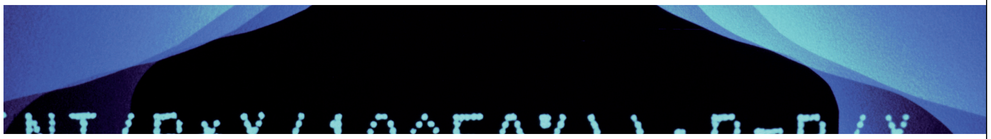
Trying to find out the cost-benefit relation of these techniques is a current topic of research. Therefore, the election of one of these techniques or the selection of a unit testing strategy is hardly trivial.

Coverage tool - Clover

Clover is a tool that measures statement and decision coverage achieved by a set of test cases. This tool can be used in an integrated way both with JUnit and TestNG.
It provides assistance in the development of software in order to be able to comply with the decision coverage during the tests. For example, it colors the statements to indicate the coverage obtained. At the same time, it generates reports that help determine at a global level the coverage obtained, making it possible to know quickly which of the classes that have not been tested in depth are more likely to present defects. The tool is easy to use and has a friendly graphic interface.

A simple example

The following example will be useful both to present the decision coverage using the Clover tool and to introduce the coverage of linearly independent paths and the Coview tool in the third part of this article.
The merge method is shown in Figure 1. This method receives two arrays of integer numbers ordered from smaller to larger. The method "takes" the elements of the two arrays and returns another array with those elements ordered from smaller to larger.



```
public int [] merge( int[ ] a, int [ ] b) {
    int i= 0;
    int j = 0;
    int k = 0;
    int c[] = new int[a.length + b.length];
    while( i < a.length && j < b.length ) {
        if( a[i] < b[j]) {
            c[k]=a[i];
            k++;
            i++;
        }
        else{
            c[k]=a[j];
            k++;
            j++;
        }
    }
    for (int iter=i; iter<a.length;iter++) {
        c[k]=a[iter];
        k++;
    }
    for (int iter=j; iter<b.length;iter++) {
        c[k]=b[iter];
        k++;
    }
    return c;
}
```

Figure 1 - Merge method source code

Figure 2-a presents a test case in which the elements of the first array are all smaller than those of the second array. Figure 2-b presents another test case in which the elements of the arrays passed must be interspersed. Both cases are codified in JUnit and make up the set of test cases of our example.

```
@org.junit.Test
public void testmerge1() {
    int a[]={1,2,3,4,5,6};
    int b[]={7,8,9,10,11};
    Operations oper= new Operations();
    int c[]=oper.merge(a,b);
    int expected[]={1,2,3,4,5,6,7,8,9,10,11};
    assertArrayEquals(expected,c);
}
```

Figure 2a -Test case testmerge1

```
@org.junit.Test
public void testmerge2() {
    int a[]={1,2,5,11};
    int b[]={3,4,8,10};
    Operations oper= new Operations();
    int c[] =oper.merge(a,b);
    int expected[]={1,2,3,4,5,8,10,11};
    assertArrayEquals(expected,c);
}
```

Figure 2b - Test case testmerge2

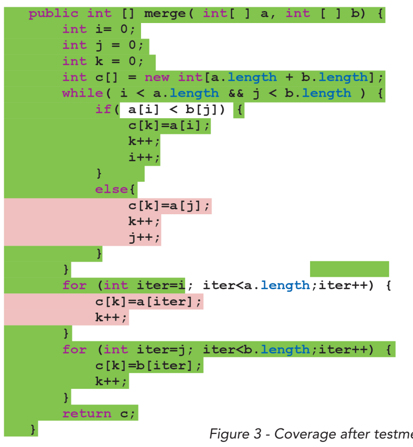
milveinticuatro

Execution using Clover

By executing only the test case testmerge1 75% decision coverage is obtained. In the merge method there are 4 decisions: a while, an if and 2 fors. Each one of these decisions can take either value false or value true. The testmerge1 case achieves in its execution that the while, as well as the second for take both the value true and false. However, the if and the first for take only true and false respectively. That is to say, 6 of the 8 possibilities of the 4 decisions are executed, and this represents 75%. Clover paints the executed statements to help visualize which code still needs to be covered. The green color indicates that the statement was executed and its result coincides with the expected one. The red color (pink, in fact) indicates that the statement has not been executed in the tests.
The decisions are painted in green when they have been executed with both their true values and are not painted in the opposite case. Those colors provide a quick visualization of the code which has not been covered by the tests.

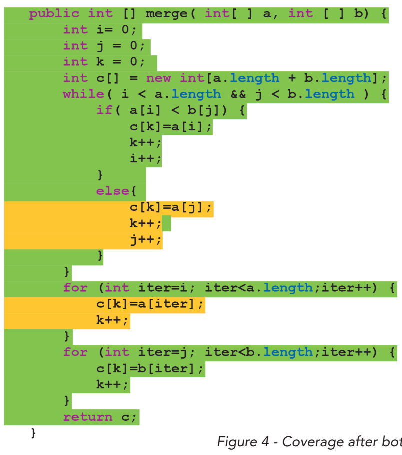
Figure 3 presents the code painted by Clover after executing the test case testmerge1. It is easy to see the statements that were not executed and the decisions that were only covered partially (if and first for).
Carrying on with the example, we execute the second test case: testmerge2. Thus we attempt to satisfy the decision criterion. When the second test case is executed using Clover certain lines are colored in yellow. Clover uses that color to show statements that were only executed by test cases that fail. That is to say, test cases whose expected results are different from the actual results. Anyway, it is always convenient to observe the JUnit or TestNG results before corroborating the code coverage; only when test cases do not fail it is interesting to observe the coverage. The code after executing both test cases is presented in Figure 4.

The defect that causes the failure in the test case testmerge2 is in the statement following the else. The statement c[k] = a[j]; must be changed for c[k] = b[j];
After correcting this defect the test cases are run again and all the code is painted green. The code has been covered in 100% of the decision criterion!



```
public int [] merge( int[ ] a, int [ ] b) {
     int i= 0;
     int j = 0;
     int k = 0;
     int c[] = new int[a.length + b.length];
     while( i < a.length && j < b.length ) {
          if( a[i] < b[j]) {
               c[k]=a[i];
               k++;
               i++;
          }
          else{
               c[k]=a[j];
               k++;
               j++;
          }
     }
     for (int iter=i; iter<a.length;iter++) {
          c[k]=a[iter];
          k++;
     }
     for (int iter=j; iter<b.length;iter++) {
          c[k]=b[iter];
          k++;
     }
     return c;
 }
```

Figure 3 - Coverage after testmerge1



```
public int [] merge( int[ ] a, int [ ] b) {
     int i= 0;
     int j = 0;
     int k = 0;
     int c[] = new int[a.length + b.length];
     while( i < a.length && j < b.length ) {
          if( a[i] < b[j]) {
               c[k]=a[i];
               k++;
               i++;
          }
          else{
               c[k]=a[j];
               k++;
               j++;
          }
     }
     for (int iter=i; iter<a.length;iter++) {
          c[k]=a[iter];
          k++;
     }
     for (int iter=j; iter<b.length;iter++) {
          c[k]=b[iter];
          k++;
     }
     return c;
 }
```

Figure 4 - Coverage after both cases

1525

Good use of the code coverage

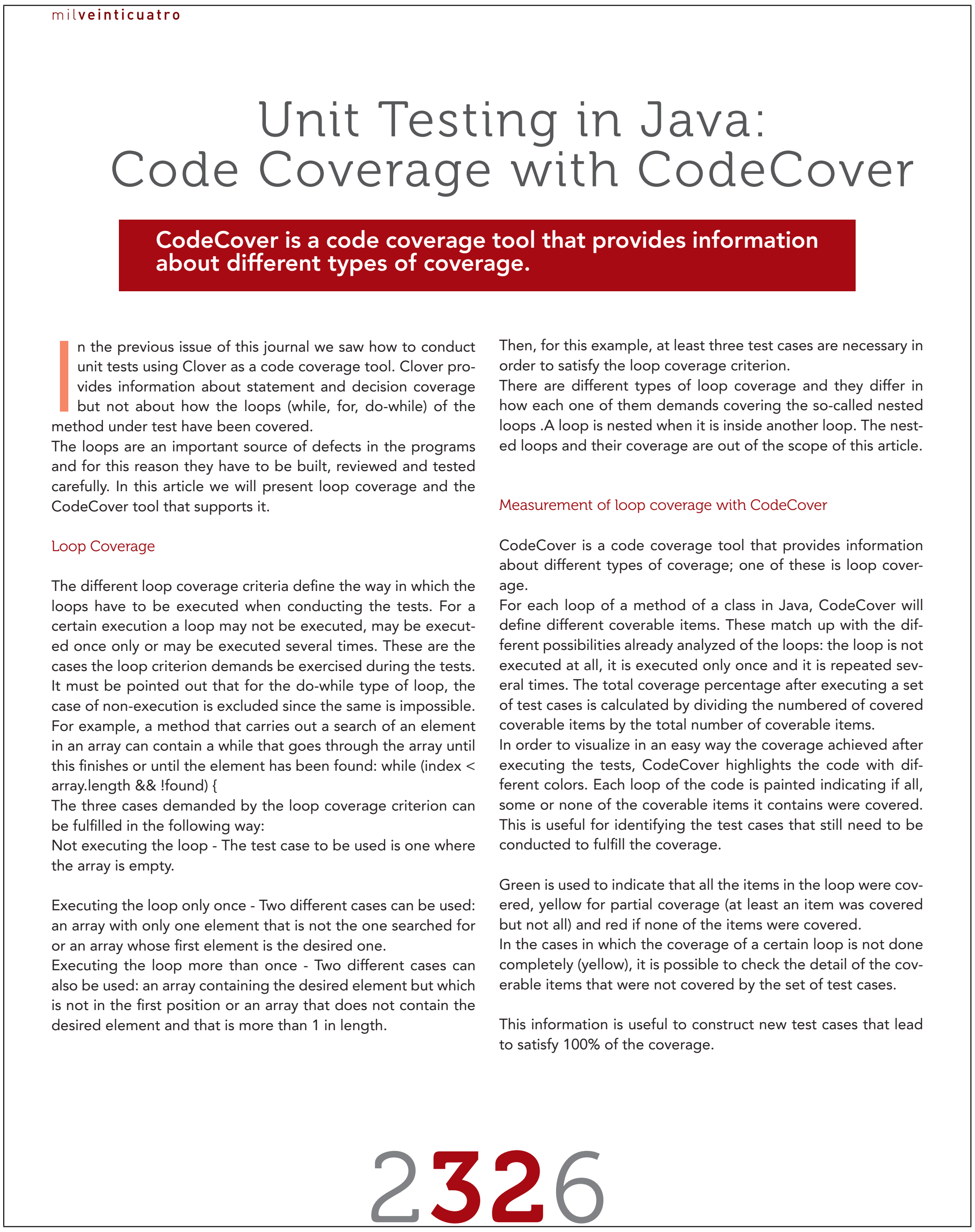
The code coverage gives us a good idea of the portions of code executed. However, it is not reasonable to rely only on these tests. A reasonable strategy is the application of personal reviews before starting the tests; this way, the quality of the product will increase.

It is also advisable to think first about black box tests that cover the different features of the unit, class or method being tested. After conducting these tests and correcting the defects, it is possible to examine the code coverage.

If the code has not been covered as it was defined in the strategy, new test cases will be generated in order to achieve that coverage. This unit verification strategy (personal reviews, black box tests, white box tests) will provide a better quality product than a strategy only based on one type of technique.

Adriana Ávila
Lucía Camilloni
Diego Vallespir
Cecilia Apa

Software Engineering Group, UdelaR
gris@fing.edu.uy



Unit Testing in Java: Code Coverage with CodeCover

CodeCover is a code coverage tool that provides information about different types of coverage.

In the previous issue of this journal we saw how to conduct unit tests using Clover as a code coverage tool. Clover provides information about statement and decision coverage but not about how the loops (while, for, do-while) of the method under test have been covered.
The loops are an important source of defects in the programs and for this reason they have to be built, reviewed and tested carefully. In this article we will present loop coverage and the CodeCover tool that supports it.

Loop Coverage

The different loop coverage criteria define the way in which the loops have to be executed when conducting the tests. For a certain execution a loop may not be executed, may be executed once only or may be executed several times. These are the cases the loop criterion demands be exercised during the tests. It must be pointed out that for the do-while type of loop, the case of non-execution is excluded since the same is impossible.
For example, a method that carries out a search of an element in an array can contain a while that goes through the array until this finishes or until the element has been found: while (index < array.length && !found) {
The three cases demanded by the loop coverage criterion can be fulfilled in the following way:
Not executing the loop - The test case to be used is one where the array is empty.

Executing the loop only once - Two different cases can be used: an array with only one element that is not the one searched for or an array whose first element is the desired one.
Executing the loop more than once - Two different cases can also be used: an array containing the desired element but which is not in the first position or an array that does not contain the desired element and that is more than 1 in length.

Then, for this example, at least three test cases are necessary in order to satisfy the loop coverage criterion.
There are different types of loop coverage and they differ in how each one of them demands covering the so-called nested loops .A loop is nested when it is inside another loop. The nested loops and their coverage are out of the scope of this article.

Measurement of loop coverage with CodeCover

CodeCover is a code coverage tool that provides information about different types of coverage; one of these is loop coverage.
For each loop of a method of a class in Java, CodeCover will define different coverable items. These match up with the different possibilities already analyzed of the loops: the loop is not executed at all, it is executed only once and it is repeated several times. The total coverage percentage after executing a set of test cases is calculated by dividing the numbered of covered coverable items by the total number of coverable items.
In order to visualize in an easy way the coverage achieved after executing the tests, CodeCover highlights the code with different colors. Each loop of the code is painted indicating if all, some or none of the coverable items it contains were covered. This is useful for identifying the test cases that still need to be conducted to fulfill the coverage.

Green is used to indicate that all the items in the loop were covered, yellow for partial coverage (at least an item was covered but not all) and red if none of the items were covered.
In the cases in which the coverage of a certain loop is not done completely (yellow), it is possible to check the detail of the coverable items that were not covered by the set of test cases.

This information is useful to construct new test cases that lead to satisfy 100% of the coverage.

An easy example

In Figure 1 the merge method is presented; the same method used as an example in the previous article (taking the corrected version as a starting point). It receives two arrays of integer numbers ordered from smaller to larger. The method “takes” the elements of the two arrays and comes up with another array with those elements arranged from smaller to larger.

Execution using CodeCover

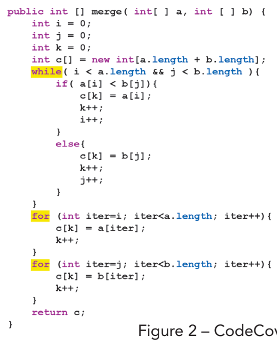
The same test cases presented in the previous article are used with CodeCover. Figure 3.a presents a test case in which the elements of the first array are all smaller than those of the second array. Figure 3.b presents another test case in which the elements of the two arrays must be interspersed. Both cases are codified in JUnit and make up the set of test cases of our example.
The merge method contains 3 loops (1 while and 2 for) that determine 9 coverable items. We will refer to the coverable items as Zero, One and Many for the cases in which the loop is not executed, is executed once and is executed several times respectively.

When executing the two test cases 55.6% coverage is obtained. This corresponds to the execution of 5 of the 9 items.

In figure 2 the highlighted code that corresponds to the execution of these cases using CodeCover is shown. It can be observed that the loops (while and for) within the method are painted in yellow; this indicated that all the loops have only a partial coverage of the coverable items.

