

**UNIVERSITA' DEGLI STUDI DI ROMA
TOR VERGATA**



**DIPARTIMENTO DI INFORMATICA,
SISTEMI E PRODUZIONE**

**CORSO DI DOTTORATO DI RICERCA IN
INFORMATICA E INGEGNERIA
DELL'AUTOMAZIONE
XVII CICLO**

Building a Body of Knowledge from Families of Software Empirical Studies:

**From Statistical Analysis to Cluster-impact Approach for
Manipulating Experimental Data Using Fuzzy Sets**

Candidate

Zeiad A. Abdelnabi

Advisor

Professor Giovanni Cantone

Anno Accademico 2005/2006

Evaluation Committee

The present dissertation has been submitted for official evaluation to the following scientists:

Professor: Gerardo Canfora
Università del sannio
Benevento - Italy

Professor: Dieter Rombach
University of Kaiserslautern
Kaiserslautern – Germany

Professor: Giorgio Ventre
Università Nabolì “Federico II”
Napoli - Italy

Professor: Daniel Pierre Bovet
Coordinatore del Dottorato
Università di Roma “Tor Vergata”
Rome - Italy

Acknowledgment

I would like to thank all people who contributed to this work, and to those people who helped me during my doctoral studies...

- **Professor Danele Bovet**
- **Professor Giovanni Cantone**
- **Professor H. Dieter Rombach**

- **Davide Falessi**
- **Davide Pace**
- **Luca Colasanti**
- **Mario Pio**
- **Sandra Ciliberti**
- **Marcus Ciolkowski**
- **Marcus Trab**
- **Ralf Carb**

Abstract

The explosive growth of the software industry in recent years has focused the attention on the problems long associated with software development.

Empirical evaluations in software engineering are important for building a body of knowledge, which is missing for several software engineering areas nowadays. Transferring such body of knowledge to industry is one of the main objectives in research.

In order to empirically achieve such an objective, it is our conviction that empirical studies should be considered from two viewpoints: one comes from the effect, the outcomes from as explicit as specific input variables or factors in traditional experimentation, where we will be able to provide studies with an initial benchmark of knowledge of software (“Effect Viewpoint”), while the other comes from the integral cause, where we will be able to understand empirical software studies as an integrated body of knowledge, based on the analysis of all possible input variables - including factors, parameters, and blocking variables - through families of experiments and other empirical studies (“Cause Viewpoint”). Along with current study, we present both Effect and Cause viewpoints.

Current dissertation is an attempt to study the state of the art of experimentation in software engineering, aiming to understand variables that cause effects when empirical investigations are targeting to Object Oriented (OO) software applications and utilities, including OO software frameworks, which are designed and adopted from a set of OO applications in a specific domain.

In the first part of current dissertation, we focus on the Effect Viewpoint; in particular, rules of empiricism are applied for investigating software-reading techniques. Two different, but related in goal, empirical packages are considered; the first package aims at evaluating the effectiveness of reading techniques for defect detection in OO C++ software frameworks, while the last package aims at evaluating the effectiveness of software testing strategies (static and dynamic techniques) for defect detection in OO event driven Java software.

In the second part of current dissertation, we focus on the Cause Viewpoint. An approach, so called Cluster-impact Approach, is defined and eventually implemented, which spotlights on sets of variables that impact on out-coming responses when conducting empirical studies. With Cluster-impact Approach, collected empirical data from families of experiments (i.e. one or more experiments,

each with one or more replications), can be significantly schematized and eventually engaged into a process that aims to build a body of knowledge for any OO software. Such a process is based on integration and formalization mechanisms: Fuzzy sets are used for transforming data from being quantitative to qualitative; transformed data are analyzed qualitatively for drawing the study conclusions.

Contents

Abstract

Acknowledgments

Contents

Part I: Empiricism In software Engineering

CHAPTER 1. Introduction

1.1. Background

1.2. An outlook

1.3. Object oriented software

1.4. Software frameworks

1.4.1. Testing software frameworks

1.4.2. Challenges in testing software frameworks

CHAPTER 2. Experimentation on Object Oriented Software Frameworks

2.1. Introduction

2.2. Related work

2.2.1. Framework understanding

2.2.2. Framework reading

2.3. Study motivations and view

2.4. Framework architecture

2.5. Functionality-based approach for framework understanding

2.5.1. From software framework artifacts to functionalities

2.5.2. Functionality types and relations

2.5.3. How to extract and characterize functionalities, and formalize understandings: Functionality rules

CHAPTER 3. Experimentation on Object Oriented Software Frameworks

3.1. Experiment planning and operation

3.1.1. Hypothesis formulation

- 3.1.2. Variables selection
- 3.1.3. Subjects
- 3.1.4. Objects
- 3.1.5. Materials and infrastructures
- 3.2. *Experiment design*
 - 3.2.1. Defects
 - 3.2.2. Experiment training and operation
- 3.3. *Experiment results and data analysis*
 - 3.3.1. Descriptive statistics
 - 3.3.2. Hypothesis testing
- 3.4. *Discussion*
- 3.5. *Threats to validity*
 - 3.5.1. Subjects
 - 3.5.2. Training session
 - 3.5.3. Materials
 - 3.5.4. Seeded defects
- 3.6. *Conclusions*

CHAPTER 4. An empirical study on object oriented software frameworks

- 4.1. *Experiment planning and operation*
 - 4.1.1. Hypothesis formulation
 - 4.1.2. Variables selection
 - 4.1.3. Subjects
 - 4.1.4. Objects
 - 4.1.5. Materials and infra structure
 - 4.1.6. Experiment design
 - 4.1.7. Defects
 - 4.1.8. Experiment training and operation
- 4.2. *Experiment results and data analysis*
 - 4.2.1. *Descriptive statistics and hypothesis tests*
 - 4.2.2. *Analyses of EPIC*
 - 4.2.3. *Analysis of overall effectiveness*
- 4.3. *Discussion*
 - 4.3.1. *Techniques' effectiveness*
 - 4.3.2. *Threats to validity*
- 4.4. *Conclusions*

CHAPTER 5. More Empirical Studies on Object Oriented Software Reading for Defects

5.1. Introduction

5.2. Starting Scenario

5.2.1. Method

5.2.2. Design

5.2.3. Subjects/Participants

5.2.4. Apparatus/Materials

5.2.5. Procedure

5.3. Results

5.4. Discussion

5.4.1. Inspecting time

5.4.2. Subject performance

5.4.3. Types of faults

5.4.4. Threats to the experiment validity

5.5. Lessons learned

Summary and discussion of Part I

Part II: Strategies for Manipulating Empirical Data

CHAPTER 6. Cluster-impact: An Approach for Manipulating Empirical Data with Fuzzy Sets

6.1. Introduction

6.1.1. Related work

6.1.2. Study motivations

6.1.3. Study view and goal

6.2. Understanding cluster-based factors

6.2.1. Cluster-impact approach

6.3. Fuzzy sets for data manipulation

6.3.1. Membership grades of Fuzzy sets

6.3.2. Fuzzy membership function

6.3.3. Evaluation of Fuzzy membership grades

6.4. Organizing experiment-based study in software inspection techniques for defect detection

6.4.1. Definition planning and design

6.4.2. Operating with Fuzzy sets

CHAPTER 7. Cluster-impact Evaluation on the Effectiveness of Reading Techniques for Defect Detection

7.1. Introduction

7.2. Building a Body of knowledge from families of empirical studies

7.2.1. Study focus

7.2.2. Infrastructure

7.2.3. Factors and variables

7.2.4. Study materials

7.3. Preparation and data management

7.3.1. Membership grads of Fuzzy sets

7.4. Manipulation of Fuzzy sets

7.4.1. Impact on Focus Interaction (ICI)

7.4.2. Impact on Factor Interaction (IFI)

7.5. Formulizing results

Summary and discussion of Part II

CHAPTER 8. Conclusions and Lessons Learned

CHAPTER 9. Appendices

CHAPTER 10. References

Part I

Empiricism in Software Engineering

Chapter 1

Introduction

1.1. Background

When developing software, everyone in the development team should be involved in implementing quality in each of the development phases they should not do their tasks without thinking about the quality, while assuming someone else along the development process will verify it.

Software testing is a process of uncovering evidence of defects in software systems. However, the evaluation of the work products created during a software development effort is more general than just checking part or all of a software system to see if it meets its specifications. In fact, developing quality of software is a complex and very expensive task, and so, software managers may avoid quality improvement processes such as design reviews and code inspections, believing that these processes only add time to the development cycle. Even, those software managers, who consider the use of software verification, are still looking for the most productive verification mechanism that insures an acceptable level of software quality. In fact, the need to know what mix of verification strategies to follow (i.e. static, dynamic) and the most productive verification approach to perform for certain types of defects (i.e. white box and black box testing, checklist driven inspection etc.) have become a serious challenge to research and practice in software engineering; researchers are still incompletely conscious of what and when methods, techniques, and tools for software improvement are significantly effective.

Defects can be introduced during any phase of the software development life cycle and results from one or more mistakes ("bugs"), misunderstandings, omissions, or even misguided intent on the part of the developers. Verification aims to understand if we are doing things right, hence comprises the efforts to find defects. Verification techniques, in their turn, are associated with approaches for detecting, locating, identifying, and tracing defects through some kinds of software artifacts. At

the end point, software verification includes classifying software defects, defining causes, and relating those defects each other.

Testing is the verification technique mostly used in practice. There are two main approaches for software to be tested, black box testing and white box testing; each of those approaches has its own characteristics, advantages and disadvantages. One of the main challenges in software engineering research nowadays is the lack of enough formal base of knowledge concerning software verification for defect detection; while researches in software engineering have established several techniques, methods, and tools for testing software [63], black-box and white-box testing included, the question still remains concerning "what" and "when" to use among those testing techniques, tools and methods. Experiments are one means to questions like those ones, and eventually evolve software quality.

Experimental research in software engineering is a promising paradigm for evaluating the feasibility, productivity, and maturity of software methods, techniques and tool, including verification techniques. Controlled experiments also help in understanding software; this tends to significant packages of knowledge concerning variables that cause effects when verifying software. In our understanding, what we really need in software engineering research is a mechanism for gathering empirical data and building a reusable body of software knowledge; such a base of knowledge may contribute to a better understand of the related effects when verifying software.

Organizing several packages of related empirical studies may facilitate building knowledge in an incremental, evolutionary manner, through the replication of experiments within families of studies [9]. A body of software knowledge is a framework that mainly depends on the credibility, reliability, and reusability of the collected empirical data rather than the number of empirical packages that we might involve in the study; in other words, collecting data with low quality cannot be recovered by increasing the number of empirical packages in the framework.

The awareness of the importance of replicating studies has been growing in the software engineering community. In software engineering studies, one way to build a strong base of software knowledge is to rely on an acceptable level of certainty; this can be achieved by emphasizing on the meaning of replicated experiments [45]; the results of any one study cannot simply be extrapolated to all environments because there are many uncontrollable sources of variation between different environments. With empirical framework packages of replicated studies, experiments can be viewed as part of common families of studies, rather than being isolated events. Common families of studies can contribute to important and relevant hypotheses that may not be suggested by individual experiments [9].

The present dissertation is an investigation on experimental software engineering for the aim of understanding the complex of variables that cause effects when empirical investigations are targeting to OO software artifacts, including software frameworks, which are reusable software artifacts that are designed and adopted from a set of OO applications in a specific domain [52].

Moreover, based on the experience that we gained during our doctoral studies while assisting in, and eventually conducting, several experiments, we figured that empirical studies should be considered from two viewpoints (see Figure 1). The one comes from the effect, i.e. outcomes from, as explicit as specific, input variables (or Factors) in traditional experimentation, where we are able to provide studies with an initial benchmark of knowledge of software (*“Effect Viewpoint”*, in our notation). The other one comes from the integral cause of that effect, where we are able to understand empirical studies as an integrated body of software knowledge, based on the analysis of all possible input variables - including factors, parameters, blocking variables etc. - through families of experiments and other empirical studies (*“Cause Viewpoint”*, in our notation). Let us note that the former viewpoint is stricter and conclusive than the latter, while the latter is more flexible, reusable and explorative than the former. Along with current study, we present both effect and cause viewpoints.

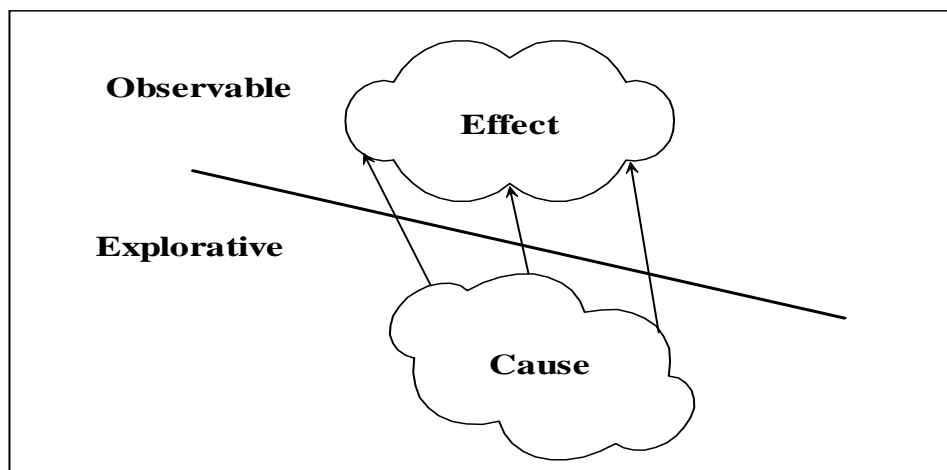


Figure 1:Effect \ Cause viewpoints

In the first part of the present dissertation, we focus on the Effect Viewpoint; rules of empiricism are used to provide artificial reality for experimenting with software verification processes. Two different, but related in goal, empirical packages are considered. One package aims at evaluating and comparing one each other the effectiveness of some reading techniques for defect detection in OO C++ software frameworks. One more package aims at evaluating the effectiveness of software

verification strategies (e.g. reading vs. dynamic testing) and techniques (in particular, checklist base code-reading versus checklist driven functional testing) for defect detection in object-oriented event driven Java software.

In the last part of present dissertation, we focus on the Cause Viewpoint. An approach, so called Cluster-impact Approach, is defined and eventually implemented which spotlights on variables that cause effects when conducting empirical studies. With Cluster-impact Approach, collected empirical data from families of experiments (i.e. one or more experiments, each with one or more replications), can be significantly schematized and eventually engaged into a process that aims to build a body of knowledge for any OO software. The process that we propose for building a body of knowledge is based on integration and formalization mechanisms; Fuzzy sets are utilized for transforming data from being quantitative to qualitative. Eventually, transformed data can be analyzed qualitatively for drawing the study conclusions.

1.2. An outlook

This dissertation is organized in two parts and conclusive remarks.

Part I consists of five chapters. Chapter 1 is an introduction to testing approaches; in particular, black box and white box verification techniques for OO software frameworks with a discussion concerning what is the mix of strategies to use, and why, when testing a software framework. Chapter 2 considers an experiment design for a package of experiments that investigates on software OO software frameworks. Chapters 3 and 4 present the packaging of two experiments, which compared the effectiveness of three reading techniques for defect detection in C++ OO software frameworks. Chapter 5 is concerned with an empirical package from experiments that compared the effectiveness of functional testing versus checklist-based code-reading for defect detection in OO event driven Java software. We shortly summarize upon the experiments that we conducted (Effect Viewpoint), and eventually put the main question of the next part of the present dissertation.

Part II consists of three chapters. Chapter 6 emphasizes on Fuzzy sets as a method for manipulating the empirical data in software engineering; moreover, experimentation process in software engineering is discussed; finally, Cluster-impact approach is presented as an approximation method between Fuzzy set based analysis and data coming from traditional empirical studies in software engineering. Chapter 7 presents the manipulation of Fuzzy sets when applied to problems associated with software verification for defect detection. We summarize Cluster-

impact Approach to software reading for defect detection in 10 formal results, with an evaluation of their significance. Finally, Chapter 8 concludes upon the study with further details concerning lessons learned and future studies.

1.3 Object oriented software

The adoption of OO technologies brings changes not only in the programming languages we use, but also in most aspects of software development. OO features, like inheritance and polymorphism, present new technical challenges to software testers; in fact, specifically for software that are analyzed and constructed by using object orientation, the sooner problems are found the cheaper they are to fix [37].

The most significant difference between software paradigms is in the way software is think, analyzed, designed, programmed and tested, both for construction and maintenance. OO software is a set of objects that essentially model a problem and then collaborate to place in effect a solution [38]. The differences between "old" and "new" systems of software developing and testing are much deeper than just a focus (on objects instead of functions), the most significant difference is in the method itself. Underlying this approach is the concept that while a solution to a problem (control) and its presentation and storage might need to change over time, the architecture, basic entities, and components of the problem itself does not change as much or as frequently; in fact, this concept is true for many of the problems that we have to solve by using software. However, OO places its own challenges to software people, from analysts to testers. In particular, those flexible and reusable OO packages, which have quite unlimited number of use cases for one application domain (e.g. software frameworks) present new challenges to software testers.

In fact, an OO application, which solution is structured from a specific problem, can be tested by adopting test cases from the specific problem domain, and extending those tests based on the particular changes enacted. Vice versa, when a software a reusable but incomplete software kernel (or "framework") is structured from adopting a set of other software most frequent "solutions" in the application domain, testers would be in the front of big challenges; that is because though that each adopted software application in the framework had a limited number of use cases, some of these use cases are related to other use cases, from different/same application, in the framework; moreover, the number of use cases is expected to increase when new adoptions take place in the software framework.

1.4. Software frameworks

OO frameworks are artifacts that are intended for reuse and extensibility [57] in a specific software application domain. Frameworks are widely used in the advanced software industry but not discussed much by the software engineering community research [39]. After the initial frameworks for graphical applications, frameworks were developed for many other domains, including air-space traffic control [52], telecommunication [36], and data management, e.g. systems such as OLE, OpenDoc, Java AWT and Beans, and .NET by Microsoft [39]. There are two types of frameworks with respect to their characteristics: White-box frameworks, which are tailored by adding specialized classes and methods in a hierarchical flow; and Black-box frameworks, which provide a set of application-specific components that are plugged together through composition [59].

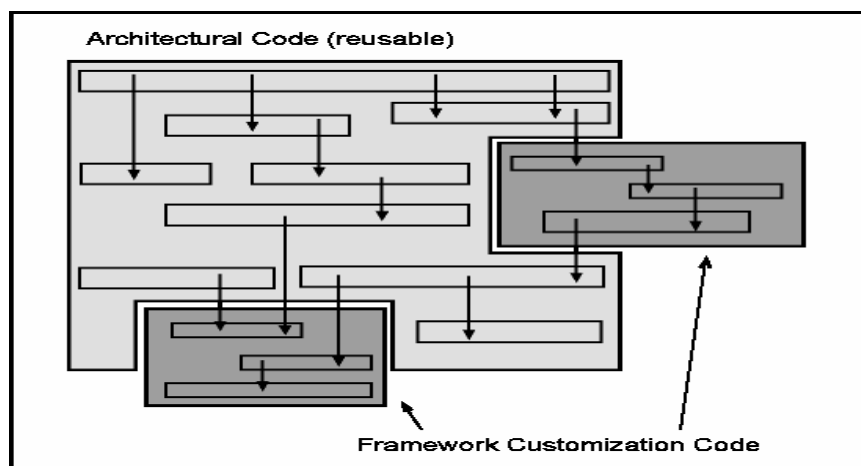


Figure 2: software frameworks

As shown in Figure 2, frameworks are generic applications that are extracted from, and abstracted on, an application domain, allow creating different functionalities from a family of applications; recently, frameworks are designed and developed by using Object Orientation approach. When designing a framework, the designer identifies, in C++ terms, certain methods of certain key (not final) classes as virtual methods or pure virtual methods (abstract methods or classes, or interfaces, in Java terms); application developers provide those methods implementation (customization). The main task of a developer who wishes to build an application using frameworks, therefore, is to provide instantiations for the pure virtual methods, and possibly override some virtual methods of the framework [28].

1.4.1. Testing software frameworks

As we already mentioned, a framework is a special type of software, which defines and implements the common parts of a family of applications, while leaving abstract (or providing standard, quite simple, implementations of) those parts that are application specific. In fact, frameworks define the basic components in the application and the way they interact. In practice, this means that a framework, as a software product, is not complete in the sense that it cannot be executed (less than in standard, quite simple, implementation). Frameworks are software that defines the abstract design of a concept with their possible interactions; the application-specific details of these concepts are not implemented as part of the framework, and will be delivered only later by the user; this means that frameworks cannot perform any meaningful task by themselves; the user of the framework is required to implement framework interfaces and, usually, some concrete classes to create a complete application.

According to the previous discussion, there are two main viewpoints through which we can see software frameworks: the framework producer, who defines the overall design and flow of the application that will use the framework; and the framework user, who has to supply the “missing details” that define his\her concrete application. Practically for some cases, a framework can be considered as software final product, from the point of view of the framework developer, as software “prime material”, from the point of view of the framework customer; that is, end-users will use frameworks to create their own final products; suppliers in this case are delivering the framework as their own product. Another case is that suppliers create frameworks as an in-house mechanism for reusing software; the internal user of the framework gets a fully functional application. However suppliers explain that a family of products could be generalized as a framework.

In both those cases, the quality of the framework (as a complete product or as an intermediate product) should be verified. Whether the clients of the framework are external customers or merely developers in the same organization, the added value in reusing the design and generic implementation encapsulated in it will be meaningless if the framework is of poor quality; this brings us to the challenge of defining and verifying the quality of a framework.

The quality of a software product is made out of several aspects; functional aspect of the product is merely one of them; the quality of the software artifacts, including source code, is one of the aspects in the overall quality of the software

product. However, even when concentrating on the functional aspect, we face quite a challenge when the software product is a framework.

When testing stand-alone applications, almost every aspect of the product's functionality is defined a-priori, and so, the activity of testing the product is in fact made of a set of test scenarios, each with an expected outcome. Testing a library has the same meaning; that is, the behavior of a library is always predefined; this would predefine the activity of testing those libraries as well; moreover, predefined testing of those libraries would keep the focus on the functionalities of the product by default, as it should be. However, when considering a framework, only some aspects of its functionality are defined a-priori.

The goal of any framework is to enable the implementation of different applications each with some specific functionalities; i.e. capabilities and behaviors. Isolating the functionality of the framework for testing purposes is not a trivial task; when considering frameworks' functionalities, we face three different types of functionalities with respect to what they do [52]:

- **Do-functionalities, DF:** describe fully implemented capabilities of the framework that every instantiated application must have.
- **Can-functionalities, CF:** describe not fully implemented capabilities that the framework has. They present customization points where the framework users can plug-in their specific code.
- **Offer-functionalities, OF:** describe fully implementation of some behaviors that the framework offers to the users.

1.4.2. Challenges in testing software frameworks

By default, users of the framework treat its functionalities based on the classification above, as a first step, users initialize the essential "parts" of the framework by instantiating some DO functionalities; some OFFER functionalities are eventually instantiated based on convenience; and finally, several interfaces or abstract classes and methods are implemented by the users in order to "hook" their applications to the framework.

Similarly, the tester of a framework should proceed with the same steps when the attempt is to dynamically test the framework. The challenge lies in deciding how to fill CAN functionalities and with what; of course, it is generally impossible to cover, and quite impossible to get equivalent samples of, all possible implementations for such non-implemented functionalities in the framework.

Another challenge in testing frameworks lies in the fact that the user of the framework is highly affected from its design and definition. The focus of these issues is the usage of the product. When the product is a framework, its usage is tightly coupled with its design.