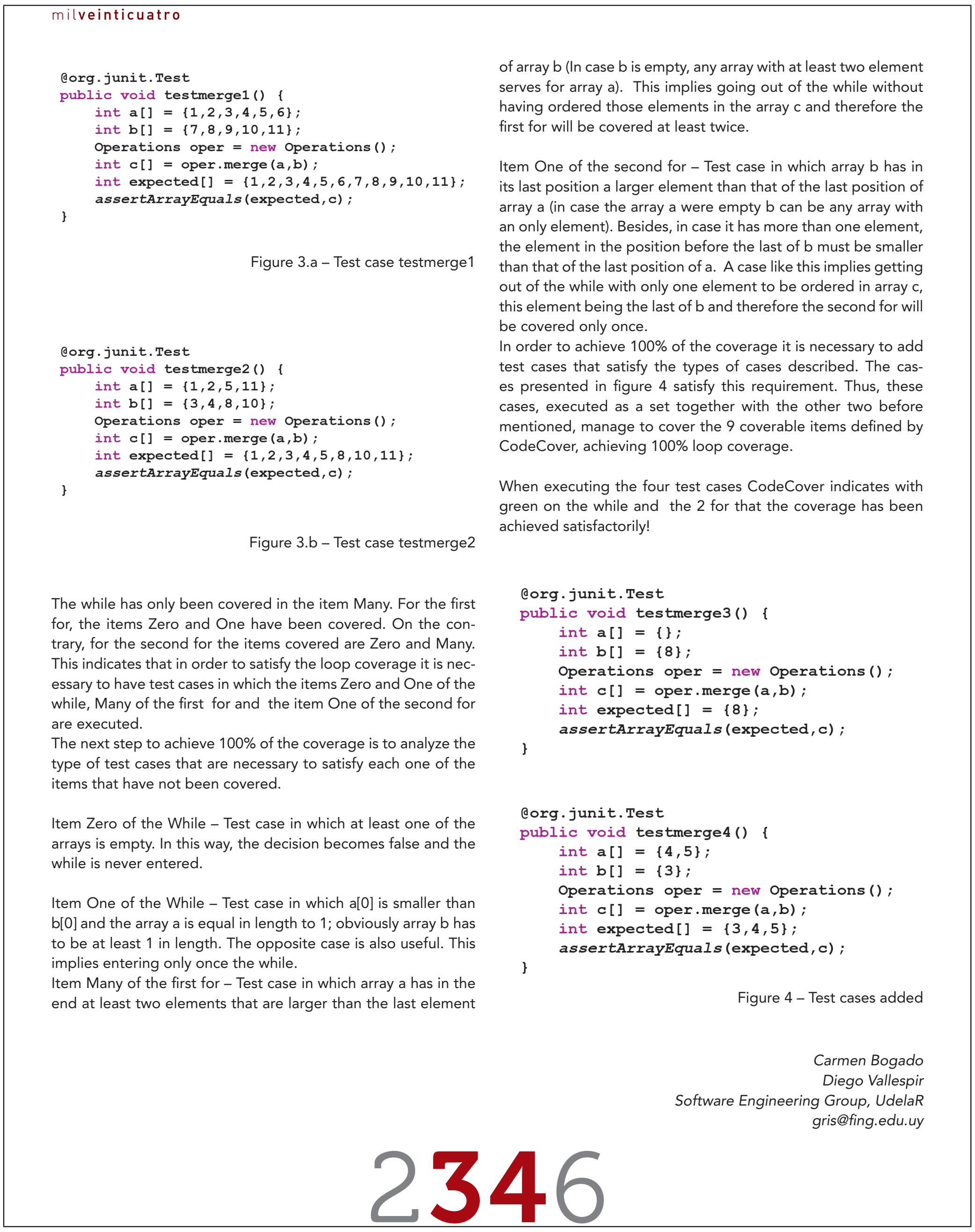
```
public int [] merge( int[ ] a, int [ ] b) {
    int i = 0;
    int j = 0;
    int k = 0;
    int c[] = new int[a.length + b.length];
    while( i < a.length && j < b.length ){
        if( a[i] < b[j]){
            c[k] = a[i];
            k++;
            i++;
        }
        else{
            c[k] = b[j];
            k++;
            j++;
        }
    }
    for (int iter=i; iter<a.length; iter++){
        c[k] = a[iter];
        k++;
    }
    for (int iter=j; iter<b.length; iter++){
        c[k] = b[iter];
        k++;
    }
    return c;
}
```

Figure 1 – Merge method



```
public int [] merge( int[ ] a, int [ ] b) {
    int i = 0;
    int j = 0;
    int k = 0;
    int c[] = new int[a.length + b.length];
    while( i < a.length && j < b.length ){
        if( a[i] < b[j]){
            c[k] = a[i];
            k++;
            i++;
        }
        else{
            c[k] = b[j];
            k++;
            j++;
        }
    }
    for (int iter=i; iter<a.length; iter++){
        c[k] = a[iter];
        k++;
    }
    for (int iter=j; iter<b.length; iter++){
        c[k] = b[iter];
        k++;
    }
    return c;
}
```

Figure 2 – CodeCover Coverage



```
@org.junit.Test
public void testmerge1() {
    int a[] = {1,2,3,4,5,6};
    int b[] = {7,8,9,10,11};
    Operations oper = new Operations();
    int c[] = oper.merge(a,b);
    int expected[] = {1,2,3,4,5,6,7,8,9,10,11};
    assertArrayEquals(expected,c);
}
```

Figure 3.a – Test case testmerge1

```
@org.junit.Test
public void testmerge2() {
    int a[] = {1,2,5,11};
    int b[] = {3,4,8,10};
    Operations oper = new Operations();
    int c[] = oper.merge(a,b);
    int expected[] = {1,2,3,4,5,8,10,11};
    assertArrayEquals(expected,c);
}
```

Figure 3.b – Test case testmerge2

The while has only been covered in the item Many. For the first for, the items Zero and One have been covered. On the contrary, for the second for the items covered are Zero and Many. This indicates that in order to satisfy the loop coverage it is necessary to have test cases in which the items Zero and One of the while, Many of the first for and the item One of the second for are executed.
The next step to achieve 100% of the coverage is to analyze the type of test cases that are necessary to satisfy each one of the items that have not been covered.

Item Zero of the While – Test case in which at least one of the arrays is empty. In this way, the decision becomes false and the while is never entered.

Item One of the While – Test case in which a[0] is smaller than b[0] and the array a is equal in length to 1; obviously array b has to be at least 1 in length. The opposite case is also useful. This implies entering only once the while.
Item Many of the first for – Test case in which array a has in the end at least two elements that are larger than the last element of array b (In case b is empty, any array with at least two element serves for array a). This implies going out of the while without having ordered those elements in the array c and therefore the first for will be covered at least twice.

Item One of the second for – Test case in which array b has in its last position a larger element than that of the last position of array a (in case the array a were empty b can be any array with an only element). Besides, in case it has more than one element, the element in the position before the last of b must be smaller than that of the last position of a. A case like this implies getting out of the while with only one element to be ordered in array c, this element being the last of b and therefore the second for will be covered only once.
In order to achieve 100% of the coverage it is necessary to add test cases that satisfy the types of cases described. The cases presented in figure 4 satisfy this requirement. Thus, these cases, executed as a set together with the other two before mentioned, manage to cover the 9 coverable items defined by CodeCover, achieving 100% loop coverage.

When executing the four test cases CodeCover indicates with green on the while and the 2 for that the coverage has been achieved satisfactorily!

```
@org.junit.Test
public void testmerge3() {
    int a[] = {};
    int b[] = {8};
    Operations oper = new Operations();
    int c[] = oper.merge(a,b);
    int expected[] = {8};
    assertArrayEquals(expected,c);
}

@org.junit.Test
public void testmerge4() {
    int a[] = {4,5};
    int b[] = {3};
    Operations oper = new Operations();
    int c[] = oper.merge(a,b);
    int expected[] = {3,4,5};
    assertArrayEquals(expected,c);
}
```

Figure 4 – Test cases added

Carmen Bogado
Diego Vallespir
Software Engineering Group, UdelaR
gris@fing.edu.uy

Unit Testing in Java: Code Coverage with CoView

CoView determines the linearly independent paths and presents them in the development environment coloring the code.

In the previous article we used the CoCover tool to know the coverage achieved by a set of test cases at cycle level in the code. This coverage does not provide information about the paths of the code that could be covered when executing the tests.

In this article we present the coverage of Linearly Independent Paths (LIP) without dealing with theoretical details, and the CoView tool, which presents visually which of these paths were executed during the tests.

Linearly Independent Paths

A path represents an execution flow from the beginning of a method (or program) till its end. The possible paths depend on the existing bifurcations in the code (if, while, for, etc.) that direct the execution flow in "one direction or another". A method with N decisions (if, for instance) and without cycles (loops) has potentially 2^N paths. If it also has cycles, the paths could be infinite. From the point of view of the code coverage it is normally impossible to cover this number of paths, or it is too expensive. A way of limiting the number of paths to cover during the tests is by using the LIP criterion. The LIP set is the set with the least number of paths that can generate any other path of the method by means of a linear combination of the same (this derives from linear algebra). It is for this reason that the criterion is interesting from the point of view of unit testing. The formal definition of the LIP set is not covered in this article.

A way of knowing the number of LIP of a method is by counting the number of "zones" in which the flow control graph is divided (graph that indicates the possible paths of a method). This number is called cyclomatic complexity of a method.

A simple example

We use the merge method presented in the previous articles as an example. In Figure 1 the flow control graph of this method is presented and the "zones" in which it is divided are listed; this indicates that the method has a cyclomatic complexity equal to 6. The zone numbered as 6 is the external one.

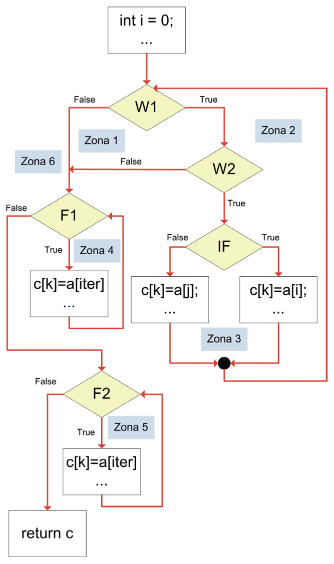
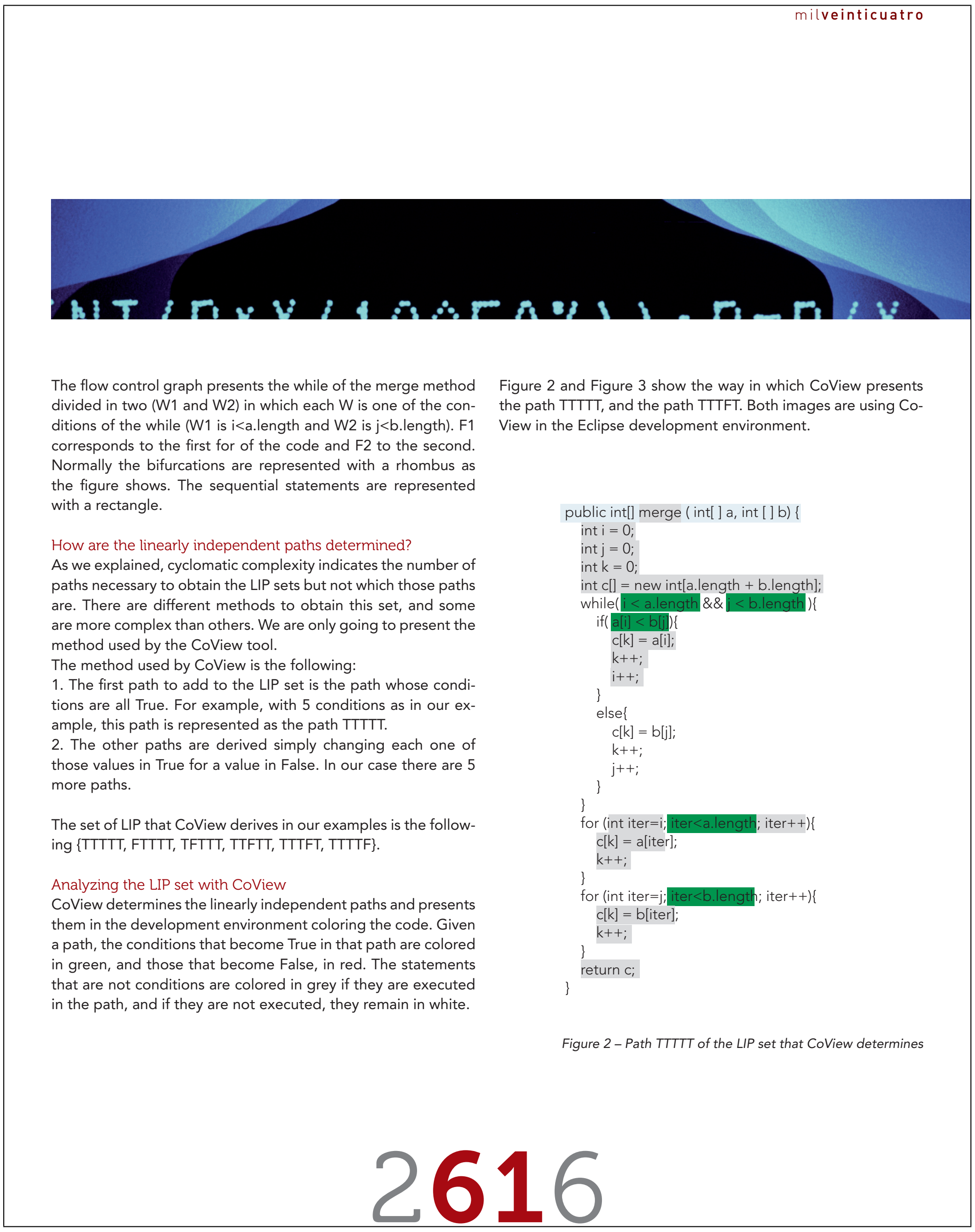


Figure 1 – Flow control graph of the merge method



milveinticuatro

The flow control graph presents the while of the merge method divided in two (W1 and W2) in which each W is one of the conditions of the while (W1 is i<a.length and W2 is j<b.length). F1 corresponds to the first for of the code and F2 to the second. Normally the bifurcations are represented with a rhombus as the figure shows. The sequential statements are represented with a rectangle.

How are the linearly independent paths determined?

As we explained, cyclomatic complexity indicates the number of paths necessary to obtain the LIP sets but not which those paths are. There are different methods to obtain this set, and some are more complex than others. We are only going to present the method used by the CoView tool.

The method used by CoView is the following:

1. The first path to add to the LIP set is the path whose conditions are all True. For example, with 5 conditions as in our example, this path is represented as the path TTTTT.

2. The other paths are derived simply changing each one of those values in True for a value in False. In our case there are 5 more paths.

The set of LIP that CoView derives in our examples is the following {TTTTT, FTTTT, TFTTT, TTFTT, TTTFT, TTTTF}.

Analyzing the LIP set with CoView

CoView determines the linearly independent paths and presents them in the development environment coloring the code. Given a path, the conditions that become True in that path are colored in green, and those that become False, in red. The statements that are not conditions are colored in grey if they are executed in the path, and if they are not executed, they remain in white.

Figure 2 and Figure 3 show the way in which CoView presents the path TTTTT, and the path TTTFT. Both images are using CoView in the Eclipse development environment.

```
public int[] merge ( int[ ] a, int [ ] b) {
   int i = 0;
   int j = 0;
   int k = 0;
   int c[] = new int[a.length + b.length];
   while( i < a.length && j < b.length ){
      if( a[i] < b[j]){
         c[k] = a[i];
         k++;
         i++;
      }
      else{
         c[k] = b[j];
         k++;
         j++;
      }
   }
   for (int iter=i; iter<a.length; iter++){
      c[k] = a[iter];
      k++;
   }
   for (int iter=j; iter<b.length; iter++){
      c[k] = b[iter];
      k++;
   }
   return c;
}
```

Figure 2 – Path TTTTT of the LIP set that CoView determines

2616

```
public int[] merge ( int[ ] a, int [ ] b) {
    int i = 0;
    int j = 0;
    int k = 0;
    int c[] = new int[a.length + b.length];
    while( i < a.length && j < b.length ){
        if( a[i] < b[j] ){
            c[k] = a[i];
            k++;
            i++;
        }
        else{
            c[k] = b[j];
            k++;
            j++;
        }
    }
    for (int iter=i; iter<a.length; iter++){
        c[k] = a[iter];
        k++;
    }
    for (int iter=j; iter<b.length; iter++){
        c[k] = b[iter];
        k++;
    }
    return c;
}
```

Figure 3 – Path TTTFT of the LIP set that CoView determines

Execution using CoView

We use the same four test cases we used in the previous article in order to test the merge method.