Therefore, if we aim to verify the quality of the framework from the user's perspective, we cannot settle only for verifying its narrow functional aspects. We must put ourselves in the position of the user who will use the framework, in particular, framework testers who will strongly need to understand the framework functionalities before implementing the proper test cases for testing each of those functionalities, especially when frameworks are complicated by the fact that frameworks have associated learning problems that affect their usefulness [59]; this would make us think that:

- 1- It is useful but not enough to test the framework dynamically, by using stubs or default functionalities.
- 2- The framework verifier should be involved with a process of code reading while implementing CAN functionalities with the proper use cases.
- 3- Understanding the structural design and implementations of the framework is essential for verifying its quality, including verification for defects.
- 4- In general, for any OO software, and in particular for any software which is open for reuse, it is not easy to include all possible use cases when dynamically testing software applications.

Chapter 2

Experimentation on Object Oriented Software Frameworks

2.1. Introduction

Techniques for detecting defects in code are fundamental to the success of any software development approach. A software development organization therefore needs to understand the utility of techniques such as reading or testing in its own environment. Controlled experiments are one means to evaluate software engineering techniques and gaining the necessary understanding about their utility [45]. Benefits of the Object-Oriented (OO) paradigm are widely involved in the architectural design of many software development methods; software frameworks are widely used in the advanced software industry; however, the software engineering community researches are slightly concerned with frameworks [39].

2.2. Related work

Several empirical studies exist that related to understanding software code [66], and reading software code for defect detection ([4], [42], [19], [49], [29]). However, few studies have been conducted so far specifically on software frameworks and most of them are related to building and designing frameworks rather than to understanding and reading for defect detection.

2.2.1. Framework understanding

The Empirical Software Engineering Group at the University of Maryland's Department of Computer Science (UMDCS ESEG) [59] conducted experimental work with novice subjects, which related to understanding frameworks for their usage in software construction. Two techniques were compared to each other: (i) the Example-based technique [57], focusing on understanding the framework dynamically through a set of example applications. These examples, taken together,

are meant to illustrate the range of functionality and behavior provided by the framework; however, such a technique can only provide examples for a limited number of the framework functionalities (i.e., only functionalities that cope with user-interfaces) [59]; (ii) the Hierarchy-based technique, which employs a step-wise approach to understanding the framework functionalities by concentrating on high-level abstract classes first, proceeding to derived classes later, and so on. The Example-based technique was identified as an effective strategy for the environment of the UMDCS ESEG experiment. However, that ESE Group was unable to provide examples for a complete coverage of the framework functionalities. The Hierarchy-based technique was not deemed effective in that environment, and was found to be very complex to apply. Finally, based on the experiment results, it was not possible for UMDCS ESEG to assume that the Hierarchy-based technique was always inferior.

2.2.2. Framework reading

Experimental studies have been conducted on reading techniques already, but those concern general OO software ([27], [19]) rather than frameworks. For instance, Dunsmore [27] used objects from a Java application, and applied three code reading techniques: Checklist-based Reading (CBR), Use case-driven Reading (UCDR), and Systematic Order-based Reading (SBR). Results showed that those techniques significantly differ with respect to the number of defects found; in particular, CBR was most effective. However, the other two techniques also have strengths. Moreover, SBR shows a significant potential for providing help in navigating code, which is strongly de-localized in many derived classes in OO software [27]. Consequently, in order to get the best results in a practical situation, Dunsmore recommended using a combination of all three reading techniques. Reading techniques for OO software were also compared in Rome Tor Vergata, where experiments were conducted both for UML analysis and design documents [18], and Java OO event-driven software [16]; while, in both cases, noticeable differences were seen between the applied techniques, some results were not statistically significant, and further studies are planned. No experiment has been applied to code reading of Object-oriented Frameworks for defect detection before of us, to the best of our knowledge. We started with a preliminary homework that we assigned to limited number of students of Rome Tor Vergata aiming to test the experiment process and materials [56]. Hence, we extended [2] the work of Dunsmore by investigating causes of variation among reading techniques, once applied for OO C++ frameworks. First, the

University of Rome Tor Vergata ESE group (URMTV ESEG) provided, the University of Kaiserslautern, Germany (UKI) validated and verified, an approach, namely Functionality-Based Approach (FBA), to understand software frameworks reading for defect detection. Then, the URMTV ESEG reasoned on the fact that frameworks have associated learning problems that affect their usefulness [10]; consequently, that Group focused on the problem of understanding the framework constructs and the necessity of providing inspectors with intelligible guidelines; subsequently, the URMTV ESEG planned and designed an experiment, and conducted that experiment at UKI, with academically novice students in the Department of Informatics (UKI-DI). Two of those reading techniques, namely Checklist-Based Reading (CBR), and Systematic order-Based Reading (SBR), were taken from scientific literature [26], while the third one, namely Functionality-Based Reading (FBR), was derived from Functionality-Based Approach. Results were that FBR seems much more effective and efficient than CBR, and that FBR is significantly more effective and efficient than SBR. However, because the experiment used CBR and SBR quite as they are with limited adaptation to frameworks, they were unable to draw strong conclusions from those results.

2.3. Study motivations and view

The motivation of the present study is to use framework understanding to guide framework reading for defect detection. Hence, we need to discuss reading for framework understanding first, then to elaborate on framework inspection, and on relations between these.

2.4. Framework architecture

In order to understand a framework, we need to know the basic entities that participate to definition of the framework architecture. In other word, compared to the Hierarchy-based technique [59], the approach that we propose for understanding frameworks, is to start from understanding constructs that express the top-most framework structuring concepts for the given framework (e.g. design patterns), and to proceed with understanding general-OO constructs: e.g. (i) basic constructs, such as classes and their relationships, the static hierarchical relationship (inheritance) included, and (ii) advanced constructs, such as meta-classes and reflection, if any, in the given framework.

We are hence faced now with assuming architectural structuring concepts for software frameworks. Based on our experience, white-box are mostly based on

Components, Interfaces, Design Patterns, and optionally Framelets as top-most entities for constructing and organizing frameworks. Such an architectural view gained confirmation [56] through several meetings that the URMTV-ESEG had with people of the applied research lab of a large company in the air-space domain, and one of their independent consultant, author of a C++ framework for that application domain [52]. During those meetings the group had the opportunity to gain experience with such a framework.

In more detail, those architectural entities are:

- **Framework Design Patterns** define domain-wide problems and present solutions in a generic abstracted form.
- **Framework Components** are fully implemented functionalities; they are used as they are by framework customers (application developers).
- **Framework Interfaces** include a collection of abstract operations that are called “hotspots” [52]. These are the only places where customization can be applied, and implementation of abstract operations (“hooking”) is the only mechanism that customer can use; in fact, based on the OO philosophy, customization is expected not be able to affect the structure and behavior of the basic framework.
- **Framework Framelets**, finally, are a kind of small frameworks that include Components, Interfaces and Design patterns; they can be used to structure building and manage documentation of complex frameworks.

2.5. Functionality-based approach for FW understanding

In our approach, in order to understand frameworks, framework’s architectural entities and their relationships should be mapped to framework’s Do-functionalities, Can-functionalities, Offer-functionalities, and their relationships; in other words, in order to understand frameworks we should read those entities, analyze their relationships, extract their functionalities, and classify these with respect to their importance, occurrence, and dependency.

We hence propose a new Functionality-based Approach to framework understanding, which is base on those concepts, and derive from such an approach a new reading technique, namely Functionality-based Reading technique, FBR.

2.5.1. From software framework artifacts to functionalities

Frameworks functionalities are solutions for a series of problems related to a specific domain. In order to understand the framework, functionalities need to be extracted

and classified from the framework entities. In such a process of extraction, entity operations are traced to framework functionality, and vice versa an operation (or set of operations) is mapped to a functionality of the application domain, or classified as local, i.e. it provides services to enact functionalities but is not a functionality in its own.

The following Section 2.5.3, reports on details concerning both the comprehension process, and the formal representation of the gained understanding gained. Let us discuss herein, in general, the process that supports such an understanding.

In a first stage (Design artifacts → Domain → Design Patterns), we start by deriving from the framework design the application domain and related design patterns. Following (Component diagrams → Class diagrams:: Class & Interface entities & relationships) we derive from the component documentation the framework classes, interfaces and related relationships. Finally, we read classes as broadly as compatible with the goal of extracting functionalities (Class → Operations:: Signatures, Comments, Pre-conditions & Post-conditions if any, Scope, Visibility etc. → Domain Functionalities).

2.5.2. Functionality types and relations

Once the set of functionalities has been detected in a framework, as previously briefly mentioned, those functionalities can be classified as in the following [52]:

Do-functionalities, DF: Functionalities that must be instantiated as they are by the user of the framework. They present the common necessary design of the framework, and constitute the implementation for basic usages of the framework; in other words, they present the minimum requirements that any framework should start with, when those frameworks are instantiated by the user. Do functionalities are independent from any of the other two types of functionalities; however, functionalities cannot be implemented and work well without Do functionalities.

Offer-functionalities, OF: Functionalities that are offered by the framework, they are those functionalities which may, or may not, be instantiated by the user with no influence on the overall functionality of the framework; this type of functionalities is one means to software

frameworks' quality; that is, increasing the amount of Offer-functionalities in a framework would support immediate reuse and, as a result, delivery to market.

Can-functionalities, CF: Functionalities that can be implemented by the user of the framework; usually, for each Can- functionality, standard implementations might be provided by default (in terms of links to Offer-functionalities); however, when the user decides to use a specialized rather than provided Can- functionality, the framework gives user the opportunity to specialize an abstract class or interface and through such a concrete class link to a specialized functionality rather than the default one; such functionality interfaces are what so called "hotspots"; this type of functionalities has the most coupling with other functionalities and presents the framework potentialities. The more hotspots are provided, the more the framework is adaptable to specific needs.

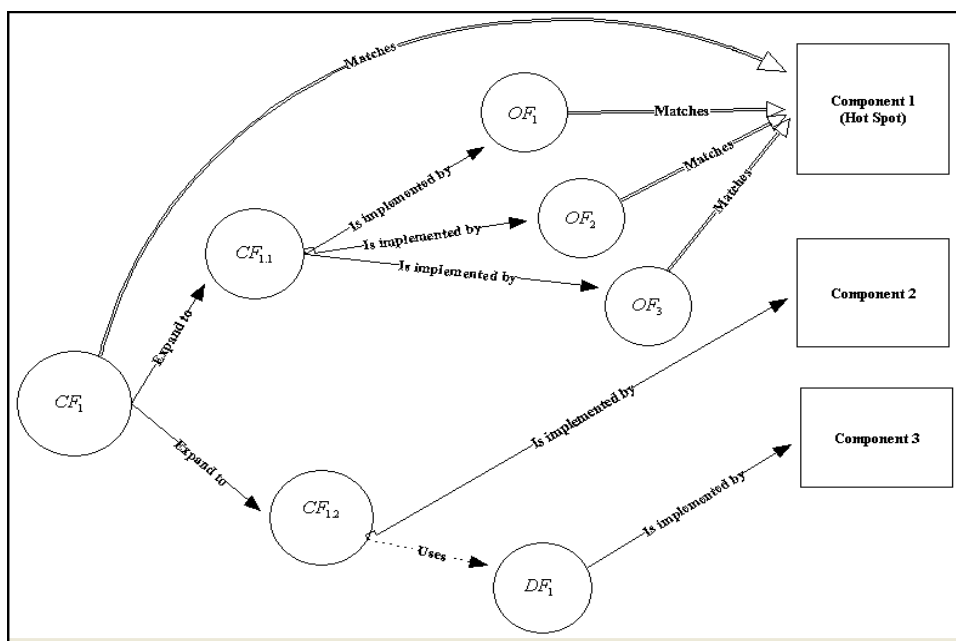


Figure 3: Functionality relationships

Can functionalities match hotspots; in general, functionalities can enter into relationships with each other; they can expand to, be implemented by, or use other functionalities:

- **Expansion Relationship** (Fx expands-to Fy): an expansion exists between a high-level functionality Fx and several lower level functionalities that collectively represent the high level functionality.