When executing these test cases CoView indicates that only 1 of the 6 paths of the LIP set have been executed. Given this result, and aiming at achieving 100% coverage, it is necessary to generate test cases so as to cover the paths of the LIP set that have not been covered.

In order to generate these test cases we analyzed the 6 paths generated by CoView. This analysis shows that several of these paths are impossible to execute. That is to say, that no test case can cover the flow control graph during its execution as indicated by the path.

Let us take the path of Figure 2 as an example (TTTTT). This path indicates that it is enter the while with i<a.length and with j<b.length (the path indicates the while becomes true). Then, to cover the path, a[i] must be smaller than b[j] (the if becomes true). When returning to the decision of the while it is not necessary to execute the body, and that is why one of the two conditions is false; then, either i is not smaller than a.length or j is not smaller than b.length. This indicates that at least one of the two fors of the end will be impossible to execute. In this way it is shown that the path TTTTT cannot be executed with any test case since the fact that the two fors execute their body will never occur.

Analyzing non-executable paths

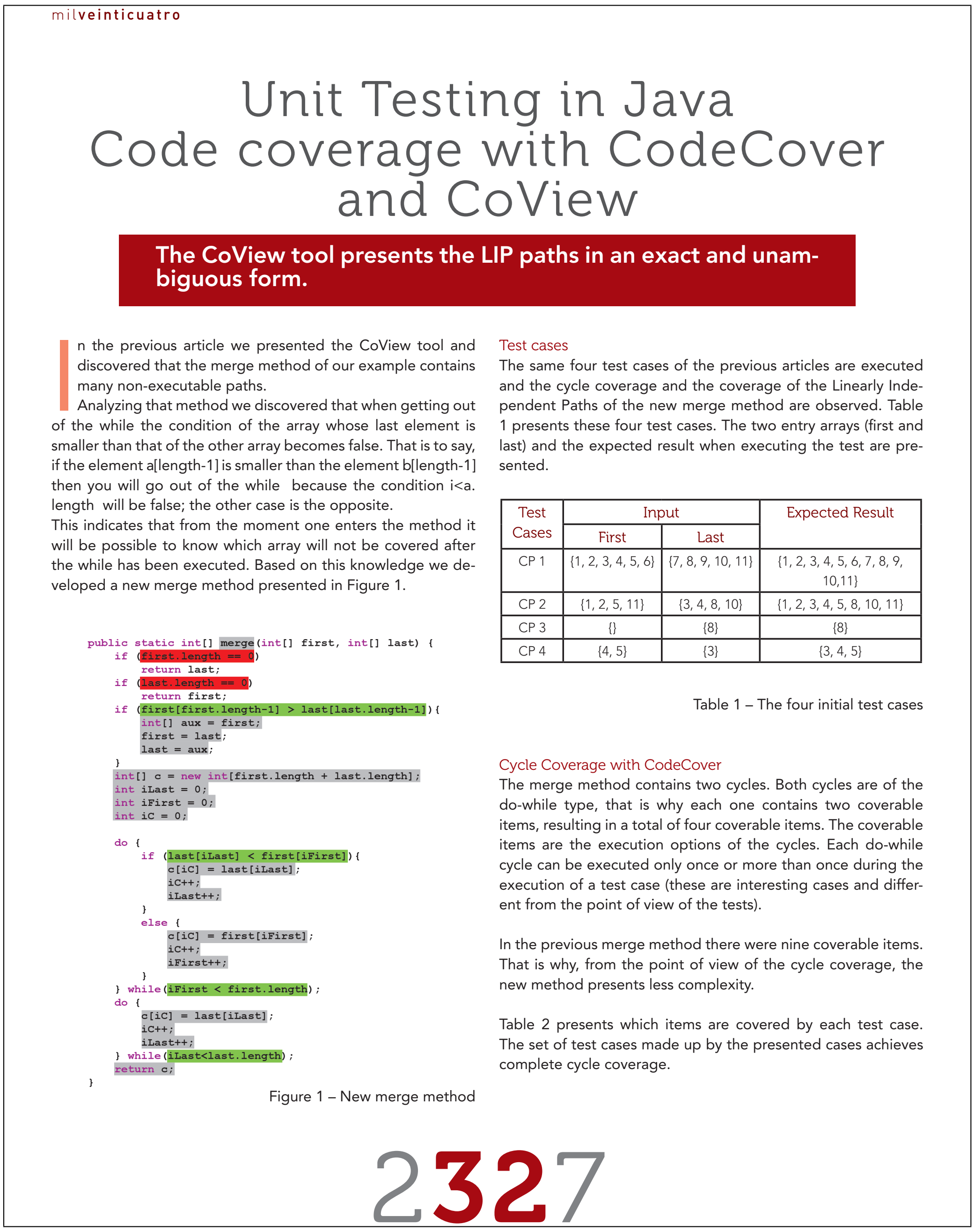
Since the set of LIP determined by CoView contains non-executable paths, it will not be possible to achieve 100% coverage. However, not satisfying the coverage becomes a secondary concern. Why are there non-executable paths in the code? Is there any way this can be solved?

In our next article we will present a new algorithm for merge that solves the problem of having so many non-executable paths and we will study the coverage achieved with different criteria, including de criterion LIP.

In this article we saw how using the LIP coverage we managed to detect that the merge method has many non-executable paths. Discovering non-executable paths is a good reason to think of a new algorithmic solution. To conclude, we must clarify two things. Firstly, it is not necessary to use the LIP criterion to analyze the non-executable paths of an algorithm; it can be done simply analyzing the flow control graph directly. Secondly, it is not always possible to get an algorithm without non-executable paths.

Diego Vallespir
Fernando Marotta

Software Engineering Group, UdelaR
gris@fing.edu.uy



Unit Testing in Java Code coverage with CodeCover and CoView

The CoView tool presents the LIP paths in an exact and unambiguous form.

In the previous article we presented the CoView tool and discovered that the merge method of our example contains many non-executable paths.

Analyzing that method we discovered that when getting out of the while the condition of the array whose last element is smaller than that of the other array becomes false. That is to say, if the element a[length-1] is smaller than the element b[length-1] then you will go out of the while because the condition i<a.length will be false; the other case is the opposite.

This indicates that from the moment one enters the method it will be possible to know which array will not be covered after the while has been executed. Based on this knowledge we developed a new merge method presented in Figure 1.

```java
public static int[] merge(int[] first, int[] last) {
    if (first.length == 0)
        return last;
    if (last.length == 0)
        return first;
    if (first[first.length-1] > last[last.length-1]){
        int[] aux = first;
        first = last;
        last = aux;
    }
    int[] c = new int[first.length + last.length];
    int iLast = 0;
    int iFirst = 0;
    int iC = 0;

    do {
        if (last[iLast] < first[iFirst]){
            c[iC] = last[iLast];
            iC++;
            iLast++;
        }
        else {
            c[iC] = first[iFirst];
            iC++;
            iFirst++;
        }
    } while(iFirst < first.length);
    do {
        c[iC] = last[iLast];
        iC++;
        iLast++;
    } while(iLast<last.length);
    return c;
}
```

Figure 1 – New merge method

Test cases

The same four test cases of the previous articles are executed and the cycle coverage and the coverage of the Linearly Independent Paths of the new merge method are observed. Table 1 presents these four test cases. The two entry arrays (first and last) and the expected result when executing the test are presented.

Test Cases	Input		Expected Result
	First	Last	
CP 1	{1, 2, 3, 4, 5, 6}	{7, 8, 9, 10, 11}	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11}
CP 2	{1, 2, 5, 11}	{3, 4, 8, 10}	{1, 2, 3, 4, 5, 8, 10, 11}
CP 3	{}	{8}	{8}
CP 4	{4, 5}	{3}	{3, 4, 5}

Table 1 – The four initial test cases

Cycle Coverage with CodeCover

The merge method contains two cycles. Both cycles are of the do-while type, that is why each one contains two coverable items, resulting in a total of four coverable items. The coverable items are the execution options of the cycles. Each do-while cycle can be executed only once or more than once during the execution of a test case (these are interesting cases and different from the point of view of the tests).

In the previous merge method there were nine coverable items. That is why, from the point of view of the cycle coverage, the new method presents less complexity.

Table 2 presents which items are covered by each test case. The set of test cases made up by the presented cases achieves complete cycle coverage.

	First do-while of the method		Second do-while of the method	
Test cases	Item One	Item Many	Item One	Item Many
CP 1	X	Covers	X	Covers
CP 2	X	Covers	Covers	X
CP 3	X	X	X	X
CP 4	Covers	X	X	Cubre
Total Set of Test Cases	**Covers**	**Covers**	**Covers**	**Covers**

Table 2 – Cycle coverage with the 4 initial cases

Linearly Independent Paths with CoView

Figure 2 presents the control flow graph of the merge method and the 7 zones into which it is divided. The number of zones is the cyclomatic number and therefore the number of Linearly Independent Paths (LIP).

The CoView tool generates a set of LIP automatically. Each bifurcation of the code (if, while, do-while, etc.) can take the values True or False. These are the bifurcations that determine each path of the code.

The merge method has 6 bifurcations; 3 if at the beginning, a first do-while, an if inside the first do-while and another do-while at the end. The paths will be represented with the values (True or False) these bifurcations take during its execution.
The paths determined by the tool are the following:

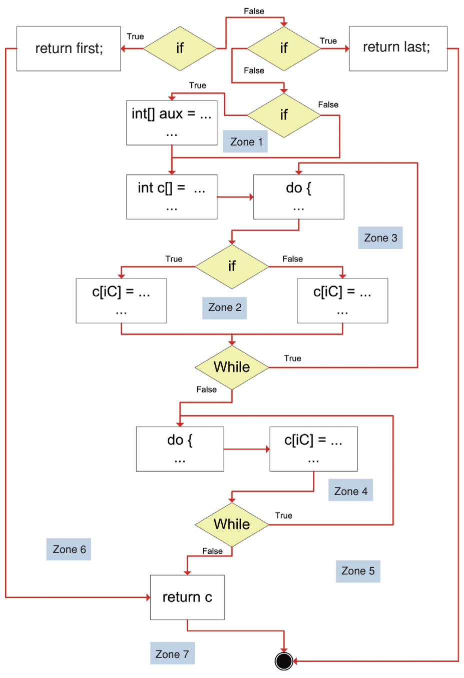
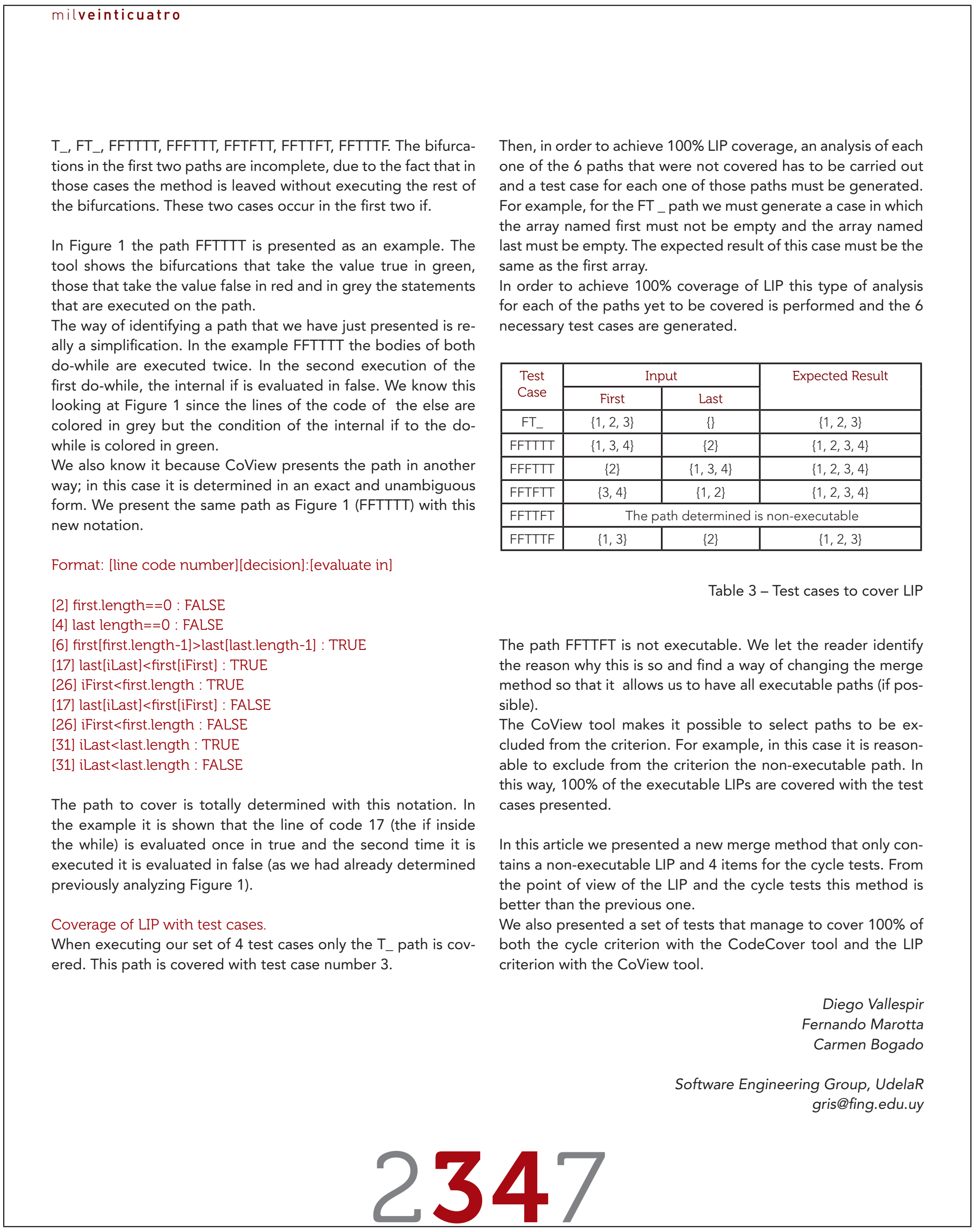


Figure 2 – Control flow graph of the merge method



T_, FT_, FFTTTT, FFFTTT, FFTFTT, FFTTFT, FFTTTF. The bifurcations in the first two paths are incomplete, due to the fact that in those cases the method is leaved without executing the rest of the bifurcations. These two cases occur in the first two if.

In Figure 1 the path FFTTTT is presented as an example. The tool shows the bifurcations that take the value true in green, those that take the value false in red and in grey the statements that are executed on the path.
The way of identifying a path that we have just presented is really a simplification. In the example FFTTTT the bodies of both do-while are executed twice. In the second execution of the first do-while, the internal if is evaluated in false. We know this looking at Figure 1 since the lines of the code of the else are colored in grey but the condition of the internal if to the do-while is colored in green.
We also know it because CoView presents the path in another way; in this case it is determined in an exact and unambiguous form. We present the same path as Figure 1 (FFTTTT) with this new notation.

Format: [line code number][decision]:[evaluate in]

```
[2] first.length==0 : FALSE
[4] last length==0 : FALSE
[6] first[first.length-1]>last[last.length-1] : TRUE
[17] last[iLast]<first[iFirst] : TRUE
[26] iFirst<first.length : TRUE
[17] last[iLast]<first[iFirst] : FALSE
[26] iFirst<first.length : FALSE
[31] iLast<last.length : TRUE
[31] iLast<last.length : FALSE
```

The path to cover is totally determined with this notation. In the example it is shown that the line of code 17 (the if inside the while) is evaluated once in true and the second time it is executed it is evaluated in false (as we had already determined previously analyzing Figure 1).

Coverage of LIP with test cases.
When executing our set of 4 test cases only the T_ path is covered. This path is covered with test case number 3.

Then, in order to achieve 100% LIP coverage, an analysis of each one of the 6 paths that were not covered has to be carried out and a test case for each one of those paths must be generated. For example, for the FT _ path we must generate a case in which the array named first must not be empty and the array named last must be empty. The expected result of this case must be the same as the first array.
In order to achieve 100% coverage of LIP this type of analysis for each of the paths yet to be covered is performed and the 6 necessary test cases are generated.