- **Implementation Relationship** (Fx is-implemented-by Fy): Fy describes an implementation of Fx. This relationship arises when a framework offers a component as a default plug-in for a hotspot.
- **Usage Relationship** (Fx uses Fy): this relationship arises when functionality Fx needs functionality Fy.

2.5.3. How to extract and characterize functionalities, and formalize understandings: Functionality rules

In the Functionality-based Approach to framework understanding, a framework is read with the objective of extracting and abstracting on the framework functionalities, tracing functionalities to the framework’s method-entries, and relating each other those abstractions. Functionalities are eventually represented in a formal structure (see Table 1) that we call with “Functionality Rule”. This structure supports many issues, including reading for defect detection, which comes later. Functionality Rules represent what a framework is expected to do in the user perspective, where it does this, and which are the cooperating functionalities, not how it does it.

Generating functionality rules is an additional documentation effort. That generation consists of representing a framework in a structured, higher-level human-interpretable form, which supports objectives of any type that require an abstract understanding of the framework as an input. Hence, that generation is not a part of, and takes place before applying, the reading technique for defect detection.

Table 1: Instance of “Do” functionality rule

Type & ID	DF[1]	
Location	CGCNet::SCListen / CGCNet::Unlisten	
Statement	The system must allow listening and stop listening to the TDM transmit timeslot, create an output file, and store the listened information in this file.	
Relationships	Uses	PostError in DF[5]
	Uses	LogEntry in DF[3]

In practice, when framework documents are read for understanding, the goal is to develop Functionality Rules; this means that functionality are both extracted and classified in one step; in this aim, framework artifacts are read in a top-down approach: domain problems and related design patterns are read first, and other software documents and code are read only if this is necessary for understanding the “what” and “where” of functionalities, and their relationships. We might reach understanding and stop reading at an early stage if the framework is well documented, that is, if functionalities are explicitly explained, abstractly registered in textual form (see “Statement” in Table 1), related to each other through expansion,

usage, and implementation relationships (see “Relationships”), and traced to specific methods (see “Location”). Otherwise framelets are accessed, if they exist in the framework. Eventually, the class code is recursively visited, and read in a concrete-method-first approach; that is, in order to extract functionalities, classes and their methods are visited starting from the top-most classes in the inheritance hierarchy, and proceeding to visit the derived ones until one reaches a concrete method. The implementation of that method is read with the goal of creating an abstraction of that method’s functionality. Methods that override such an implementation in sub-classes, if any, are also visited and read in order to refine, verify and update the abstraction established before. If the current method under examination sends a message, the called method in the destination class is visited only if this is necessary for a better understanding of the current functionality statement. When the classes and their methods in the framework have been visited, a certain number of functionality abstractions and their mapping to the enacting methods will be available. At this point, expansion, usage, and implementation relationships are established between those abstractions (Table 1). Successively, those abstractions are classified as Do, Can, or Offer Functionalities, respectively, as shown by the line “Type & ID” in Table 1. Finally, Functionality Rules are obtained, each consisting of four entities: textual statement of the abstract functionality, classification (i.e., DF, CF, or OF), location of the associated bottom-most enacting method, and relationships with other functionalities.

Chapter 3

An empirical study on Object Oriented Software Frameworks

Comparing Code Reading Techniques Applied to Object-oriented Software Frameworks with regard to Effectiveness

Abstract

This chapter presents an experimental research [2] for comparing the effectiveness and defect detection rate of code-reading techniques, once applied to C++ coded object-oriented frameworks. We present an experiment that compared three reading techniques for inspection of software frameworks. Two of those reading techniques, namely Checklist-based Reading, and Systematic Order-based Reading, were adopted from scientific literature, while the third one, namely Functionality-based Reading, was derived from the Functionality-based Approach. The results of the experiment are that (1) Functionality-based Reading is much more effective and efficient than Checklist Based Reading. (2) Functionality-based Reading is significantly more effective and efficient than Systematic Order-based Reading.

3.1. Experiment planning and operation

Figure 4 presents an overview of the stages in our experiment plan. In Stage S_1 , subjects partially received a large amount of knowledge and materials concerning the different experiment related topics through their course of OO programming, as will be explained later. In Stage S_2 , subjects were given a period of one week to understand basic materials and documentation of the training object through the Web site, and ask the experiment team about unclear issues. Stages S_3 and S_4 were executed on the experiment day; they represent two main events: S_3 represents the training session, where the subjects applied their inspection technique to a training

framework after having attended a lecture explaining the details of their technique. The S_4 stage is the experiment session.

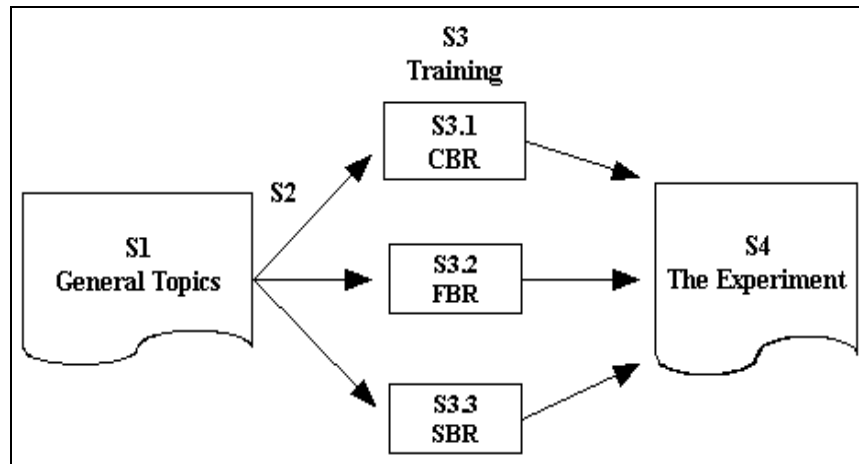


Figure 4: The experiment planning

After the experiment, as a further stage, techniques were described in detail to, and initial results were shown and discussed with, the experiment participants in an open session. Therefore, all students had the opportunity to learn, discuss, and acquire about all the experiment materials and topics. This stage was excluded from Figure 4, since it is external from the experiment period and aimed to cope with the ethical problem that might occur when different knowledge is given to students of the same academic course [22].

3.1.1. Hypothesis formulation

In the following, the null and alternative hypotheses in our study are presented for the experiment goal.

- $H_{0.1(\text{Effectiveness})}$: CBR, FBR, and SBR Effectiveness perform insignificantly different.
- $H_{1.1(\text{Effectiveness})}$: CBR, FBR, and SBR Effectiveness perform significantly different.

3.1.2. Variables selection

Concerning independent variables, the software reading technique was assumed as our factor, and the following ones as treatments: CBR, FBR, and SBR. In addition, techniques' Effectiveness is the dependent variables. The Effectiveness of a technique is the number of new or seeded true defects found by applying that technique divided by the number of known defects.

3.1.3. Subjects

Subjects are sophomore and junior students of the year 2003 of the Department of Computer Science at the University of Kaiserslautern in a course on OO Java programming and introduction to software. In total, 84 students participated at the very end of that course. Students had already attended more than two courses of computer science. We classified subjects as beginners, basic, moderately expert, and professional subjects, respectively. Participation was considered as a practical assignment; subjects would upgrade their score by 2.5 points (score ranges from 60).

3.1.4. Objects

Objects are real and professional software C++ OO Frameworks, developed and tested by well-known organizations [36]. The training object used for the experiment session is a controller for airspace communications [52]. The object used during the experiment session is the Telephony Device Controller [36]. In both frameworks, the subjects had to inspect 1000 lines of code.

3.1.5. Materials and infrastructures

As mentioned previously (see S_1 Figure 4), subjects received several general documents; those documents were described through the course lessons and were made available through the Web [64], such as: (i) An introduction to reading techniques as a part of the course in which subjects attending. (ii) Direct links on the Internet, where subjects can learn about frameworks and the training object. (iii) Frameworks terminologies and constructs for the experiment objects. General introduction on the experiment's reading techniques; without giving any further details about the techniques rules (guidelines). (iv) A presentation and documents that describe differences between Java and C++. On the training day (see S_3 in Figure 4), subjects were given a tutorial concerning the following topics: (v) UML Class diagrams, frameworks, and reading techniques: detailed description with examples [63].

3.2. Experiment design

One factor with three treatments is used for comparing means of the dependent variable for each treatment [39] [40]. Treatments had the same number of subject groups. Indeed, we assigned subjects to each technique by passing through the following steps:

- 1) Based on Stratified (or Blocked) Sampling Design [40], we classified subjects into two main categories with respect to, their knowledge of English language (C1), and their knowledge of C++ and UML (C2).
- 2) We arranged subjects into balanced groups of two people, each group included one member of C1; the remaining members of C2 were grouped randomly taking into account their level of experience (Convenience Sampling Design) [40], [67].
- 3) We randomly assigned techniques to groups so that each group had to use only one technique [40], [67].

3.2.1. Defects

In order to compare reading techniques we seeded defects into the experiment object. Seeding defects is expected to satisfy a realistic categorization and probability distribution [54]. Table 2 presents the actual numbers of defects that we seeded per category. In fact, our decision was to seed a very limited number of defects in each category. Moreover, we utilized five different categories only, and most of the seeded defects concerned the implementation of classes and their methods.

Table 2: Number of defects seeded per category

Category code	Defects	Category code	Defects
A: Initialization	3	D: Polymorphism	1
B: Computation	3	E: Interface	1
C: Control	3		

These facts would not introduce bias in favor of any of the reading techniques. That is because of following reasons: (i) Frameworks do not include parts that are typically in charge of the application programmer (e.g. screens), and our experiment framework does not include concurrency, events, exceptions and related handlers; consequently, we had to take in consideration few items of the IBM Orthogonal Defect Classification [35]. (ii) In order to better expose the capabilities of reading techniques [16], objects should be seeded with a relatively small number of defects (approximately equivalent to 0.1% of inspected code lines, 11 defects per KLOC, in our case). (ii) Defects in Table 2 are distributed according to known fault distributions ([54] pg. 337, [60]).

3.2.2. Experiment training and operation

The training session was planned on the same day as the experiment session, subjects performed the training session in a laboratory hosting 70 terminals; they had already been introduced to frameworks, to the training object, and to C++. The

subjects received separate lectures on their inspection technique; while subjects of one technique were taking their lecture, the other groups received their training in the laboratory. In the experiment session, the presence of the subjects was checked, and each group was allocated to one PC. Groups were under the continuous direct control of observers; discussion was permitted only within the same group. The experiment was conducted in one run; students were informed that time was open for up to four hours.

3.3. Experiment results and data analysis

We consider now the cross-match between reading techniques with groups and submitted data, including “true defects” (occurring when a subject submits a true defect, a false-positive being the submission as a defect of something that is actually right), categories, and timing.

3.3.1. Descriptive statistics

Data presented here are collected from groups who really attended the experiment, as shown by the second data-row of Table 3, while the third data-row presents the number of groups who submitted records they identified as defects, and the last row reports on groups who submitted true defects. Groups did submit 103 records identified as defects; 25 submissions were true defect detections, while 78 were false positives. Table 4 explains the total number of true defect detections and their percentages with respect to the total number of true defects detected by all groups. In addition, it shows the number of false positives and their percentages with respect to the total number of false positives reported by all groups. Defect Occurrence is the count of different true defect detected defects out of the total known ones as shown in the last row of Table 4.

Table 3: Groups assignment and attendances

Description	CBR	FBR	SBR
Groups based on registration form	13	13	13
Groups attended the experiment	10	10	8
Groups submitted data	8	10	7
Groups submitted true defects	3	9	5

Table 4: Defects and submissions for each technique

Description	All	CBR	FBR	SBR
Number of true defects	25	5	14	6
Percentage of true defects	100%	20%	56%	24%
Number of false positives	78	36	19	23
Percentage of false positives	100%	46%	24%	30%
Defect occurrences	11	4	9	5

In Figure 5, categories of true defect detections are shown for each technique; based on Table 2, we used letters as references for each involved defect category: A: Initialization, B: Computation, C: Control, D: Polymorphism, and E: Interface.

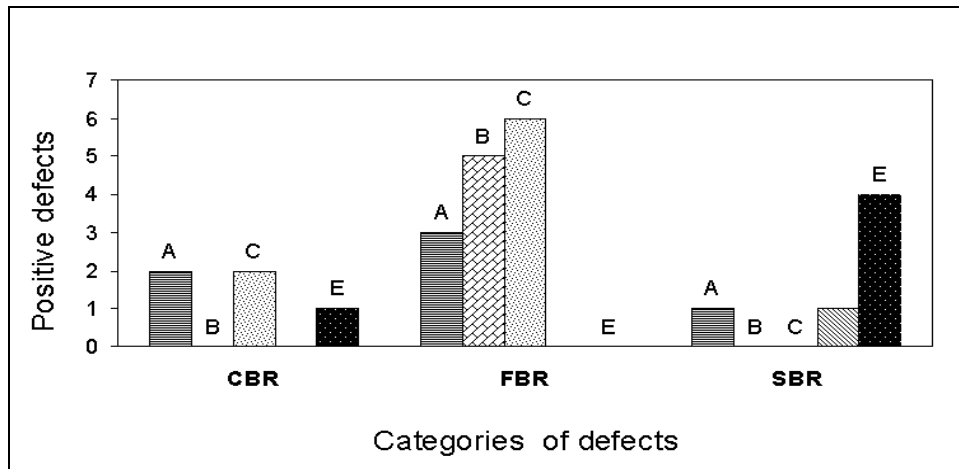


Figure 5: Defect categories vs. techniques for true defects

Table 5 presents number of groups who submitted timed true defects. Note that, compared with Table 3; three groups did not register detection time for one defect per each group. For each technique, Figure 6 shows a plot of true detections, as submitted in subsequent intervals of 30 minutes.

Table 5: Submissions for timed true defects

Description	All	CBR	FBR	SBR
Number of timed true defects	22	4	13	5
Percentage of timed true defects	100%	18%	59%	23%

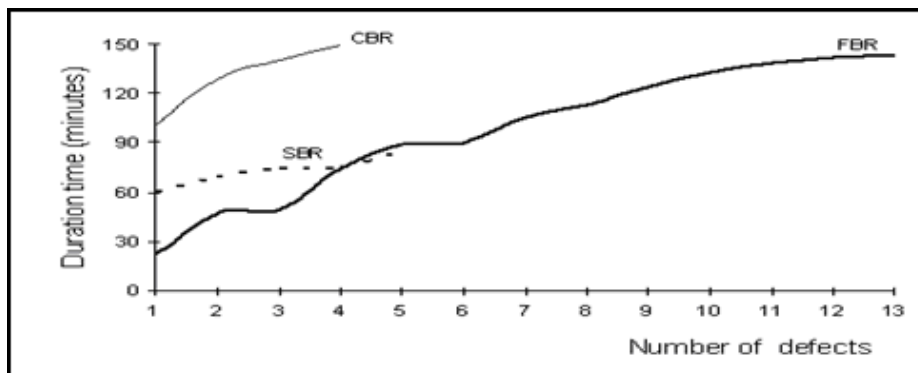


Figure 6: The number of true defects per time interval

3.3.2. Hypothesis testing

Figure 7 shows means of Effectiveness for CBR, FBR and SBR groups; those means are 0.50, 1.60, and 0.75 defects, and the most occurred numbers of true defects (the “ranges”) are 2, 3, and 2 true defects, respectively (note that, in the

followings, real numbers of those means are cast to their approximate numbers of type “natural” since defects cannot be fractioned).

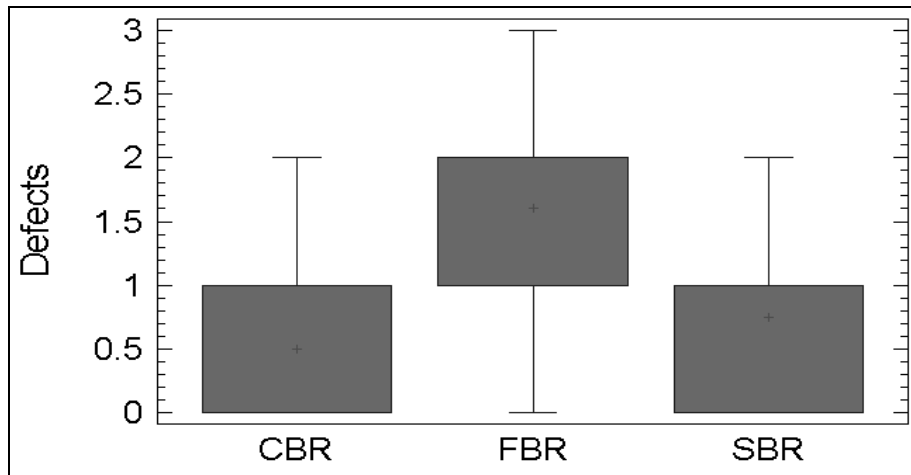


Figure 7: Box & Whiskers plot for effectiveness

For testing the Effectiveness of the experiment reading techniques, we applied an F-test; the ANOVA table in Table 6 shows the result of this test.

Table 6: Testing for the effectiveness of techniques

Source	S.S.	df	M.S.	F-Ratio	P-Value
Between groups	6.564	2	3.282	5.00	0.015
Within groups	16.400	25	0.656		
Total	22.964	27			

Based on the shown P-value, we can reject the null hypothesis, $H_{0.1}$ (Effectiveness), at 95% significance level; that is, we can conclude that there is a significant difference between the effectiveness of CBR, FBR, and SBR.

In order to identify which of the three techniques is significantly different, we apply for Multiple Range statistical testing [31]. In particular, Fisher's least significant difference (LSD) testing was used in our case. In Table 7, the results of the LSD test for Effectiveness show that (i) FBR technique is significantly different at the 95% confidence level when compared to CBR and SBR, and (ii) There is no significant difference at the 95% confidence level between the CBR and SBR.

Table 7: Fisher's LSD for effectiveness of techniques

Technique	LSD Value	$\mu_1 - \mu_2$	Result
CBR vs. FBR	0.746	-1.100	Significant
CBR vs. SBR	0.791	-0.250	Insignificant
FBR vs. SBR	0.791	0.850	Significant

3.4. Discussion

Qualitative and quantitative analysis for OO framework reading show that FBR, seems to be significantly more effective and productive than CBR and SBR.

The number of groups who submitted data, including true defects, as well as the defect occurrences, proved to be the highest value when applied for FBR compared to CBR and SBR. Based on the last two data-rows of Table 3, FBR groups submitted much more data than SBR groups, and SBR groups submitted a more than CBR groups. In particular, 80% of CBR groups, 87.5% of SBR groups, and 100% of FBR groups submitted issues they identified as defects. Among those groups, CBR, SBR, and FBR groups provided true defects in 37.5%, 71%, and 90% of their submissions, respectively. In general, CBR, SBR, and FBR groups who submitted true defects were 30%, 62.5%, and 90% (of the CBR, SBR and FBR participating groups), respectively.

The effectiveness of reading techniques, which represents averages on group performances, according to Figure 7, FBR technique detected true defects twice or more as often as SBR and CBR.

3.5. Threats to validity

3.5.1. Subjects

According to the number of participants, subjects were arranged into groups of two people. We avoided assigning a specific task to each one of the subjects in the same group; however, we could be faced with a threat to the experiment's internal validity because of subject pairs. Although subjects were introduced to differences between the Java language and C++ through a lecture including a 30-page document, subjects still represent a threat to internal experiment validity; the reason for this is that subjects were given their course on OO programming with Java, while the experiment objects are C++ frameworks. In addition, because of organizational difficulties during training, the lecture given to the SBR groups was too short to sufficiently explain their technique. As a result, we could be faced with a problem with the SBR groups. In particular, concluding their work earlier than the other groups could be an index of their limited understanding of SBR rules.

3.5.2. Training session

Training time was very strict with preplanned activities during the training session. As a result: (i) A decision was made to let subjects practice with the experiment's web

site and the assistance tools rather than having them apply the inspection techniques directly. (ii) While the research team was managing many tasks in parallel, unwelcome technical problems occurred in many different ways; the slow speed of some PCs and the performance of the experiment web site were the most critical technical problems during the training session.

3.5.3. Materials

Although we gave all groups the framework requirements, it is possible that CBR and SBR groups did not know how to invest them in a useful way for applying their techniques. This is because the CBR and SBR techniques are not tailored to frameworks, and it is probable that our adaptations of CBR and SBR were not sufficient to guarantee an efficient and effective inspection of the framework requirement documents. In contrast, FBR is tailored to frameworks, and forces one to read functionality rules carefully, which are abstract representations of the framework requirements. As a result, the passage from framework requirements to code reading was probably smoother for FBR groups than for CBR and SBR groups. Such a probable result should be specifically measured in future experiments, in order to know the impact of techniques on understanding. However, in the presented experiment, we avoided giving FBR groups any additional instruction about how to inspect framework requirements; above all, no defects were meant to be seeded in the framework documents. Finally, we did not consider defects related to framework requirements, in this experiment (4 defects detected by two FBR groups).

3.5.4. Seeded defects

Difficulties were found in seeding defects related to polymorphism and inheritance. Moreover, because our objects do not have input data, we did not consider omission or commission faults ([19], [54]). Finally, because subjects included sophomores who had not yet attended courses on relationships, we decided not to seed or consider faults concerning how framework entities relate to each other. As a result, we concluded that we probably failed to identify/define common framework defects and their distribution rate.

3.6. Conclusions

The chapter has presented and discussed the Functionality-based Approach and the Functionality-based Reading (FBR) technique for defect detection, which are specifically oriented to software frameworks. Comparing both to quite-generic

reading techniques, like Checklist-based Reading (CBR), and Object-oriented reading techniques, like Systematic Order-based Reading (SBR), the present study has shown that FBR is a promising reading technique for software frameworks, and seems to perform much better than CBR and SBR regarding both effectiveness and defect detection rate. SBR, in its turn, showed to have strength with respect to CBR.

In particular: Concerning group performance, it seems that FBR helps groups to detect defects much better than SBR, and the same holds with less strength for SBR versus CBR. Concerning the effectiveness of reading techniques, it seems that the FBR technique is able to detect true defects twice or more as often as the SBR and CBR techniques. Two further important results can be pointed out:

- Framework oriented reading techniques for defect detection are required to consider not only classes and their relationships, but also constructs like components, interfaces, design-patterns, and their relationships.
- Frameworks must be understood before inspecting them for defects, in order to extract “what” they are expected to provide, “where” the entry to each one of the functionalities they provide is located in the code, and “which” other functionalities cooperate with that functionality.

FBR is a reading technique that meets the above-mentioned points. Through defect detection, in order to know “How well” a framework implements its requirements, FBR inputs results from a previous phase of a framework understanding, guides inspectors to read framework constructs, and supports those inspectors in finding answers concerning “how” to inspect frameworks, “where” to start from, and “what” to inspect next.

However, it is too early to confirm results from this study. Further improvements and replications are strongly recommended and necessary before any definitive conclusions can be drawn. As a further study, a study on defining/identifying categories and distribution of defects in frameworks would be useful.

Chapter 4

Replication of An empirical study on Object oriented software frameworks

Evaluating Comparatively the Effectiveness of a New Inspection Technique for Object-Oriented Software Frameworks: a Replicated Experiment with Students in Different Levels of Experience

Abstract

This chapter is concerned with results from a replicated experiment for comparing the effectiveness of reading techniques, as refined and conducted with sophomores at the University of Rome “Tor Vergata”. The study, which was applied to C++ object-oriented frameworks, compared three reading techniques: Checklist-based Reading, Systematic Order-based Reading, and Functionality-based Reading. The latter was derived from the Functionality-based Approach previously defined by the Empirical Software Engineering Group of that University. Results were: (i) Systematic Order-based Reading and Functionality-based Reading are much more effective than Checklist-based Reading; (ii) There is no significant difference between Systematic Order-based Reading and Functionality-based Reading; in particular: (iii) Techniques performed significantly different when inspecting abstract and concrete frameworks' classes.