Test Case	Input		Expected Result
	First	Last	
FT_	{1, 2, 3}	{}	{1, 2, 3}
FFTTTT	{1, 3, 4}	{2}	{1, 2, 3, 4}
FFFTTT	{2}	{1, 3, 4}	{1, 2, 3, 4}
FFTFTT	{3, 4}	{1, 2}	{1, 2, 3, 4}
FFTTFT	The path determined is non-executable		
FFTTTF	{1, 3}	{2}	{1, 2, 3}

Table 3 – Test cases to cover LIP

The path FFTTFT is not executable. We let the reader identify the reason why this is so and find a way of changing the merge method so that it allows us to have all executable paths (if possible).
The CoView tool makes it possible to select paths to be excluded from the criterion. For example, in this case it is reasonable to exclude from the criterion the non-executable path. In this way, 100% of the executable LIPs are covered with the test cases presented.

In this article we presented a new merge method that only contains a non-executable LIP and 4 items for the cycle tests. From the point of view of the LIP and the cycle tests this method is better than the previous one.
We also presented a set of tests that manage to cover 100% of both the cycle criterion with the CodeCover tool and the LIP criterion with the CoView tool.

Diego Vallespir
Fernando Marotta
Carmen Bogado

Software Engineering Group, UdelaR
gris@fing.edu.uy

Unit Testing in Java Data-Driven with TestNG

The approach of the tests based on Data-Driven aims at the separation of the logic of the test from its scenario.

While conducting unit tests it is common to work with similar test cases where the only difference between them is the test data used. In this article we present the approach based on *Data-Driven*; this approach allows us to separate the logic of the test from its data.
In the previous article we presented a new version of the merge method and a test set with 9 test cases. These test cases are the example in this article.

Data-Driven approach.

The approach of the tests based on *Data-Driven* aims at the separation of the logic of the test from its scenario. Thus it makes it possible to use the same logic with different sets of test cases (input data and expected results). These data are kept (and then when the tests are executed they are obtained) from a source of external data such as an XML file or an electronic spreadsheet.
At the moment when the tests that use *Data-Driven* are going to be executed, the first data of the data file is taken, the test is conducted with that data and the obtained results are verified. The same is done successively for all the items in the file.

TestNG tool

The TestNG tool is a software framework for unit and integration testing in Java. It was created based on JUnit and NUnit (for .NET), but introducing new functionality that try to make it more powerful and easy to use.

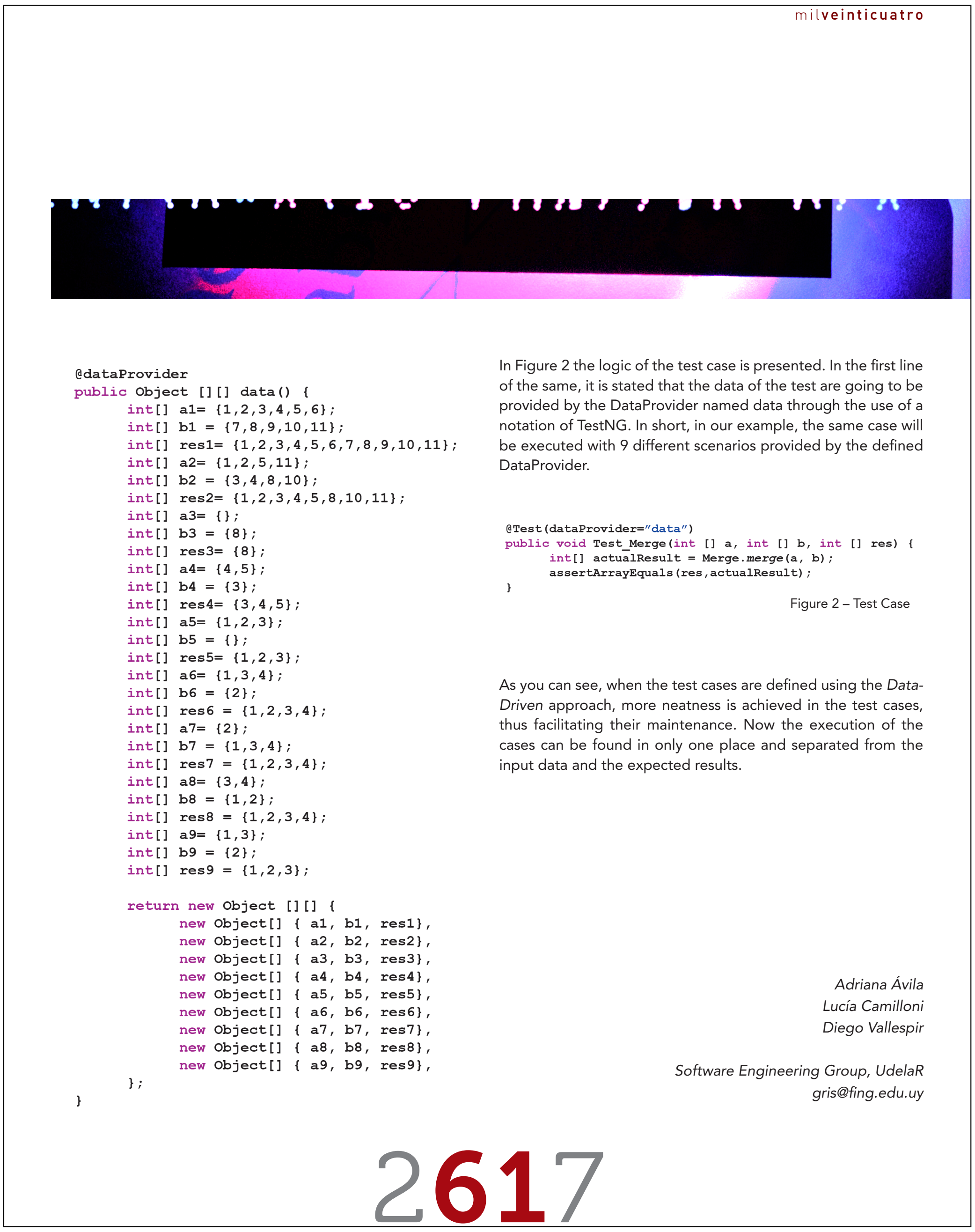
One of its functions is to provide software for tests that use *Data-Driven*. The tool requires that a method that will be the container of the test data be defined. It is not possible to input the test data directly from an external data source. Although this limits the tool, it is relatively simple to create a method that translates the data of an external file to the method required by TestNG.

Using the Data-Driven approach with the merge method

The use of this approach with the TestNG tool for the *merge* method is presented below. This method receives two arrays of integers (each one arranged from smaller to larger) and returns an array that is the ordered union of the elements of both arrays. The signature of the method is the following:

```
public int[] merge( int [ ] a, int [ ] b)
```

Using the notation *@dataProvider* it is pointed out that a method is a data supplier for a test case. The method must return an *Object[ ][ ]* where each *Object[ ]* can be assigned to a list of parameters of the test case.
The data container for the 9 test cases defined in the previous article is shown in Figure 1. The *9 Object[]* created at the end of the method as part of the array of arrays of objects are the 9 input data and expected results for the test cases.



```
@dataProvider
public Object [][] data() {
	int[] a1= {1,2,3,4,5,6};
	int[] b1 = {7,8,9,10,11};
	int[] res1= {1,2,3,4,5,6,7,8,9,10,11};
	int[] a2= {1,2,5,11};
	int[] b2 = {3,4,8,10};
	int[] res2= {1,2,3,4,5,8,10,11};
	int[] a3= {};
	int[] b3 = {8};
	int[] res3= {8};
	int[] a4= {4,5};
	int[] b4 = {3};
	int[] res4= {3,4,5};
	int[] a5= {1,2,3};
	int[] b5 = {};
	int[] res5= {1,2,3};
	int[] a6= {1,3,4};
	int[] b6 = {2};
	int[] res6 = {1,2,3,4};
	int[] a7= {2};
	int[] b7 = {1,3,4};
	int[] res7 = {1,2,3,4};
	int[] a8= {3,4};
	int[] b8 = {1,2};
	int[] res8 = {1,2,3,4};
	int[] a9= {1,3};
	int[] b9 = {2};
	int[] res9 = {1,2,3};

	return new Object [][] {
		new Object[] { a1, b1, res1},
		new Object[] { a2, b2, res2},
		new Object[] { a3, b3, res3},
		new Object[] { a4, b4, res4},
		new Object[] { a5, b5, res5},
		new Object[] { a6, b6, res6},
		new Object[] { a7, b7, res7},
		new Object[] { a8, b8, res8},
		new Object[] { a9, b9, res9},
	};
}
```

In Figure 2 the logic of the test case is presented. In the first line of the same, it is stated that the data of the test are going to be provided by the DataProvider named data through the use of a notation of TestNG. In short, in our example, the same case will be executed with 9 different scenarios provided by the defined DataProvider.

```
@Test(dataProvider="data")
public void Test_Merge(int [] a, int [] b, int [] res) {
	int[] actualResult = Merge.merge(a, b);
	assertArrayEquals(res,actualResult);
}
```

Figure 2 – Test Case

As you can see, when the test cases are defined using the *Data-Driven* approach, more neatness is achieved in the test cases, thus facilitating their maintenance. Now the execution of the cases can be found in only one place and separated from the input data and the expected results.

Adriana Ávila
Lucía Camilloni
Diego Vallespir

Software Engineering Group, UdelaR
gris@fing.edu.uy

Appendix B

Conceptos de Ingeniería de Software Empírica

En este Apéndice se incluye un Reporte Técnico que introduce a la ingeniería de software empírica.

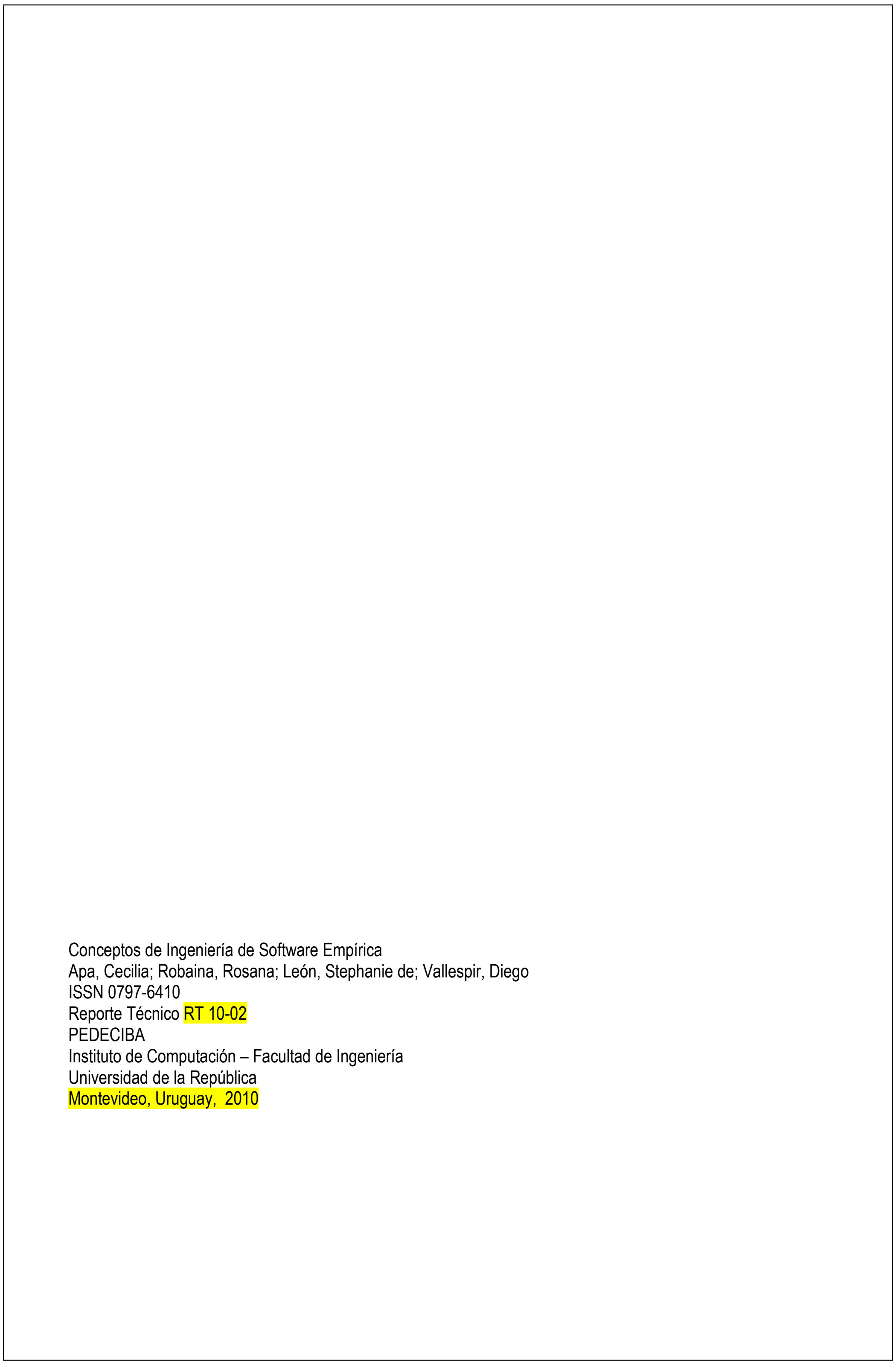
PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 10-02

Conceptos de Ingeniería de Software Empírica

Cecilia Apa, Rosana Robaina, Stephanie de León, Diego Vallespir

2010



Conceptos de Ingeniería de Software Empírica
Apa, Cecilia; Robaina, Rosana; León, Stephanie de; Vallespir, Diego
ISSN 0797-6410
Reporte Técnico RT 10-02
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, 2010

Conceptos de Ingeniería de Software Empírica

Cecilia Apa, Rosana Robaina, Stephanie de León, Diego Vallespir
Grupo de Ingeniería de Software
Instituto de Computación
{ceapa, rrobaina, sdeleon, dvallesp}@fing.edu.uy

12 de marzo de 2010

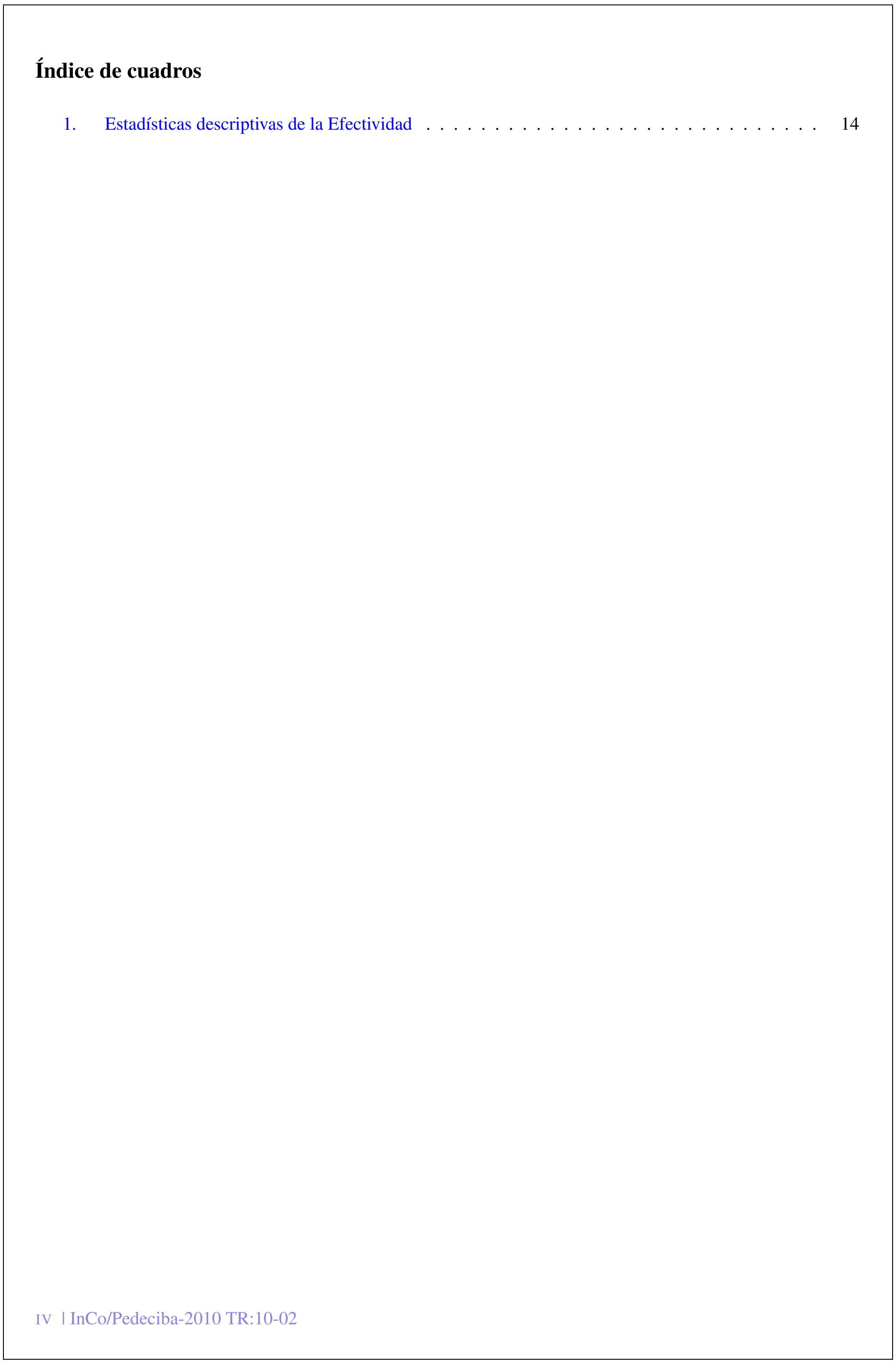
Abstract

En este artículo se presentan conceptos teóricos básicos de la Ingeniería de Software Empírica, así como también técnicas y herramientas de experimentación. La experimentación es un método que se usa para corresponder ideas o teorías con la realidad, proporcionando evidencia que soporte las hipótesis o suposiciones que se creen válidas. La experimentación en la Ingeniería de Software no ha alcanzado aún la madurez que tiene la experimentación en otras disciplinas (por ejemplo, biología, química, sociología). Sin embargo, en los últimos años ésta área en la Ingeniería de Software ha cobrado gran importancia y su actividad ha sido creciente.