4.1. Experiment planning and operation

Compared to the basic experiment [2], variables, objects and materials are inconsiderably changed through the replication process; however, in order to enable further comparable studies, we made several major changes on the process flow: Phase A, is a couple of 90 minutes lectured sessions, which commonly considered software quality issues, verification techniques for defect detection, frameworks essentials, and the practical assignment of the students. Phase B, is concerned with subjects who had submitted their participation request; we used participation requests to collect information about students' skills in English language, C++, and

UML; subsequently, students were arranged into pairs using a specific sampling technique. Phase C, which represents the tutorial and training sessions. Phase D, which represents the experiment run and automated data collection. Note that, we provide the experiment with a survey in order to understand the effect of empirical studies on learning.











Phase	Activity	Subjects	Material
Phase A	Theoretical phase 		Problem and Goal frameworks related Knowledge
Phase B	Registration and Groups Assignment		
Phase C	Training phase  		Tutorial and Training
Phase D	Experiment phase  		The Experiment

Figure 8: The experiment planning and operation

4.1.1. Hypothesis formulation

In the following, the null and alternative hypotheses in our study are presented for the experiment goal.

- $H_{0.1}$ (*Effectiveness*): CBR, FBR, and SBR effectiveness perform insignificantly different.
- $H_{1.1}$ (*Effectiveness*): CBR, FBR, and SBR effectiveness perform significantly different.

4.1.2. Variables selection

Concerning independent variables, the software reading technique was assumed as our factor, and the following ones as treatments: CBR, FBR, and SBR. In addition, techniques' Effectiveness is the dependent variables. The Effectiveness of a technique is the number of true defects found by applying that technique divided by the number of known (seeded and new-found) defects.

The effectiveness relates both the number of true detections and the inspection technique. The effectiveness therefore can be illustrated from two, specific rather simpler, different perspectives:

- Effectiveness per Defect Category, EPDC; is the number of true detections versus defect categories.
- Effectiveness per Inspected Classes, EPIC; is the number of true defects detections versus the type (concrete or abstract) of class to be inspected in the framework.

4.1.3. Subjects

Subjects are graduate students (Bachelor engineers in Informatics) of the 2003/04 academic year in the DISP at the URm2, in their final year for the Magisterial degree in Informatics, in the very end of a course in OO Software Analysis and Design; they can be considered as moderately experts in OO programming and design; they also have basic knowledge of UML. In total, 60 students participated. We classified subjects as beginners, basic, moderately expert, and professional subjects, respectively. Participation was considered as a practical assignment; subjects would upgrade their score by up to 2 points (the score ranges from 18 to 30). Students will be receiving information concerning the experiment results.

4.1.4. Objects

The experiment was conducted with one object for all subjects and all treatments; the training and experiment objects were extracted from one framework in the domain of telecommunications, so called Telephony Device Controller [36]. The framework was developed in C++, powered with OO design; according to authors of the framework [36], the latter was tested and no syntactic defects are expected. The experiment object contains 1115 LOC and 408 comment lines in 4 classes. As shown in Table 8, the experiment object consists of four classes to be inspected, three abstracts and one concrete class; four defects were seeded in two abstract classes (class A and class B), while three defects were seeded in the concrete class (class C). Based on our experience [19], we decided to seed a small of defects. In fact, in order to better expose the capabilities of reading techniques, objects should be seeded with approximately 0.1% of inspected code lines, 11 defects per KLOC.

Table 8: Summary statistics of classes to be inspected

Classes	Type	LOC	# Methods	Coupling
Class A	Abstract	209	15	5
Class B	Abstract	117	11	3
Class C	Concrete	718	13	7
Class D	Abstract	71	14	2

4.1.5. Materials and infrastructures

Subjects received several general documents (see "phase A" in Figure 8) and were made available through the Web [60], such as: (i) An introduction to software quality and reading techniques. (ii) Frameworks terminologies and constructs for the experiment objects. (iii) An introduction to the experiment reading techniques with no

further details about the experiment techniques rules to apply at experiment time (guidelines were not given for training but distributed to subjects in the beginning of their experiment run). Table 9 shows provided documentations and guidelines for each reading technique. Unlike the basic experiment, we have provided all groups with FBA documents as general guidelines for understanding of the framework.

Table 9: Documentations provided to groups

Documentations	CBR	FBR	SBR
FBA Documents and UML artifacts	†	†	†
Technique Description	†	†	†
The object domain guidelines	†	†	†
Statistics concerning the object classes			†

4.1.6. Experiment design

One factor with three treatments is used for comparing means of the dependent variable [40] [67]. In order to contribute the experiment, subjects were requested to fill an electronic form that includes a few questions concerned with their capabilities in English language, C++, and UML; we decided to use such data to counterbalance the experiment inputs. Based on Stratified (or Blocked) Sampling Design [40], we classified subjects into two main categories with respect to, their knowledge of English language (C1), and their knowledge of C++ and UML (C2). We then arranged subjects into balanced groups of two people such that each group included one member of (C1). The remaining members of (C2) were grouped randomly taking into account their level of experience (Convenience Sampling). Finally, we randomly assigned techniques to groups so that each group had to use only one technique (C3) [40], [67].

4.1.7. Defects

Table 10 presents the number of seeded defects per category. We seed four different categories only because frameworks do not include parts that are typically in charge of the application programmer (e.g. screens), and our experiment framework does not include concurrency, events, exceptions and related handlers; consequently, we had to take in consideration few items of the IBM Orthogonal Defect Classification [35]. Defects in Table 10 are distributed according to known fault distributions [54] pg. 337.

Table 10: Number of defects seeded per category

Category code	Defects	Category code	Defects
A: Initialization	3	C: Control	4
B: Computation	3	D: Polymorphism	1

4.1.8. Experiment training and operation

The experiment was conducted in three consecutive sessions. Groups were controlled 10 minutes before starting their session. Four observers at least were available in a laboratory that contains 20 local-networked terminals. Each inspection session lasted 180 minutes; groups were unable to cooperate with each other during the experiment.

4.2. Experiment results and data analysis

Let us now consider some descriptive statistics; Data presented here are collected from groups who really attended the experiment. As we are interested in this study on the effectiveness of experiment reading techniques, we will acutely consider changes on the number of true defects (occurring when a subject submits a real defect) and false positives (the submission as a defect of something that is actually right in the framework).

4.2.1. Descriptive statistics and hypothesis tests

Groups attended the experiment are presented in Table 11, two groups withdrawn from CBR sampling for the sake of the experiment validity.

Table 11: Defects and submissions for each technique

Description	CBR	FBR	SBR
Groups' attendance	13	13	13
Groups submitted data	11	9	11
Groups after data reduction	9	9	11
Groups submitted true defects	6	8	9

Table 12: Defects and submissions for each technique

Description	CBR	FBR	SBR
Number of true defects	10	32	38
Number of false true defects	62	40	51
True defects submitted (in average)	1	4	4
False positives submitted (in average)	8	4	5
True defects to false positives	10%	44%	43%

Moreover, number of true defects and false positives are shown in the first pair of data rows in Table 11; second pair of data rows describes the average of all groups' submission means. Last data row is the ratio between the number of true defects and the number of false positives.

4.2.2. Analyses of EPIC

The total average of EPIC for all groups is shown in Figure 9. Note that class D has no seeded defects.

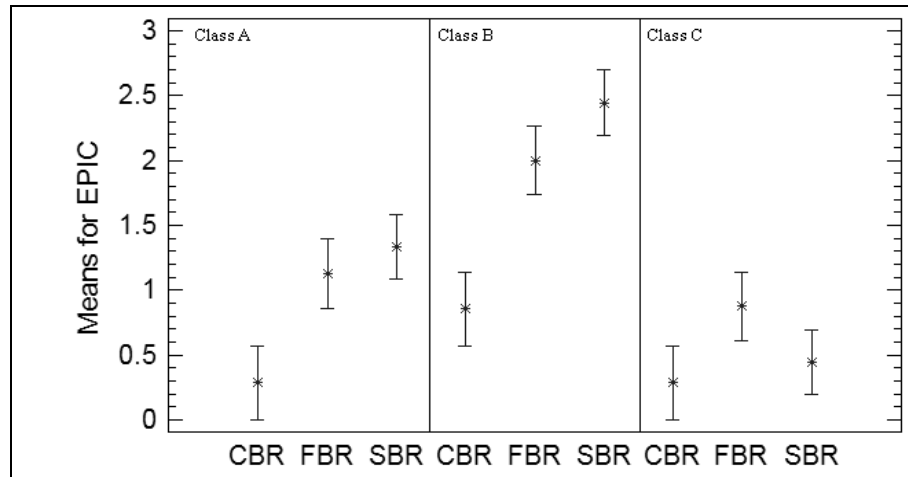


Figure 9: Means for EPIC in the inspected classes

As shown in Table 13, EPIC is tested; the aim is to check for significant changes in defect detection when classes are changed. According to resulted P-value, CBR, FBR, and SBR differ significantly within/between classes to be inspected.

Table 13: Testing for the EPIC of techniques

Source	S.S.	df	M.S.	F-Ratio	P-Value
Between groups	37.20	8	4.65	8.16	0.000
Within groups	35.91	63	0.57		
Total	73.11	71			

Table 14: Fisher's LSD for EPIC of techniques

Category	Technique (Class)	LSD Value	$\mu_1 - \mu_2$
I	CBR(A) vs. FBR(A)	-0.839	0.781
	CBR(A) vs. SBR(A)	-1.048	0.760
	CBR(B) vs. FBR(B)	-1.143	0.781
	CBR(B) vs. SBR(B)	-1.587	0.760
II	FBR(A) vs. FBR(B)	-0.875	0.754
	FBR(B) vs. FBR(C)	1.125	0.754
	SBR(A) vs. SBR(B)	-1.111	0.711
	SBR(A) vs. SBR(C)	0.889	0.711
	SBR(B) vs. SBR(C)	2.000	0.711

4.2.3. Analysis of overall effectiveness

Table 14 presents only results that have statistical significant difference. It is a class-based statistical comparison between/within CBR, FBR, and SBR for true defects per

class. Results are grouped in two categories: techniques for similar classes are placed in category (I); classes for similar techniques are placed in category (II).

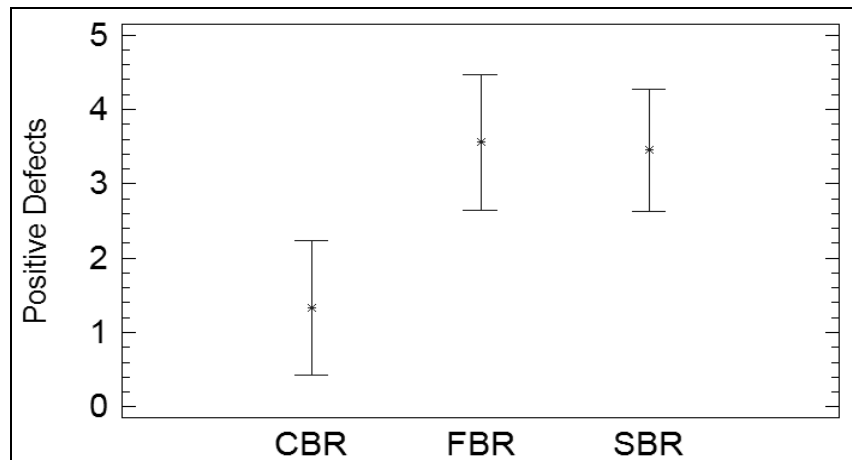


Figure 10: Overall means for true defect detections

Figure 11 shows the overall means for true defect detections. For testing the overall effectiveness, we applied an F-test; the ANOVA table in Table 15 shows the result of this test. Based on the shown P-value, we can reject the null hypothesis, H_0 , at 95% significance level; that is, we can conclude that there is a significant difference between the techniques' effectiveness.

Table 15: Testing for the effectiveness of techniques

Source	S.S.	df	M.S	F-Ratio	P-Value
Between groups	35.47	2	17.74	5.02	0.01
Within groups	91.84	26	3.53		
Total	127.31	28			

In order to identify which of the three techniques is significantly different, we apply for Multiple Range statistical testing [31]. In particular, Fisher's least significant difference (LSD) testing was used in our case.

Table 16: Fisher's LSD results for the effectiveness of techniques

Technique (Category)	LSD Value	$\mu_1 - \mu_2$	Results
CBR vs. FBR	-1.357	0.740	Significant
CBR vs. SBR	-0.857	0.720	Significant
FBR vs. SBR	-0.810	0.720	Insignificant

In Table 16, the LSD test for Effectiveness shows that (i) SBR and FBR technique are significantly different at the 95% confidence level when compared to CBR, and: (ii) There is no significant difference at the 95% confidence level between FBR and SBR.

4.3. Discussion

4.3.1. Techniques' effectiveness

According to resulted p-value in Table 15, FBR and SBR are significantly more effective than CBR, while no sufficient significant evidence to confirm changes between FBR and SBR.

For what concerns the provided documentations (see Table 9), CBR subjects were uncertain about the goodness of their performance; according to the questioner, FBR and SBR groups were strongly close to the documents that actually helped them to understand the framework functionalities. FBR groups had the FBA as an inspection requirement, they were following each functionality starting from the bottom-most concrete classes; however, in software frameworks, FBA is affirmative as much as comprehension details are sufficient. In addition to the FBA, we provided SBR groups with an extra document that presents abstract specifications for the experiment object. This document is required for applying SBR; however, according to the original guidelines of SBR [26], subjects are required to build those abstract specifications, while we already had prepared them.

For what concerns the techniques effectiveness, testing the effectiveness per Inspected Classes, (EPIC), showed that techniques' performances in abstract classes have significant differences (Category I in Table 14); (i) CBR groups insignificantly differ when comparing their performance for abstract and concrete classes (Category II in Table 14), that is because CBR technique performs randomly with respect to classes to be inspected. (ii) FBR groups are much more effective when inspecting concrete classes. Moreover, because of the fact that FBR employs the implemented classes (Bottom-most classes) to understand and eventually inspect the abstract ones, the goodness in performance of FBR seems to be proportional to the goodness of functionality rules which are derived from the FBA. In our object, class C had to implement nine methods that are virtual methods of class B (six) and class A (three), respectively; this implies that FBR deals perfectly with classes that have most coupling and most implementation. (iii) SBR groups: SBR technique inspects the framework classes based on an ordinal and systematic procedure that is related to class coupling and de-localization state. According to the experiment object, SBR groups started with class A since it has the lowest coupling; class B came next, and class C was the last; they used the FBA to understand methods during their inspection. According to Figure 9, SBR groups have spent most of their effort in passing between abstract classes. In such a mechanism, it is

probable that SBR groups were not able to understand the framework functionalities as much as FBR groups since their technique is not forcing them to follow the framework functions up to their implementations.

4.3.2. Threats to validity

This experiment was planned, designed, and conducted as a replication of the basic one [2]. We highly intended, therefore, to increase the credibility of the current experiment by avoiding most of the threats that we explored in the basic experiment, most of them were related to the waterfall type of, and the lack of assistance feedback in, the experiment process.

What actually still under consideration is the inspection in pairs; that is, according to the number of participants, subjects were arranged into groups of two people. We avoided assigning a specific task to each one of the subjects in the same group; however, we could be faced with a threat to the experiment's internal validity because of subject pairs.

In addition, Difficulties were still exists in seeding defects related to polymorphism and inheritance. Moreover, because our objects do not have input data, we did not consider omission or commission faults ([19], [54]). Finally, because subjects included sophomores who had not yet attended courses on relationships, we decided not to seed or consider faults concerning how framework entities relate to each other. As a result, we concluded that we probably failed to identify/define common framework defects and their distribution rate.

4.4. Conclusions

A controlled experiment was conducted as the first replication of the basic one [2]; the goal was to evaluate changes in the effectiveness of three reading techniques of software frameworks for defect detection.

Basic results support the idea that Checklist-Based Reading, Functionality-Based Reading, and Systematic-order Based Reading perform differently once applied for the experiment OO software framework.

Moreover, for what concerns Checklist-Based Reading, we are able to confirm the weak performance of this technique once applied for OO software frameworks. One reason for such a conclusion is that reading techniques that do not force the inspector to understand the framework domain and functionalities could be insufficient with respect to OO frameworks' inspecting requirements.

Chapter 5

More Empirical Studies on Object Oriented Software Reading for Defects

Effectiveness of Code Reading and Functional Testing with Event-Driven Object-Oriented Software

Abstract

This chapter is concerned with experimental comparisons of code reading and functional testing (including fault identification) of concurrent event-driven Java software. Our initial idea was that functional-testing is more effective than code reading with respect to concurrent event-driven OO software. A controlled experiment was initially conducted with sophomore students (inexperienced subjects). Subsequently, it was replicated with some changes with junior and senior students (moderately experienced subjects). We also conducted a further replication with Master students, which is not considered in this Chapter. The experiment goal was studied from different perspectives, including effect of techniques on the different types of faults. Results can be overviewed as the following: 1) Concerning the initial, *basic experiment*: with inexperienced subjects and a strict interval of inspecting time of two hours, there was no statistically significant difference between the techniques under consideration; subjects performance indicator was 62% for code reading and 75% for functional testing. 2) Concerning the (first) *replication*: with moderately expert subjects, again a strict interval of inspecting time of two hours, and more than twice number of seeded faults, there was no statistically significant difference between the techniques; subjects performance indicator was 100% for code reading and 92% for functional testing; subjects performance indicator shows that more experienced subjects were asking for more inspecting time; however, functional testing performed much better than in the basic experiment. Computation faults were the most detectable for code reading while control faults were the most detectable for functional testing. Moreover, moderately expert subjects were more effective than inexperienced ones in detecting interface and event types of faults. Furthermore

moderately expert functional testers detected many preexistent (non-seeded) faults, while both inexperienced subjects, and moderately experienced code readers could not detect non-seeded faults.

5.1. Introduction

This Chapter presents an empirical study where effectiveness of two logically different fault detection and identification techniques were compared through a controlled experiment in order to address the uncertainty of how to, and when use one technique rather than the other one, or overlap such techniques, in order to test software effectively.

Similarly to other studies known from the literature, this chapter analyzes both static and dynamic software-testing techniques. In the followings, two techniques are considered and referred as Code Reading for defect identification, CR, and Functional Testing & fault Identification FTI, where CR is a static testing technique and FTI includes a dynamic black box testing [4]. In particular, we present and discuss results from a couple of experiments, the basic experiment (BE) and the first of its replications (R1), that we conducted at The University of Rome “Tor Vergata”, Department of Informatics, Systems and Production, in the Experimental Software Engineering Group (URM2-DISP-ESEG). While both effectiveness and efficiency of software testing techniques were investigated at URM2 [16], this Chapter reasons on effectiveness only.

Our experiments show some changes with similar previous research, and we can classify some of these changes as major changes, other ones as minor changes. In fact: 1) Our study emphasizes on specific experiment objects. In particular, our experiment objects are concurrent OO software games; these are realistic software, not “toy” games; they are driven by keyboard and mouse events rather than data. 2) Our subjects are students in Informatics engineering, from sophomores up to seniors. 3) Our experiments are technology-based and run on Internet; documents are in electronic format only, paper supports are not needed, forms are electronically presented, submitted, and database stored; our checklist items are assertions, i.e. Boolean statements to answer by True (Checked) or False (Unchecked) rather than questions to answer by Yes (Checked) or Not (Unchecked).

Based on the specific nature of the software application that we used as objects, the following question arises:

Q: Can results from previous empirical studies, which concerned static and dynamic testing of data-driven programs, be extended to other kinds of software, e.g. concurrent OO event-driven software?

In order to start reasoning about such a question, let us begin by noting that both CT and FTI include code searching for fault identification, but FTI is required to enact one preliminary step: program execution for failure detection. In our reasoning, this does not imply that CR performs better than FTI, at least for some types of faults, with OO event-driven software. In other words, confirmation of previous studies, namely the Basili and Selby empirical study on effectiveness of testing strategies [4], is not quite an obvious conjecture at this point. In fact, checklists only drive CR testers to read code for fault identification, while two items, checklists and detected failures, drive searches of FTI testers; this could help FTI testers to perform better than CR testers, when time necessary to detect a program failure is very small (and, by the way, the program execution is fast enough.)

Based on such initial reasoning, our experiment idea can be derived:

When software user-interfaces are friend and user-interactions is limited to simple actions, e.g. clicking mouse or pressing keys, functional testing performs better than code reading in the average.

Our supposition in this area is hence that FTI would perform better than CR in the average, relying on the following causes:

- Due to the specific type of applications that we are considering, because of their user interfaces and their type of interaction with users, the visibility of failures should be very high and the time strictly necessary to detect such a failure should be very small; consequently, the FTI failure detection phase should have minor impact on the overall testing process.
- The subsequent FTI code-search should perform better than CR with respect to the common task of identifying software defects; in fact, while both techniques use the same static objects and point to get fully identifying faults, the FTI search is driven by known failures and specified assertions, while CR is driven by assertions only. This should make a major difference.

We can now come to define the primary goal of our study. In GQM (Goal-Question-Metrics) terms [8] [60] [5], this can be expressed as in the follows: To analyze checklist-driven Code reading and Functional testing for the purpose of comparing effectiveness and detection rate with respect to testing concurrent event-driven OO software –actually Java games- for defect detection and identification, in the context of advanced software-technology facilities and students of different level

of experience, from the point of view of a research group, which also aims to transfer resulting experiment process and guidelines to industrial settings in the mid-term.

As already mentioned in the statement of the primary goal, we aim to move experiment to industry. The strategic goal of our study is thus to learn by a family of experiments how to achieve stability in experiment processes, documents and supporting tools, in the aim of reducing the experiment risk and improving the expected yield on investment. This is considered to be the prerequisite to introduce the techniques into practice.

As the remaining of the Chapter deeply shows, based on the experiment goal, the following items characterize our experiment definition:

Research hypotheses are concerned with whether performances of two testing techniques differ significantly when applied by inexperienced and moderately expert students of Informatics engineering, respectively. In particular, the null and alternative hypotheses tested by our experiments can be formally stated as in the following:

- H0 (Effectiveness): CR and FTI perform insignificantly different in detecting faults.
- H1 (Effectiveness): CR and FTI perform significantly different in detecting faults.

Testing technique is the experiment factor. Experience of subjects is variation, an “inter-experiment” factor.

CR and FTI are the experiment treatments. Inexperienced or moderately expert subjects are variations, inter-experiment treatments.

Informatics Engineering sophomores, in the position of inexperienced subjects, and additionally, juniors and seniors, in the position of moderately expert practitioners, are two samples of experiment subjects.

Concurrent event-driven Java games are the experiment objects.

Effectiveness is the experiment dependent variable, where:

Effectiveness (Mean)—the number of preexistent or seeded defects found, rated to the total number of known defects.

The impact of treatments on the dependent variable is observed by the following four perspectives: Inspecting time, Number of seeded faults, Subjects’ performance, Types of seeded faults.

In summary, in our experiments, we can identify testing technique as the experiment factor, CR and FTI as the treatments, and effectiveness as the dependent variable that is correlated to the factor. In order to investigate the impact of the treatments on the dependent variable, each our experiment controls remaining independent variables at fixed values, including the experience of subjects. However,

in order to evaluate the impact of the latter on the dependent variable, we let it change from the first to the second experiment by fixing it at the level “Basic experienced subjects” in the first case, and at “Moderately experienced subjects”, in the second case, hence establishing an “inter-experiment” factor. Moreover, in both cases we use different perspectives to observe the impact of treatments on the dependent variable, namely the number and type of seeded faults, the available inspection time, and the subjects’ performance.