Aquí se presenta un proceso para realizar experimentos formales. Este proceso es el que sigue el Grupo de Ingeniería de Software de esta Facultad para realizar sus experimentos formales.

Índice

Índice de figuras



Índice de cuadros

1. Introducción

El Grupo de Ingeniería de Software (GrIS) del Instituto de Computación, Facultad de Ingeniería, Universidad de la República se encuentra realizando experimentos formales para conocer el comportamiento de distintas técnicas de verificación [10, 8, 9, 7]. Además, hace varios años que se realizan pruebas de procesos de desarrollo de software en el marco de una asignatura llamada Proyecto de Ingeniería de Software [6]. Si bien estas pruebas no son formales, es interesante en un futuro formalizarlas.

Para poder realizar experimentos formales se deben conocer los conceptos, las técnicas y las herramientas normalmente usadas en la Ingeniería de Software Empírica (ISE). Esta área, relativamente nueva de la Ingeniería de Software (IS), ha causado un impacto considerable en la comunidad científica y en la industria, teniendo su propia revista internacional (Empirical Software Engineering: An International Journal)[1] desde el año 1996.

Este reporte tiene como objetivo presentar los conceptos fundamentales de ISE. Se pretende que este documento sea utilizado por Proyectos de Grado de la carrera Ingeniería en Computación que se encuentran realizando trabajos de ISE con el GrIS. Distintos estudiantes de Proyecto de Grado se encuentran trabajando con nosotros en estos temas y parece razonable tener un documento que sea común a todos estos proyectos. De esta manera los estudiantes pueden usar este documento como punto de partida para comprender la ISE. Además, pueden incluir este documento como parte de su informe de proyecto evitando tener un enfoque distinto de la ISE en cada Proyecto de Grado.

Este reporte se basa casi completamente en los libros *Experimentation in Software Engineering: An Introduction* [11], *Basics of Software Engineering Experimentation* [2] y *Software Metrics - A Rigorous And Practical Approach* [1].

En la sección 2 se presentan los distintos enfoques y estrategias de la ISE. Una de estas estrategias es la de experimentos formales, estos se describen en la sección 3. Por último, en la sección 4 se describe un proceso para llevar adelante un experimento formal.

2. Enfoques y Estrategias

La ISE utiliza métodos y técnicas experimentales como instrumentos para la investigación. La evidencia empírica proporciona un soporte para la evaluación y validación de atributos (p.e. costo, eficiencia, calidad) en varios tipos de elementos de Ingeniería de Software (p.e. productos, procesos, técnicas, etc.). Se basa en la experimentación como método para corresponder ideas o teorías con la realidad, la cual refiere a mostrar con hechos las especulaciones, suposiciones y creencias sobre la construcción de software.

Se pueden distinguir dos enfoques diferentes al realizar una investigación empírica: el enfoque cualitativo y el cuantitativo. El enfoque **cualitativo** se basa en estudiar la naturaleza del objeto y en interpretar un fenómeno a partir de la concepción que las personas tienen del mismo. Los datos que se obtienen de estas investigaciones están principalmente compuestos por texto, gráficas e imágenes, entre otros.

El enfoque **cuantitativo** se corresponde con encontrar una relación numérica entre dos o más grupos. Se basa en cuantificar una relación o comparar variables o alternativas bajo estudio. Los datos que se obtienen en este tipo de estudios son siempre valores numéricos, lo que permite realizar comparaciones y análisis estadístico.

Es posible utilizar los enfoques cualitativos y cuantitativos para investigar el mismo tema, pero cada enfoque responde a diferentes interrogantes. Se puede considerar que estos enfoques son complementarios más que competitivos, ya que el enfoque cualitativo puede ser usado como base para definir la hipótesis que luego puede ser correspondida cuantitativamente con la realidad. Cabe destacar que las investigaciones cuantitativas pueden obtener resultados más justificables y formales que los cualitativos.

Hay 3 tipos principales de técnicas o estrategias para la investigación empírica: las encuestas, los casos de estudio y los experimentos.

Las **encuestas** se utilizan o bien cuando una técnica o herramienta ya ha sido usada o antes de comenzar a hacerlo. Son estudios retrospectivos de las relaciones y los resultados de una situación. Se puede realizar este tipo de investigación cuando una técnica, o herramienta ya ha sido utilizada o antes de que ésta sea introducida. Las encuestas son realizadas sobre una muestra representativa de la población, y luego los resultados son gene-

[1]http://www.springer.com/computer/programming/journal/10664

ralizados al resto de la población. El ámbito donde son más usadas es en ciencias sociales, por ejemplo, para determinar cómo la población va a votar en la siguiente elección.

En la Ingeniería de Software Empírica las encuestas se utilizan de forma similar, se obtiene un conjunto de datos de un evento que ha ocurrido para determinar cómo reacciona la población frente a una técnica, herramienta o método particular, o para determinar relaciones o tendencias. En un estudio es fundamental seleccionar correctamente las variables a estudiar, pues de ellas dependen los resultados que se pueden obtener. Si los resultados no permiten concluir sobre los objetivos del estudio se han elegido mal las variables.

Una de las características más relevantes de las encuestas es que proveen un gran número de variables para estudiar. Esto hace posible construir una variedad de modelos y luego seleccionar el que mejor se ajusta a los propósitos de la investigación, evitando tener que especular cuáles son las variables más relevantes. Dependiendo del diseño de la investigación (cuestionario) las encuestas pueden ser clasificadas como cualitativas o cuantitativas.

Los **casos de estudio** son métodos observacionales, se basan en la observación de una actividad o proyecto durante su curso. Son utilizados para monitorear proyectos, o actividades y para investigar entidades o fenómenos en un período específico.

En un caso de estudio se identifican los factores clave que pueden afectar la salida de una actividad, y se documentan las entradas, las limitaciones, los recursos y las salidas. El nivel de control de la ejecución es menor en los casos de estudio que en los experimentos. Esto se debe principalmente a que en los casos de estudio no se controla, sólo se observa, contrario a lo que ocurre en los experimentos.

Los casos de estudio son muy útiles en el área de Ingeniería de Software, se usan en la evaluación industrial de métodos y herramientas. Además, son fáciles de planificar aunque los resultados son difíciles de generalizar y comprender. Los casos de estudio no manipulan las variables, sino que éstas son determinadas por la situación que se está investigando.

Al igual que las encuestas, los casos de estudio pueden ser clasificados como cualitativos o cuantitativos dependiendo de lo que se quiera investigar del proyecto en curso.

Los **experimentos** son generalmente ejecutados en un ambiente de laboratorio, el cual brinda un alto grado de control. El objetivo en un experimento es manipular una o más variables y controlar el resto. Un experimento es una técnica formal, rigurosa y controlada de llevar a cabo una investigación.

En las secciones siguientes se profundiza en los experimentos formales como técnica de investigación.

3. Experimentos Formales

Como se mencionó anteriormente, los experimentos son una técnica de investigación en la cual se quiere tener un mejor control del estudio y del entorno en el que éste se lleva a cabo.

Los experimentos son apropiados para investigar distintos aspectos de la IS, como ser: confirmar teorías, explorar relaciones, evaluar la exactitud de los modelos y validar medidas. Tienen un alto costo respecto de las otras técnicas de investigación, pero a cambio ofrecen un control total de la ejecución y son de fácil replicación.

3.1. Terminología

En esta sección se presentan los términos más comunmente usados en diseño experimental. Se usan dos ejemplos de experimentos a lo largo de esta sección para introducir dichos términos.

En el primer ejemplo se tiene un experimento en el campo de la medicina, mediante el cual se quiere conocer la efectividad de los analgésicos en las personas entre 20 y 40 años de edad, llamado «Efec-Analgésicos».

En el segundo ejemplo, se quiere conocer la efectividad de 5 técnicas de verificación sobre un conjunto de programas, llamado «Efec-Técnicas».

Los objetos sobre los cuales se ejecuta el experimento son llamados **Unidades Experimentales** u objetos experimentales. La unidad experimental en un experimento de Ingeniería de Software podría llegar a ser el proyecto de software como un todo o cualquier producto intermedio durante el proceso.

Para *Efec-Analgésicos* se tiene que la unidad experimental es un grupo de personas entre 20 y 40 años de edad, en ese grupo de personas es en donde se observa el efecto de los analgésicos. En el ejemplo de *Efec-Técnicas*, se tiene que la unidad experimental es el conjunto de programas sobre los cuales se aplican las técnicas de verificación.

Aquellas personas que aplican los métodos o técnicas a las unidades experimentales se les llama **Sujetos Experimentales**. A diferencia de otras disciplinas, en la IS los sujetos experimentales tienen un importante efecto en los resultados del experimento, por lo tanto es una variable que debe ser cuidadosamente considerada.

En *Efec-Analgésicos* los sujetos son aquellas personas que administran los analgésicos a ser consumidos por los pacientes (enfermeros por ejemplo). Cómo los enfermeros administran los analgésicos a los pacientes no es algo que se espere vaya a afectar el experimento. La forma en que un enfermero administra un analgésico a un paciente es poco probable que sea diferente a la de otro, y aunque lo fuera, no se espera que afecte los resultados del experimento.

En *Efec-Técnicas* los sujetos pueden ser ingenieros que aplican la técnica en un conjunto particular de programas (unidad experimental). En este caso, los resultados del experimento podrían diferir mucho de acuerdo a la formación y experiencia de los ingenieros, así como también la forma en que las técnicas son aplicadas, incluso el estado de ánimo del verificador podría influir en los resultados.

El resultado de un experimento es llamado **Variable de Respuesta**. Este resultado debe ser cuantitativo. Una variable de respuesta puede ser cualquier característica de un proyecto, fase, producto o recurso que es medida para verificar los efectos de las variaciones que se provocan de una aplicación a otra. En ocasiones, a una variable de respuesta se le llama también variable dependiente.

En *Efec-Analgésicos* la efectividad podría ser medida en el grado de alivio del dolor en un determinado lapso de tiempo, o bien qué tan rápido el analgésico alivia el dolor. En ambos casos, la variable debe ser expresada cuantitativamente. En el primer caso se podría tener una escala, en la cual cada valor signifique un grado de alivio del dolor, en el segundo caso, el lapso de tiempo en que el analgésico es efectivo, se podría medir en minutos.

Para *Efec-Técnicas* la efectividad podría ser medida de acuerdo a la cantidad de defectos que encuentra la técnica sobre la cantidad de defectos totales del software verificado.

Un **Parámetro** es cualquier característica que permanezca invariable a lo largo del experimento. Son características que no influyen o que no se desea que influyan en el resultado del experimento o en la variable de respuesta. Los resultados del experimento serán particulares a las condiciones definidas por los parámetros. El conocimiento resultante podrá ser generalizado solamente considerando los parámetros como variables en sucesivos experimentos y estudiando su impacto en las variables de respuesta.

En el ejemplo de *Efec-Analgésicos* se tiene que el rango de edades (entre 20 y 40 años de edad) es un parámetro del experimento, los resultados serán particulares para el rango establecido.

En *Efec-Técnicas* un parámetro posible es el tamaño del software a ser verificado (por ejemplo: que tenga entre 200 y 500 LOCs). Otro parámetro para este experimento podría ser la experiencia de los verificadores, en este caso se podría fijar la experiencia en un determinado nivel.

Cada característica del desarrollo de software a ser estudiada que afecta a las variables de respuesta se denomina **Factor**. Cada factor tiene varias alternativas posibles. Lo que se estudia, es la influencia de las alternativas en los valores de las variables de respuesta. Los factores de un experimento son cualquier característica que es intencionalmente modificada durante el experimento y que afecta su resultado.

El factor en *Efec-Analgésicos* es «los analgésicos», en *Efec-Técnicas* tenemos que el factor es «las técnicas de verificación». Para ambos casos el factor se varía intencionalmente (se varía el tipo de analgésico o tipo de técnica de verificación) para ver cómo afecta en la efectividad.

Los posibles valores de los factores en cada unidad experimental son llamados **Alternativas** o niveles. En algunos casos también se les llama tratamientos.

Las alternativas de *Efec-Analgésicos* son los distintos tipos de analgésicos que se estudian en el experimento (p.e. Aspirina, Zolben, etc). De igual forma, para *Efec-Técnicas* las distintas alternativas son los 5 tipos distintos de técnicas que se estudian.

El intento de ajustar determinadas características de un experimento a un valor constante no es siempre posible. Es inevitable y a veces indeseable tener variaciones de un experimento a otro. Éstas variaciones son conocidas como **Bloqueo de Variables** y dan lugar a un determinado tipo de diseño experimental, llamado *block design*.

Una variable indeseada para *Efec-Analgésicos* podría ser el «umbral del dolor». Si se aplica una alternativa de analgésico a personas con umbral del dolor alto y otra alternativa a personas con umbral del dolor bajo, se tendría una variación indeseada, ya que la efectividad que se mida de los distintos tipos de analgésico va a variar no solamente por el tipo de analgésico administrado sino por el nivel de umbral del dolor del paciente al cual se lo administra.

En el caso de *Efec-Técnicas*, podría resultar que la experiencia de los verificadores resultase una variación indeseada si no se la tiene en cuenta previamente. Una forma de bloquear la experiencia en verificación podría ser dividir a los participantes en dos grupos: uno de verificadores experientes y otro sin experiencia.

Cada ejecución del experimento que se realiza en una unidad experimental es llamada **experimento unitario** o experimento elemental. Lo que significa que cada aplicación de una combinación de alternativas de factores por un sujeto experimental en una unidad experimental es un experimento elemental.

Un experimento elemental es cada terna $<analgésico_i, enfermero_j, paciente_k>$ para el ejemplo de *Efec-Analgésicos*. Para el ejemplo de *Efec-Técnicas* sería la terna $<técnica_i, verificador_j, software_k>$.

La figura 1 ilustra la interacción entre los distintos tipos de componentes de un experimento.

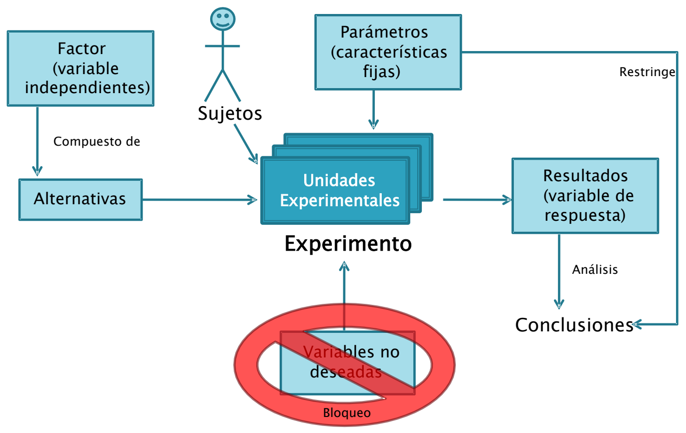


Figura 1. Componentes en un experimento de Ingeniería de Software

3.2. Principios generales de diseño

Muchos aspectos deben ser tenidos en cuenta cuando se diseña un experimento. Los principios generales de diseño son: aleatoriedad, bloqueo y balance. A continuación se describe en qué consiste cada principio.

Aleatoriedad: el principio de aleatoriedad es uno de los principios de diseño más importantes. Todos los métodos de análisis estadístico requieren que las observaciones sean de variables independientes aleatorias. Por consiguiente, tanto las alternativas de los factores como los sujetos tienen que ser elegidos de forma aleatoria, ya que los sujetos tienen un impacto crítico en el valor de las variables de respuesta.

La aleatoriedad que se puede aplicar a un experimento también depende del tipo de diseño que se haya elegido. Por ejemplo, si se tienen dos factores A y B, cada uno con dos posibles alternativas (a1, a2, b1 y b2), las alternativas deben ser combinadas de la siguiente forma: a1b1, a1b2, a2b1, a2b2, ya que cuando se tienen dos factores se quiere observar el efecto de cada alternativa por separado y de la interacción entre ambas.

Esta combinación de alternativas es especificada por el tipo de diseño experimental que se eligió. Sin embargo, las cuatro combinaciones deben ser asignadas de forma aleatoria a los proyectos y sujetos, y es ahí en donde la aleatoriedad se aplica.

Bloqueo: la técnica de bloqueo se usa cuando se tienen factores que probablemente tengan efectos indeseados en las variables de respuesta y éstos efectos son conocidos y controlables.

Como se mencionaba en el ejemplo de *Efec-Técnicas* en la sección anterior, algunos verificadores podrían tener experiencia en el uso de las técnicas de verificación y otros no. Entonces, para minimizar el efecto de la experiencia, se agrupan a los participantes en dos grupos, uno con verificadores experientes y otro sin experiencia.

Balance: el balance es deseable ya que simplifica y fortalece el análisis estadístico de los datos, aunque no es necesario. Tomando como ejemplo el experimento de *Efec-Analgésicos* nuevamente, sería deseable que la cantidad de personas a las cuales se les administra Zolben sea igual a la cantidad de personas que se les administra Aspirina.

3.3. Tipos de Diseño

En el proceso del diseño experimental, primero se debe decidir (basándose en los objetivos del experimento) a qué factores y alternativas estarán sujetas las unidades experimentales y qué parámetros deben ser establecidos. Luego, se debe examinar si existe la posibilidad de que algunos de los parámetros no pueda mantenerse en un valor constante, en ese caso se debe tener en cuenta cualquier variación indeseable. Finalmente, se debe elegir qué variables de respuesta serán medidas y cuáles serán los objetos y sujetos experimentales.

Teniendo establecidos los parámetros, factores, variables de bloqueo y variables de respuesta, se debe elegir el tipo de diseño experimental, en el cual se establece cuántas combinaciones de experimentos unitarios y alternativas deben haber.

Los distintos tipos de diseño experimental dependen del objetivo del experimento, del número de factores, de las alternativas de los factores y de la cantidad de variaciones indeseadas, entre otros.

Los tipos de diseño experimental se dividen en diseños de *un solo factor* y diseños de *múltiples factores*. A continuación se profundiza en los experimentos de un solo factor.

3.3.1. Diseño de un solo factor (*One-Factor Design*)

Para experimentos con un solo factor existen distintos tipos de diseños estándar, los principales son: los completamente aleatorios y los aleatorios con comparación por pares.

Los diseños **completamente aleatorios** son los tipos de diseño más simples, en los cuales se intenta comparar dos o más alternativas aplicadas a un determinado número de unidades experimentales, en donde cada unidad experimental se ve afectada una única vez, y por ende, por una sola alternativa. La asignación de las alternativas a los experimentos debe ser de forma aleatoria para asegurar la validez del análisis de datos.

Tomando como ejemplo *Efec-Técnicas* y suponiendo que el conjunto de programas sobre el cual se quiere conocer la efectividad de las técnicas lo componen diez programas distintos, se tendría que asignar las técnicas y los ingenieros de forma aleatoria a los programas que se vayan a verificar.

Una posible asignación aleatoria sería tener en una bolsa los nombres de todas las técnicas de verificación a aplicar, en donde la primera que se extraiga se aplique al programa P_1, la segunda a P_2 y así hasta el programa P_{10}. Luego de tener las duplas Programa-Técnica, efectuar la misma asignación aleatoria con los participantes: el primer participante extraído se lo asigna la dupla (P_1, T_x), el segundo a la dupla (P_2, T_y), y así sucesivamente.

El análisis estadístico que se puede hacer a este tipo de experimentos varía según si se aplican 2 o más alternativas para el factor.

Los diseños **aleatorios con comparación por pares** tienen como objetivo encontrar cuál es la mejor alternativa respecto de una determinada variable de respuesta. Estos tipos de diseño tienen la particularidad de que las alternativas se aplican al mismo experimento, instanciado en más de una unidad experimental.

Para el experimento de *Efec-Técnicas* no sería una buena decisión que cada ingeniero verificara 2 veces el mismo programa. En la segunda instancia de verificación, el ingeniero posee conocimiento tanto de los defectos del programa como de la tarea de verificar propiamente dicha (aunque sea con una técnica distinta). Por esto, para comparar las dos técnicas, ambas tienen que ser aplicadas por primera vez por ingenieros distintos, pero con similares características (ya que encontrar uno igual es imposible). La alternativa que debe aplicar cada ingeniero al programa debe ser asignada de forma aleatoria y no debe verificar un mismo programa más de una vez.

En este tipo de diseños se bloquean cierto tipo de variables que representan restricciones en la aleatoriedad que se le puede dar. Tomando como ejemplo nuevamente a *Efec-Técnicas*, si un verificador sin experiencia aplica más de una técnica durante el experimento, no sería deseable asignar al azar la técnica que cada verificador aplica en cada verificación.

Existe un efecto de aprendizaje en el cual, luego de que un verificador ejecutó una verificación, éste generó conocimiento sobre la verificación en sí, independientemente de la técnica que haya aplicado, y éste conocimiento influye significativamente en la segunda instancia de verificación que vaya a aplicar. Por tanto, la aleatoriedad en el orden de la asignación de técnicas en este ejemplo no es del todo deseable.

4. Proceso Experimental

Como se mencionó anteriormente, los experimentos son una técnica de investigación en la cual se quiere tener un mejor control del estudio y del entorno en el que éste se lleva a cabo.

Los experimentos son apropiados para investigar distintos aspectos de la IS, como ser: confirmar teorías, explorar relaciones, evaluar la exactitud de los modelos y validar medidas. Tienen un alto costo respecto de las otras técnicas de investigación, pero a cambio ofrecen un control total de la ejecución y son de fácil replicación.

El proceso para llevar a cabo un experimento está formado por varias fases: definición, planificación, operación, análisis e interpretación y presentación.

La primer fase es la de **definición**, en donde se define el experimento en términos del problema, objetivos y metas. La siguiente fase es la **planificación**, en la cual se determina el diseño del experimento. En la fase de **operación** se ejecuta el diseño del experimento, en donde se recolectan los datos que serán analizados posteriormente en la fase de **análisis e interpretación**. En esta última fase, conceptos estadísticos son aplicados para analizar los datos. Por último, se muestran los resultados obtenidos en la fase de **presentación**.

En la figura 2 se muestra una visión general de todo el proceso. Cada una de las fases que lo componen se detallan a continuación.

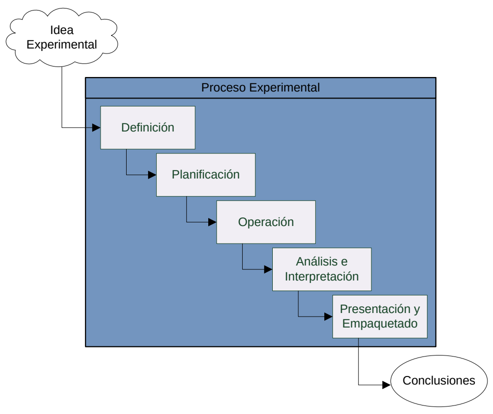


Figura 2. Visión general del Proceso Experimental

4.1. Definición

En la fase de Definición se determinan las bases del experimento, que se ilustra en la figura 3. Para ello se debe definir **el problema que se quiere resolver**, **propósito del experimento** y **los objetivos y metas** del mismo.

Para el planteo del objetivo del experimento se debe definir *el objeto de estudio*, que es la entidad que va a ser estudiada en el experimento. Puede ser un producto, proceso, recurso u otro. También se debe establecer el *propósito*: la intención del experimento. Por ejemplo, evaluar diferentes técnicas de verificación.

Se debe definir además el *foco de calidad*, que refiere al efecto primario que esta bajo estudio, ejemplos son la efectividad y el costo de las técnicas de verificación. El propósito y el foco de calidad son las bases para las hipótesis del experimento.

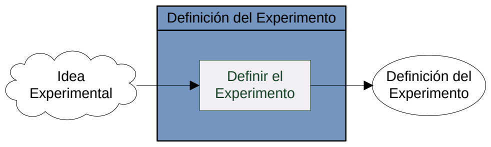


Figura 3. Fase de Definición del Experimento

Otro aspecto que debe estar presente es la *perspectiva*, que refiere al punto de vista con que los resultados obtenidos son interpretados. Por ejemplo, los resultados de la comparación de técnicas de verificación pueden verse desde la perspectiva de un experimentador, de un investigador o de un profesional. Un experimentador verá el estudio como una demostración de como una técnica de verificación puede ser evaluada. Un investigador puede ver el estudio como una base empírica para refinar teorías sobre la verificación de software, enfocándose en los datos que apoyan o refutan estas teorías. Un profesional puede ver el estudio como una fuente de información sobre qué técnicas de verificación deberían aplicarse en la práctica.

Junto con los aspectos mencionados se debe definir el *contexto*, que es el ambiente en el que se ejecuta el experimento. En este punto se deben definir los *sujetos* que van a llevar a cabo el experimento y los *artefactos* que son utilizados en la ejecución del mismo. Se puede caracterizar el contexto de un experimento según el número de sujetos y objetos definidos en él: un solo objeto y un solo sujeto, un solo sujeto a través de muchos objetos, un solo objeto a través de un conjunto de sujetos, o un conjunto de sujetos y un conjunto de objetos.

4.2. Planificación

La planificación es la fase en la que se define como se va a llevar a cabo el experimento. Esta fase consta de las etapas: selección del contexto, formulación de las hipótesis, elección de las variables, selección de los sujetos, diseño del experimento, instrumentación y evaluación de la validez, que se muestran en la figura 4.

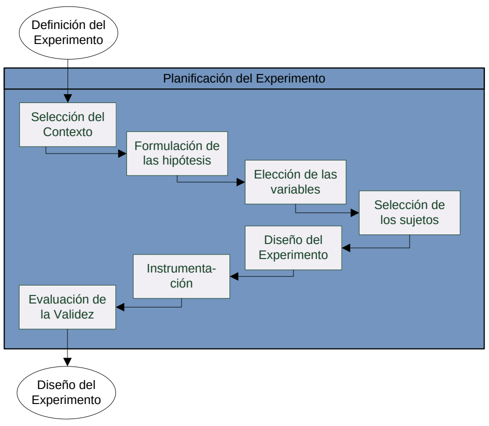


Figura 4. Fase de Planificación del Experimento

La etapa de **selección del contexto** es la etapa inicial de la planificación. En esta etapa se amplía el contexto definido en la etapa de Definición, especificando claramente las características del ambiente donde ejecuta el experimento. Se define si el experimento se va a realizar en un proyecto real (en línea, *on-line*) o en un laboratorio (fuera de línea, *off-line*), características de los sujetos y si el problema es «real» (problema existente en la industria) o «de juguete». También se debe definir si el experimento es válido para un contexto específico o para un dominio general de la Ingeniería de Software.

Una vez que los objetivos están claramente definidos se pueden transformar en una hipótesis formal. La **formulación de las hipótesis** es una fase muy importante dentro de la etapa de planificación, ya que la verificación de la misma es la base para el análisis estadístico. En esta fase se formaliza la definición del experimento en la hipótesis.

Usualmente se definen dos hipótesis, la hipótesis nula y la hipótesis alternativa. La hipótesis nula, denotada H_0, asume que no hay una diferencia significativa entre las alternativas, con respecto a las variables dependientes que se están midiendo. Establece que si hay diferencias entre las observaciones realizadas, éstas son por casualidad, no producto de la alternativa aplicada. Esta hipótesis se asume verdadera hasta que los datos demuestren lo contrario, por lo que el foco del experimento está puesto en rechazarla. Un ejemplo de hipótesis nula es: «No hay diferencia en la cantidad de defectos encontrados por las técnicas de verificación».

En cambio la hipótesis alternativa, denotada H_1, afirma que existe una diferencia significativa entre las alternativas con respecto a las variables dependientes. Establece que las diferencias encontradas son producto de la aplicación de las alternativas. Ésta es la hipótesis a probar, para esto se debe determinar que los datos obtenidos son lo suficientemente convincentes para desechar la hipótesis nula y aceptar la hipótesis alternativa. Un ejemplo de hipótesis alternativa es, si se están comparando dos técnicas de verificación, decir que una encuentra más defectos que la otra. En caso de haber más de una hipótesis alternativa se denotan secuencialmente: H_1, H_2, H_3, $\ldots$, H_n.

Una vez definida la hipótesis, se debe identificar qué variables afectan a la/s alternativa/s. Luego de identificadas las variables se debe decidir el control a ejercer sobre las mismas.

La **selección de las variables** dependientes como la de las independientes están relacionadas, por lo que en muchos casos se realizan en simultáneo. Seleccionar estas variables es una tarea muy compleja, que en ocasiones implica conocer muy bien el dominio. Es importante definir las variables independientes y analizar sus características, para así investigar y controlar los efectos que ejercen sobre las variables dependientes. Se deben identificar las variables independientes que se pueden controlar y las que no. Además, se deben identificar las variables dependientes, mediante las cuales se mide el efecto de las alternativas. Generalmente hay sólo una variable dependiente y se deriva de la hipótesis.

Otro aspecto importante al llevar a cabo un experimento es la **selección de los sujetos**. Para poder generalizar los resultados al resto de la población, la selección debe ser una muestra representativa de la misma. Cuanto más grande es la muestra, menor es el error al generalizar los datos. Existen dos tipos de muestras que se pueden seleccionar: la probabilística, donde se conoce la probabilidad de seleccionar cada sujeto; y la no-probabilística, donde esta probabilidad es desconocida.

Luego de definir el contexto, formalizar las hipótesis, y seleccionar las variables y los sujetos, se debe **diseñar el experimento**. Es muy importante planear y diseñar cuidadosamente el experimento, para que los datos obtenidos puedan ser interpretados mediante la aplicación de métodos de análisis estadísticos.

Para comenzar a diseñar un experimento se debe elegir el diseño adecuado. Se debe planificar y diseñar el conjunto de las combinaciones de alternativas, sujetos y objetos, que conforman los experimentos unitarios. Se describe cómo estos experimentos unitarios deben ser organizados y ejecutados.

La elección del diseño del experimento afecta el análisis estadístico y viceversa, por lo que al elegir el diseño del experimento se debe tener en cuenta qué análisis estadístico es el mejor para rechazar la hipótesis nula y aceptar la alternativa.

Luego de diseñar el experimento y antes de la ejecución es necesario contar con todo lo necesario para la correcta ejecución del mismo. La **instrumentación** involucra, de ser necesario, capacitación a los sujetos, preparación de los artefactos, construcción de guías, descripción de procesos, planillas y herramientas. También implica configurar el hardware, mecanismos de consultas y experiencias piloto, entre otros. La finalidad de esta fase es proveer todo lo necesario para la realización y monitorización del experimento.

4.3. Evaluación de la Validez

Una pregunta fundamental antes de pasar a ejecutar el experimento es cuán válidos serían los resultados. Existen cuatro categorías de amenazas a la validez: validez de la conclusión, validez interna, validez del constructo y validez externa.

Las amenazas que afectan la **validez de la conclusión** refieren a las conclusiones estadísticas. Amenazas que afecten la capacidad de determinar si existe una relación entre la alternativa y el resultado, y si las conclusiones obtenidas al respecto son válidas. Ejemplos de estas son la elección de los métodos estadísticos, y la elección del tamaño de la muestra, entre otros.

Las amenazas que influyen en la **validez interna** son aquellas referidas a observar relaciones entre la alternativa y el resultado que sean producto de la casualidad y no del resultado de la aplicación de un factor. Esta «casualidad» es provocada por elementos desconocidos que influyen sobre los resultados sin el conocimiento de los investigadores. Es decir, la validez interna se basa en asegurar que la alternativa en cuestión produce los resultados observados.

La **validez del constructo** indica cómo una medición se relaciona con otras de acuerdo con la teoría o hipótesis que concierne a los conceptos que se están midiendo. Un ejemplo se puede observar al momento de seleccionar los sujetos en un experimento. Si se utiliza como medida de la experiencia del sujeto el número de cursos que tiene aprobados en la universidad, no se está utilizando una buena medida de la experiencia. En cambio, una buena medida puede ser utilizar la cantidad de años de experiencia en la industria o una combinación de ambas cosas.

La **validez externa** está relacionada con la habilidad para generalizar los resultados. Se ve afectada por el diseño del experimento. Los tres riesgos principales que tiene la validez externa son: tener los participantes equivocados como sujetos, ejecutar el experimento en un ambiente erróneo y realizar el experimento en un momento que afecte los resultados.

4.4. Operación

Luego de diseñar y planificar el experimento, éste debe ser ejecutado para recolectar los datos que se quieren analizar. La operación del experimento consiste en tres etapas: preparación, ejecución y la validación de los datos, que se muestran en la figura 5.

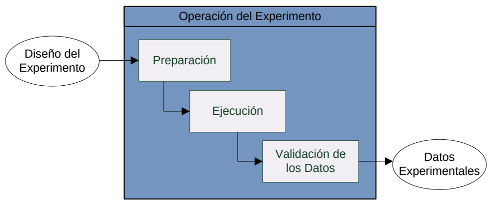


Figura 5. Fase de Operación del Experimento

En la etapa de preparación se seleccionan los sujetos y se preparan los artefactos a ser utilizados.

Es muy importante que los sujetos estén motivados y dispuestos a realizar las actividades que les sean asignadas, ya sea que tengan conocimiento o no de su participación en el experimento. Se debe intentar obtener consentimiento por parte de los participantes, que deben estar de acuerdo con los objetivos de la investigación. Los resultados obtenidos pueden volverse inválidos si los sujetos al momento que deciden participar no saben lo que tienen que hacer o tienen un concepto erróneo.

Es importante considerar la sensibilidad de los resultados que se obtienen de los sujetos, por ejemplo: es importante asegurar a los participantes que los resultados obtenidos sobre su rendimiento se mantienen en secreto y no se usarán para perjudicarlos en ningún sentido. Se debe tener en cuenta también los incentivos, ya que ayudan a motivar a los sujetos, pero se corre el riesgo de que participen sólo por el incentivo, lo que puede ser perjudicial para el experimento. En caso de no tener otra alternativa que no sea engañar a los sujetos, se debe procurar explicar y revelar el engaño lo más temprano posible.

Como se vio en la instrumentación, para que los sujetos comiencen la ejecución es necesario tener prontos todos los instrumentos, formularios, herramientas, guías y otros artefactos que sean necesarios para la ejecución del experimento. Muchas veces se debe preparar un conjunto de instrumentos especial para cada sujeto y otras se utiliza el mismo conjunto de artefactos para todos los sujetos.

Existen muchas formas distintas de ejecutar los experimentos, la duración varía desde días hasta años.

Los datos pueden ser recolectados de las siguientes formas:

- Manualmente mediante el llenado de formularios por parte de los sujetos.
- Manualmente soportado por herramientas.
- Mediante entrevistas.
- Automáticamente por herramientas.

La primera es la forma más común y no requiere mucho esfuerzo por parte del experimentador. Tanto en los formularios como en los métodos soportados por herramientas no es posible identificar inconsistencias o defectos hasta que no se recolecte la información, o hasta que los sujetos los descubran. En las entrevistas, el contacto con los sujetos es mucho mayor permitiendo una mejor comunicación con ellos durante la recolección de datos. Éste método es el que requiere mas esfuerzo por parte del investigador.

Un aspecto muy importante a la hora de ejecutar los experimentos es el ambiente de ejecución, tanto si el experimento se realiza dentro de un proyecto de desarrollo común o si se crea un ambiente ficticio para su ejecución. En el primer caso el experimento no debería afectar el proyecto más de lo necesario, ya que la razón de realizar el experimento dentro de un proyecto es ver los efectos de las alternativas en el ambiente del proyecto. Si el experimento cambia demasiado el ambiente del proyecto, éstos efectos se perderían.

Cuando se obtienen los datos, se debe chequear que fueron recolectados correctamente y que son razonables. Algunas fuentes de error son que los sujetos llenen mal sus planillas, o no recolecten los datos seriamente, lo que hace que se descarten datos. Es importante revisar que los sujetos hagan un trabajo serio y responsable y que apliquen las alternativas en el orden correcto, en otras palabras: que el experimento sea ejecutado en la forma en que fue planificado. De lo contrario los resultados podrían ser inválidos.

4.5. Análisis e Interpretación

Luego de que finaliza la ejecución del experimento y se cuenta con los datos recolectados, comienza la fase de análisis de los mismos conforme a los objetivos planteados.

Un aspecto importante a considerar en el análisis de los datos es la **escala de medida**. La escala de medida utilizada para recolectar los datos restringe el tipo de cálculos estadísticos que se pueden realizar. Una medida es un mapeo de un atributo de una entidad a un valor de medida, por lo general un valor numérico. Las entidades son objetos que se observan en la realidad, por ejemplo, una técnica de verificación.

El propósito de mapear los atributos en un valor de medida es caracterizar y manipular los atributos formalmente. La medida seleccionada debe ser válida, por tanto, no debe violar ninguna propiedad necesaria del atributo que mide, y debe ser una caracterización matemática adecuada del atributo.

El mapeo de un atributo a un valor de medida puede realizarse de varias formas. Cada tipo de mapeo posible de un atributo se conoce como escala. Los tipos más comunes de escala son:

- Escala Nominal.- Es la menos poderosa de las escalas. Solo mapea el atributo de la entidad en un nombre o símbolo. El mapeo puede verse como una clasificación de las entidades acorde al atributo. Ejemplos de escala nominal son clasificaciones, etiquetados, entre otras.

- Escala Ordinal.- La escala ordinal categoriza las entidades según un criterio de ordenación. Es más poderosa que la escala nominal. Ejemplos de criterios de ordenación son «mayor que», «mejor que» y «más complejo». Ejemplos de escala nominal son grados, complejidad del software, entre otras.

- Escala de intervalo.- La escala de intervalo se utiliza cuando la diferencia entre dos medidas es significativa, pero no el valor en si mismo. Este tipo de escala ordena los valores de la misma forma que la escala ordinal, pero existe la noción de «distancia relativa» entre dos entidades. Esta escala es más poderosa que la ordinal. Ejemplos de escala de intervalo son la temperatura medida en Celsius o Fahrenheit.

- Escala ratio (cociente de dos números).- Si existe un valor cero significativo y la división entre dos medidas es significativa, se puede utilizar una escala ratio. Ejemplos de escala ratio son distancia, temperatura medida en Kelvin, etc.

Después de obtener los datos es necesario interpretarlos para llegar a conclusiones válidas. La interpretación se realiza en tres etapas: caracterizar el conjunto de datos usando estadística descriptiva, reducción del conjunto de datos y realización de las pruebas de hipótesis que se ilustran en la figura 6.

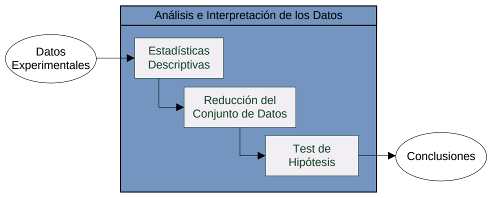

Análisis e Interpretación de los Datos
Datos Experimentales
Estadísticas Descriptivas
Reducción del Conjunto de Datos
Test de Hipótesis
Conclusiones

Figura 6. Fase de Análisis e Interpretación de los Datos del Experimento

4.5.1. Estadística Descriptiva

La **estadística descriptiva** se utiliza antes de la prueba de hipótesis, para entender mejor la naturaleza de los datos y para identificar datos falsos o anormales. Los aspectos principales que se examinan son: la tendencia central, la dispersión y la dependencia. A continuación se presentan las medidas más comunes de cada uno de estos aspectos. Para ello se asume que existen $x_1 \ldots x_n$ muestras.

Las **medidas de tendencia central** indican «el medio» de un conjunto de datos. Entre las medidas más comunes se encuentran: la media aritmética, la mediana y la moda.

La *media aritmética* se conoce como el promedio, y se calcula sumando todas las muestras y dividiendo el total por el número de muestras:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

La *media*, denotada $\bar{x}$, resume en un valor las características de una variable teniendo en cuenta a todos los casos. Es significativa para las escalas de intervalo y ratio.

La *mediana*, denotada $\tilde{x}$, representa el valor medio de un conjunto de datos, tal que el número de muestras que son mayores que la mediana es el mismo que el número de muestras que son menores que la mediana. Se calcula ordenando las muestras en orden ascendente o descendente, y seleccionando la observación del medio. Este cálculo está bien definido si n es impar. Si n es par, la mediana se define como la media aritmética de los dos valores medios. Esta medida es significativa para las escalas ordinal, de intervalo y ratio.

La *moda* representa la muestra más común. Se calcula contando el número de muestras para cada valor único y seleccionando el valor con más cantidad. La moda esta bien definida si hay solo un valor más común que los otros. Si este no es el caso, se calcula como la mediana de las muestras más comunes. La moda es significativa para las escalas nominal, ordinal, de intervalo y ratio.

La media aritmética y la mediana son iguales si la distribución de las muestras es simétrica. Si la distribución es simétrica y tiene un único valor máximo, las tres medidas son iguales.

Las medidas de tendencia central no proveen información sobre la dispersión del conjunto de datos. Cuanto mayor es la dispersión, más variables son las muestras, cuanto menor es la dispersión, más homogéneas a la media son las muestras.

Las **medidas de dispersión** miden el nivel de desviación de la tendencia central, o sea, que tan diseminados o concentrados están los datos respecto al valor central. Entre las principales medidas de dispersión están: la varianza, la desviación estándar, el rango y el coeficiente de variación.

La *varianza* (s^2) que presenta una distribución respecto de su media se calcula como la media de las desviaciones de las muestras respecto a la media aritmética. Dado que la suma de las desviaciones es siempre cero, se toman las desviaciones al cuadrado:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{2}$$

Se divide por $n-1$ y no por n, porque dividir por $n-1$ provee a la varianza de propiedades convenientes. La varianza es significativa para las escalas de intervalo y ratio.

La *desviación estándar*, denotada s, se de

ne como la raíz cuadrada de la varianza:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{3}$$

A menudo esta medida se prefiere sobre la varianza porque tiene las mismas dimensiones (unidad de medida) que los valores de las muestras. En cambio, la varianza se mide en unidades cuadráticas. La desviación estándar es significativa para las escalas de intervalo y ratio.

La dispersión también se puede expresar como un porcentaje de la media. Este valor se llama *coeficiente de variación*, y se calcula como:

$$100 \cdot \frac{s}{\bar{x}} \tag{4}$$

Esta medida no tiene dimensión y es significativa para la escala ratio. Permite comparar la dispersión o variabilidad de dos o más grupos.

El **rango** de un conjunto de datos es la distancia entre el valor máximo y el mínimo:

$$range = x_{max} - x_{min} \tag{5}$$

Es una medida significativa para las escalas de intervalo y ratio. Cuando el conjunto de datos consiste en muestras relacionadas de a pares (xi; yi) de dos variables, X e Y, puede ser interesante examinar la dependencia entre estas variables. Las principales medidas de dependencia son: regresión lineal, covarianza y el coeficiente de correlación lineal.

4.5.2. Reducción del Conjunto de Datos

Para las pruebas de hipótesis se utilizan métodos estadísticos. El resultado de aplicar estos métodos depende de la calidad de los datos. Si los datos no representan lo que se cree, las conclusiones que se derivan de los resultados de los métodos son incorrectas. Errores en el conjunto de datos pueden ocurrir por un error sistemático, o por lo que se conoce en estadística con el nombre de outlier. Un outlier es un dato mucho más grande o mucho más chico de lo que se puede esperar observando el resto de los datos.

Las estadísticas descriptivas se ven fuertemente influenciadas por aquellas observaciones que su valor dista significativamente del resto de los valores recolectados. Estas observaciones llevan el nombre de *outliers*.

Los *outliers* influencian las medidas de dispersión, aumentando la variabilidad de lo que se está midiendo. En algunos casos se realiza un análisis acerca de estos valores que difieren mucho de la media y se decide quitarlos de los datos a analizar porque no son representativos de la población, ya que fueron causados por algún tipo de anomalía: errores de medición, variaciones no deseadas en las características de los sujetos, entre otras.

Quitar *outliers* requiere de un análisis pormenorizado, por quitar outliers se demoró en detectar el agujero de la capa de ozono. [2]

Una vez identificado un outlier se debe identificar su origen para decidir qué hacer con él. Si se debe a un evento raro o extraño que no volverá a ocurrir, el punto puede ser excluido. Si se debe a un evento extraño que

[2] En 1985 los científicos británicos anunciaron un agujero en la capa de ozono sobre el polo sur. El reporte fue descartado ya que observaciones más completas, obtenidas por instrumentos satelitales, no mostraban nada inusual. Luego, un análisis más exhaustivo reveló que las lecturas de ozono en el polo sur eran tan bajas que el programa que las analizaba las había suprimido automáticamente como outliers en forma equivocada.

puede volver a ocurrir, no es aconsejable excluir el valor del análisis, pues tiene información relevante. Si se debe a una variable que no fue considerada, debería ser considerado para basar los cálculos y modelos también en esta variable.

4.5.3. Pruebas de Hipótesis

El objetivo de la **prueba de hipótesis** es ver si es posible rechazar cierta hipótesis nula H_0. Si la hipótesis nula no es rechazada, no se puede decir nada sobre los resultados. En cambio, si es rechazada, se puede declarar que la hipótesis es falsa con una significancia dada (α). Este nivel de significancia también es denominado nivel de riesgo o probabilidad de error, ya que se corre el riesgo de rechazar la hipótesis nula cuando en realidad es verdadera. Este nivel está bajo el control del experimentador.

Para probar H_0 se define una unidad de prueba t y un área crítica C, la cual es parte del área sobre la que varía t. A partir de estas definiciones se formula la prueba de significancia de la siguiente forma:

- Si $t \in C$, rechazar H_0
- Si $t \notin C$, no rechazar H_0

Por ejemplo, un experimentador observa la cantidad de defectos detectados por LOC de una técnica de verificación desconocida bajo determinadas condiciones, y quiere probar que no es la técnica B, de la cual sabe que en las mismas condiciones (programa, verificador, etc.) detecta 1 defecto cada 20 LOC. El experimentador sabe que también pueden haber otras técnicas que detecten 1 defecto cada 20 LOC. A partir de esto se define la hipótesis nula: "H_0: La técnica observada es la B". En este ejemplo, la unidad de prueba t es cada cuantos LOC se detecta un defecto y el área crítica es $C = \{1, 2, 3, \ldots, 19, 21, 22, \ldots\}$. La prueba de significancia es: si $t \leq 19$ o $t \geq 21$, rechazar H_0, de lo contrario no rechazar H_0.

Si se observa que $t = 20$, la hipótesis no puede ser rechazada ni se pueden derivar conclusiones, pues pueden haber otras técnicas que detecten un defecto cada 20 LOC.

El área crítica, C, puede tener distintas formas, lo más común es que tenga forma de intervalo, por ejemplo: $t \leq a$ o $t \geq b$. Si C consiste en uno de estos intervalos es unilateral. Si consiste de dos intervalos ($t \leq a$, $t \geq b$, donde $a < b$), es bilateral.

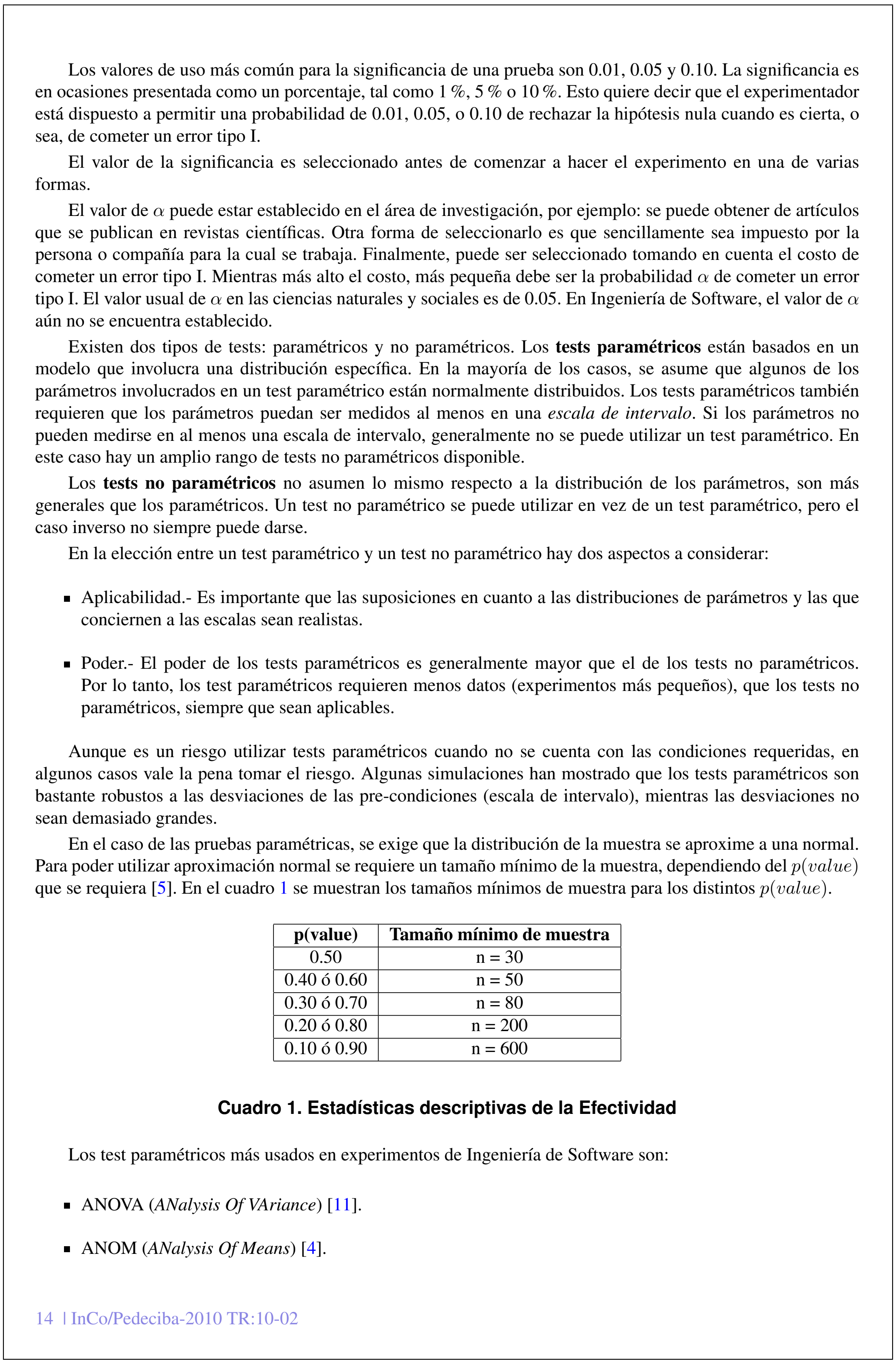
Hay varios métodos estadísticos, de aquí en adelante denotados *tests*, que pueden utilizarse para evaluar los resultados de un experimento, más específicamente para determinar si se rechaza la hipótesis nula. Cuando se lleva a cabo un *test* es posible calcular el menor valor de significancia posible (denotado p-valor) con el cual es posible rechazar la hipótesis nula. Se rechaza la hipótesis nula si el p-valor asociado al resultado observado es menor o igual que el nivel de significancia establecido.

Las siguientes son tres probabilidades importantes para la prueba de hipótesis:

- α = P(cometer el error tipo I) = P(rechazar $H_0|H_0$ es verdadera). Es la probabilidad de rechazar H_0 cuando es verdadera.
- β = P(cometer el error tipo II) = P(no rechazar $H_0|H_0$ es falsa). Es la probabilidad de no rechazar H_0 cuando es falsa.
- Poder = $1 - \beta$ = P(rechazar $H_0|H_0$ es falsa). El poder de prueba es la probabilidad de rechazar H_0 cuando es falsa.

El experimentador debería elegir un test con un poder de prueba tan alto como sea posible. Hay varios factores que afectan el poder de un test. Primero, el test en sí mismo puede ser más o menos efectivo. Segundo, la cantidad de muestras: mayor cantidad de muestras equivale a un poder de prueba más alto. Otro aspecto es la selección de una hipótesis alternativa unilaterial o bilateral. Una hipótesis unilateral da un poder mayor que una bilateral.

La probabilidad de cometer un error tipo I se puede controlar y reducir. Si la probabilidad es muy pequeña, sólo se rechazará la hipótesis nula si se obtiene evidencia muy contundente en contra de esta hipótesis. La probabilidad máxima de cometer un error tipo I se conoce como la significancia de la prueba (α).



Los valores de uso más común para la significancia de una prueba son 0.01, 0.05 y 0.10. La significancia es en ocasiones presentada como un porcentaje, tal como 1 %, 5 % o 10 %. Esto quiere decir que el experimentador está dispuesto a permitir una probabilidad de 0.01, 0.05, o 0.10 de rechazar la hipótesis nula cuando es cierta, o sea, de cometer un error tipo I.

El valor de la significancia es seleccionado antes de comenzar a hacer el experimento en una de varias formas.

El valor de α puede estar establecido en el área de investigación, por ejemplo: se puede obtener de artículos que se publican en revistas científicas. Otra forma de seleccionarlo es que sencillamente sea impuesto por la persona o compañía para la cual se trabaja. Finalmente, puede ser seleccionado tomando en cuenta el costo de cometer un error tipo I. Mientras más alto el costo, más pequeña debe ser la probabilidad α de cometer un error tipo I. El valor usual de α en las ciencias naturales y sociales es de 0.05. En Ingeniería de Software, el valor de α aún no se encuentra establecido.

Existen dos tipos de tests: paramétricos y no paramétricos. Los **tests paramétricos** están basados en un modelo que involucra una distribución específica. En la mayoría de los casos, se asume que algunos de los parámetros involucrados en un test paramétrico están normalmente distribuidos. Los tests paramétricos también requieren que los parámetros puedan ser medidos al menos en una *escala de intervalo*. Si los parámetros no pueden medirse en al menos una escala de intervalo, generalmente no se puede utilizar un test paramétrico. En este caso hay un amplio rango de tests no paramétricos disponible.

Los **tests no paramétricos** no asumen lo mismo respecto a la distribución de los parámetros, son más generales que los paramétricos. Un test no paramétrico se puede utilizar en vez de un test paramétrico, pero el caso inverso no siempre puede darse.

En la elección entre un test paramétrico y un test no paramétrico hay dos aspectos a considerar:

- Aplicabilidad.- Es importante que las suposiciones en cuanto a las distribuciones de parámetros y las que conciernen a las escalas sean realistas.

- Poder.- El poder de los tests paramétricos es generalmente mayor que el de los tests no paramétricos. Por lo tanto, los test paramétricos requieren menos datos (experimentos más pequeños), que los tests no paramétricos, siempre que sean aplicables.

Aunque es un riesgo utilizar tests paramétricos cuando no se cuenta con las condiciones requeridas, en algunos casos vale la pena tomar el riesgo. Algunas simulaciones han mostrado que los tests paramétricos son bastante robustos a las desviaciones de las pre-condiciones (escala de intervalo), mientras las desviaciones no sean demasiado grandes.

En el caso de las pruebas paramétricas, se exige que la distribución de la muestra se aproxime a una normal. Para poder utilizar aproximación normal se requiere un tamaño mínimo de la muestra, dependiendo del $p(value)$ que se requiera [5]. En el cuadro 1 se muestran los tamaños mínimos de muestra para los distintos $p(value)$.

p(value)	**Tamaño mínimo de muestra**
0.50	n = 30
0.40 ó 0.60	n = 50
0.30 ó 0.70	n = 80
0.20 ó 0.80	n = 200
0.10 ó 0.90	n = 600

Cuadro 1. Estadísticas descriptivas de la Efectividad

Los test paramétricos más usados en experimentos de Ingeniería de Software son:

- ANOVA (*ANalysis Of VAriance*) [11].

- ANOM (*ANalysis Of Means*) [4].

Ambos tests (ANOVA y ANOM), pueden utilizarse para diseños de un solo factor con múltiples alternativas. En ambos test la hipótesis nula refiere a la igualdad de las medias (como es habitual en los test paramétricos):

$H_0 : \bar{x}_1 = \bar{x}_2 = \ldots = \bar{x}_I$

En ANOVA, la variación en la respuesta se divide en la variación entre los diferentes niveles del factor (los diferentes tratamientos) y la variación entre individuos dentro de cada nivel. El objetivo principal del ANOVA es contrastar si existen diferencias entre las diferentes medias de los niveles de las variables (factores).

En el caso de ANOM, este test no solamente responde a la pregunta de si hay o no diferencias entre las alternativas, sino que cuando hay diferencias, también dice cuáles alternativas son mejores y cuáles peores.

Los test no paramétricos más usado son:

- Kruskal Wallis.
- Mann-Whitney.

En el caso de los test no paramétricos, la hipótesis nula refiere a la igualdad de las medianas:

$H_0 : \tilde{x}_1 = \tilde{x}_2 = \ldots = \tilde{x}_I$

Rechazar H_0 significa que existe evidencia estadística como para afirmar de que hay diferencias entre las alternativas. En el caso de que hubiera más de dos alternativas, para conocer cuál es la alternativa que difiere es necesario comparar las alternativas de a dos.

En el caso de Kruskal Wallis, a pesar de no requerir una distribución normal para las muestras, sus resultados se pueden ver afectados por lo que se le llama «heterocedasticidad» de los datos. Cuando una muestra presenta datos heterocedásticos (no presentan homocedasticidad) el test de Kruskal Wallis podría dar un resultado no significativo (no rechazando H_0), aunque haya una diferencia real entre las muestras (debería rechazar H_0).

Para probar la homocedasticidad de los datos se suele utilizar el test de Levene. Las hipótesis del test de Levene son:

- $H_0 : \sigma_1 = \sigma_2 = \ldots = \sigma_k$ donde σ_a es la varianza de la muestra a.
- $H1 : \sigma_i \neq \sigma_j = \ldots = \sigma_k$ para al menos un par de muestras (i, j), donde σ_a es la varianza de la muestra a.

Para poder aplicar ANOVA, y en algunos casos Kruskal-Wallis, es necesario que el test de Levene no sea significativo (no se rechaza H_0), o sea, que las varianzas de las muestras sean similares o iguales. Esto prueba la homocedasticidad de los datos.

Una vez que se prueba que al menos dos de las k muestras provienen de poblaciones distintas (datos heterocedásticos) se puede aplicar, entre otros, el test de Mann-Whitney para comparar las muestras dos a dos.

Si se presume que una alternativa puede ser mejor o peor que el resto, esto quiere decir que hay un «ordenamiento» entre ellas, lo aconsejable es realizar un test de ordenamiento. Algunos test de ordenamiento son:

- Jonckheere-Terpstra Test. [3]
- Test para alternativas ordenadas L. [3]

Para los test de ordenamiento, las hipótesis que se plantean son las siguientes:

$H_0 : \tilde{x}_1 = \tilde{x}_2 = \ldots = \tilde{x}_I$

$H_1 : \tilde{x}_1 \leq \tilde{x}_2 \leq \ldots \leq \tilde{x}_I$ (con al menos una desigualdad estricta)

4.6. Presentación y Empaquetado

En la presentación y el empaquetado de un experimento es esencial no olvidar aspectos e información necesaria para que otros puedan replicar o tomar ventaja del experimento y del conocimiento ganado durante su ejecución.

El esquema de reporte de un experimento generalmente cuenta con los siguientes títulos: Introducción, Definición del Problema, Planificación del Experimento, Operación del Experimento, Análisis de Datos, Interpretación de los Resultados, Discusión y Conclusiones, y Apéndice.

En la *Introducción* se realiza una introducción al área y los objetivos de la investigación. En la *Definición del Problema* se describe en mayor profundidad el trasfondo de la investigación, incluyendo las razones para realizarla. En la *Planificación del Experimento* se detalla el contexto del experimento incluyendo las hipótesis, que se derivan de la definición del problema, las variables que se deben medir (tanto independientes como dependientes), la estrategia de medida y análisis de datos, los sujetos que participaran de la investigación y las amenazas a la validez.

En la *Operación del Experimento* se describe como preparar la ejecución del mismo, incluyendo aspectos que permitan facilitar la replicación y descripciones que indiquen cómo se llevaron a cabo las actividades. Debe incluirse la preparación de los sujetos, cómo se recolectaron los datos y cómo se realizó la ejecución.

En el *Análisis de Datos* se describen los cálculos y los modelos de análisis específicos utilizados. Se debe incluir información, como por ejemplo, tamaño de la muestra, niveles de significancia y métodos estadísticos utilizados, para que el lector conozca los pre-requisitos para el análisis. En la *Interpretación de los Resultados* se rechaza la hipótesis nula o se concluye que no puede ser rechazada. Aquí se resume cómo utilizar los datos obtenidos en el experimento. La interpretación debe realizarse haciendo referencia a la validez. También se deben describir los factores que puedan tener un impacto sobre los resultados.

Finalmente, en *Discusión y Conclusiones* se presentan las conclusiones y los hallazgos como un resumen de todo el experimento, junto con los resultados, problemas y desviaciones respecto al plan. También se incluyen ideas sobre trabajos a futuro. Los resultados deberían ser comparados con los obtenidos por trabajos anteriores, de manera de identificar similitudes y diferencias. La información que no es vital para la presentación se incluye en el Apéndice. Esto puede ser, por ejemplo, los datos recavados y más información sobre sujetos y objetos. Si la intención es generar un paquete de laboratorio, el material utilizado en el experimento puede ser proveído en el apéndice.

Referencias

[1] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology, February 1998. 1

[2] N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, 2001. 1

[3] E. J. Martínez. Notas del curso de posgrado maestría en estadística matemática. Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2004. 4.5.3

[4] P. Nelson, M. Coffin, and K. Copeland. *Introductory statistics for engineering experimentation*. Elsevier Science, California, 2003. 4.5.3

[5] M. Spiegel. *Estadística - 2da Edición*. Mc.Graw-Hill, Madrid, 1991. 4.5.3

[6] J. Triñanes. Construcción de un banco de pruebas de modelos de proceso. In *Jornadas Iberoamericanas de Ingeniería de Software e Ingeniería del Conocimiento*, 2004. 1

[7] D. Vallespir, C. Apa, S. De León, R. Robaina, and J. Herbert. Effectiveness of five verification techniques. In IEEE-Computer-Society, editor, *Proceedings of the International Conference of the Chilean Computer Society*, 2009. 1

[8] D. Vallespir, F. Grazioli, and J. Herbert. A framework to evaluate defect taxonomies. In *Proceedings of the XV Argentine Congress on Computer Science*, 2009. 1

[9] D. Vallespir and J. Herbert. Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. In IEEE-Computer-Society, editor, *Proceedings of the Mexican International Conference in Computer Science*, 2009. 1

[10] 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. 1

[11] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. 1, 4.5.3