5.2. Starting scenario

A milestone of empirical research on the effectiveness and efficiency of software testing strategies is the Basili and Selby 1987 IEEE TSE paper[4], and is rooted in the related experimental research that Basili initiated in the 70’s and continued to develop in the ‘80s. This research suggests that experienced subjects are most efficient when using code reading, and that experiments involving students confirm the results with minor significance. In particular, for what concerns this Chapter, the major results from the Basili and Selby’s study are: 1) With professional programmers, CR detects more software faults than FTI; 2) With advanced students, CR and FTI are not different. 3) Both the number of faults observed, and total effort in detection depend on the type of software tested. 4) CR detects more interface faults than FTI. 5) FTI detects more control faults than CR.

The Basili and Selby study has been replicated a certain number of times with minor or major changes along the last fifteen years, and some supporting tools have also been developed and published. The web sites of the International Software Engineering Research Network, ISERN, and the Empirical Software Engineering Research Network, ESERNET, can reach many of these works. On 1995, Kamsties and Lott conducted quasi-replications of the Basili and Selby study at the University of Kaiserslautern [42]. Results suggested that inexperienced subjects can apply code reading as effectively as an execution-based validation technique, but they are most efficient when using functional testing. Based on such experience, the Basili and Selby study was also replicated in the Strathclyde EFoCS group, where the role of replication in experimental software was investigated in the first part of 90’s [13]; in addition; Dunsmore, Roper, and Wood [26] conducted a series of three empirical studies on a rigorous approach to OO code inspection on 2001. More recently, Juristo, Moreno and Vegas conducted a replication at the Polytechnic of Madrid [49]. Finally, our replications at the URM2 DISP ESEG were conducted with major changes, and further our replications are still ongoing [15].

Regarding projects that relate to this study:

- The experiment is part of the “Experimental Informatics” project, partially supported by our Ministero dell’Istruzione, dell’Università e della Ricerca Scientifica (MIUR), Grant 020906003 URM2-DISP.
- The experiment aims to produce results for the EC-funded Empirical Software Engineering Research Network (ESERNET).
- The experimental activities have been continually supported by the URM2 project “Cooperation among the ESE groups of URM2-DISP, UMDCS, UKI-DI,” Grants 010209013-2001, 010209015-2002 URM2-DISP.

5.2.1. Method

To enable interested readers to conduct replications, the present Section first describes in some detail how we conducted the study, the roles and people involved, and the process enacted.

Concerning the roles involved. The roles basically involved with this study are:

- An expert in empirical/experimental software engineering (ESE).
- An expert in statistical methods and tools.
- Other roles involved are:
- A system and database administrator and security manager.
- A professional, expert in software technology and deeply aware of the state of the art in our software industry.

Concerning the people involved. Subsequently to role definition, the following individuals, who also authored the present Chapter, filled those roles (in the same order as listed above):

- A professor of ESE at URM2-DISP.
- A Research-Doctoral¹ student with a Bachelor in Statistical Science from a foreign university.

¹ Because the organization of academic degrees changes world wide, in order to prevent misunderstanding, let us explicitly recall the present Italy engineering degrees. Diploma is a five years degree, which ends with a graded, public dissertation of the Laurea Thesis in presence of ten or more local faculties. Bachelor is a three years degree, which ends with a graded, public presentation of a project to five or more local faculties. Specialized Bachelor is a post-bachelor two years degree, which ends similarly to Diploma. Research Doctor is a post-diploma and post-specialized-bachelor three years degree, which needs defense and ends with a public formal dissertation of the Doctoral Thesis in presence of faculties coming from different universities.

- A Diplomat in Computer Science, actually serving as employee at the URM2 Computing and Documentation Center.
- An Informatics engineering PhD, actually with Rational Software.

A Bachelor Engineer student in Informatics, preparing her final project, performed in the additional role of Software engineer Assistant.

Concerning the process enacted. Let us consider now the process that we enacted in order to develop the basic experiment and its replication. The BE work was started on the very beginning of November 2001 with experiment definition. Based on the time constraints placed by the Bachelor student team-member, in that she intended to get her grade in less than six months, we came to choose for the experiment process an iterative-incremental model, which extends on the process model shown by Wohlin et al. [67]. As deeply explained in the remaining, during the definition of BE, the decision was made to use a balanced multi-test within object study, that is to have the same number of subjects for each testing technique (treatment) and to use one experiment object for multiple subjects. The further decision was made to conduct both the basic experiment and replication in lab through Internet. The development was hence planned of on-line guidelines able to direct subjects to carry out their work, and eventually access and fill out defect forms. The Bachelor student team-member was given the tasks of developing the Web site and the experiment material, with the guide and responsibility of the professor team-member. We selected the experiment and training objects. In addition, based on the technical literature, we categorized defects. Eventually, the number of defects to seed was determined; actual defects were retained or created from the scratch and hence seeded into objects. When the plan for defect seeding was completed, submission forms were designed. Web pages were created in accordance with these forms and placed on the experiment Web site. A repository to store defect submissions was created. The Web site was enhanced to provide complete guidelines for subjects, from initial authentication up to defect submission-through-Web forms. Experiment documents and objects were made available to the research-team via Internet and access-protected by password and through a firewall. The training material was also developed and placed on the same Internet-connected server. The Web server performance was tested by a standard package. A couple of weeks before the start of the experiment, the training material was published on the development server, for access through the Internet, and information was given to registered participants. Finally, both the basic experiment and replication were conducted in lab. On-line guidelines directed subjects to carry out their work, which

lasted two consecutive hours. When a subject detected a defect, guidelines directed him or her to access and fill out the proper form.

Concerning the language used. Our community is still missing a unified ESE language. However, in order to model our activities and products we used the language implicitly defined by Wohlin et al. [67] (see also Chapter 2 of this book.)

5.2.2. Design

As noted earlier, the goal of this study is to carry out comparisons of two testing techniques, focusing on their effectiveness, which is the experiment dependent variable. The research hypotheses of the experiment are hence concerned with whether those two testing techniques differ significantly when applied to concurrent OO event-driven Java software for what concerns their effectiveness. Accordingly, we identified testing technique as the experiment factor, CR and FTI as the treatments, and effectiveness as the dependent variable that is correlated to the factor. In addition, there are independent variables that we are expected to control at fixed values; furthermore, we should reason about the type of design to adopt. We will look now at each of these identified entities in more detail.

Concerning the factor. In order to obtain comparable results, the testing techniques are specified to produce the same type of results, whatever their nature might be. In our case, techniques are required to come to fully identified faults, including the identification of faulted line of code, and fault category. Let us note that, beside the testing technique, our study, which was structured in a basic experiment and a set of replications (with variations!), includes a further “inter-experiment” factor, which is the experience of participating subjects.

Concerning treatments. CR is a step-by-step procedure that guides individual inspectors in uncovering defects in software-coded artifacts. Such a technique is required to provide a systematic and well-defined way of inspecting software code, allowing for feedback and improvement. CR works on the application code as it is; it is hence a static technique. FTI can be seen as organized in two phases: functional testing and fault identification. The former is a dynamic black-box testing technique; it looks for program failures, and requires the execution of the software application. The latter consists in a code search for faults that caused a detected failure. Both our treatments are checklist driven; specifically, they are driven by assertions, i.e. a list of predicates in natural language on the program’s specified behavior and non-functional requirements. In particular, for both techniques, testers are given program informal requirements (intended functional behaviors and non-functional

requirements) and some construction documents, including source code. Checklists are also given or tester is requested for their development before starting with test.

As already mentioned, checklists models drive CR testers to read software-code and to compare the asserted requirements with coded ones. A defect is located and identified when an inaccuracy is detected in one of the inspected software document with respect to the asserted requirements. Checklists also drive FTI testers to compare the specified requirements with program actual behave. For functional testing, testers additionally are given the executable version of source code (or the source code interpreter; in our case, the Java machine). Functional testing only chance is to detect application failures. Subsequently, FTI testers are required an additional step: in order to fully identify faults that caused an observed failure, they access software construction documents, eventually the application code.

Concerning further independent variables. These are the intra-experiment independent variables, which are controlled at fixed values:

- The type of knowledge and experience of subjects.
- The level of knowledge and experience within the same samples of subjects both with concurrent OO event-driven concepts, and Java programming language.
- Team size.
- Duration of the experiment runs.
- Quality and complexity of guidelines and other material supporting the experiment.
- Supporting technologies, including document supports, form-submission technology, tools and instruments used, screen type and size, and network performance.
- Interaction among subjects and between subjects and observers.
- Other environmental characteristics.

Since all these variables are independent for each student in the experiment, no further strategy needs to be applied to allocate the subjects into groups.

Table 17 summarizes components of the basic experiment design and replication design, showing the fixed values for the applied techniques, where N, P, M, and I stand for Not, Practitioner, Inexperienced, and Moderately expert subject, respectively.

- “Non-practitioner” was the fixed value that we evaluated for the type of knowledge and experience of all our subjects.

- “Inexperienced subject” and “Moderately expert subject” were the fixed values that we evaluated for the level of knowledge and experience of BE subjects (Sophomores) and R1 subjects (advanced students, actually Juniors and Seniors), respectively.

Table 17: Components of the BE and R1

Description	BE		R1	
	CR	FTI	CR	FTI
Type and level of knowledge and experience	NPI	NPI	NPM	NPM
Number of subjects	23	22	13	12
Team size	1	1	1	1
Experiment-run duration (in minutes)	120	120	120	120
Number of experiment objects	1	1	1	1
Number of defects seeded	38	38	95	95
Laboratory	L1	L1	L2	L2
Technology	ICT	ICT	ICT	ICT
No. of runs	1	1	1	1

Based on Table 17:

- One person was the fixed value that we chose for the experiment-team size.
- We excluded to split an experiment in multiple runs.
- 120 minutes was the fixed valued that we chose for duration of each experiment run.
- Information and communication technology (ICT) was used to support all the experiments. Guidelines with minor treatment-dependencies were also established for all the experiments. So the quality, complexity, and performance of the supporting materials were fixed at common fixed values.

Concerning the type of design. Since our common aim is to compare two testing techniques, the experimental design for our basic experiment and its replication is the “One factor with two treatments” for comparing the mean of the dependent variable for each treatment [67]. Moreover, the study is an offline, controlled experiment—specifically, to be conducted under controlled conditions at a laboratory. Furthermore, there is an inter-experiment variable, which is the experience of subjects. Indeed, our subjects are assigned to each technique using completely randomized design such that each subject uses only one treatment on one object. Treatments are designed to have the same number of subjects so that design is made balanced. To make sure that subjects will be assigned randomly,

their names are designed to be listed with serial number attached; the development of a small application is required for generating random numbers without redraw for each technique. The subsequent step is to assign an experiment code for each subject and technique besides the experiments themselves; as already mentioned, we coded Code reading technique as “CR”, Functional testing technique as “FTI”, the Basic experiment as “BE”, and the (first) replication as “R1”. Both in the basic experiment and replication, each subject is treated by his/her experiment code along the experiment process.

5.2.3. Subjects/Participants

This Chapter reports on a stage of our experimental work on testing techniques, when materials and processes still required for testing, improvement, refinement, and tuning. Too much risk was thus still involved with any tentative of transferring the study to industry. Consequently, we addressed students for the position of experiment subjects. In particular, skilled students of Computer and Information Engineering, including sophomores, juniors and seniors, were involved. Participation was voluntary; we did not pay subjects for participating nor wanted to grade them through the experiment. As a return for participating, subjects were given the opportunity to be trained free on advanced programming topics in class sessions, through Internet and, for R1, through participant-reserved lab training sessions.

Based on convenience sampling, our subjects were defined as the 2002-year population of the URM2-DISP students in three different courses: OOCF, for the basic experiment; Object-oriented analysis and design (OOAD), and Experimental Software Engineering (ESE), for R1.

Eventually, 45 out of 150 sophomores participated on voluntary basis to BE. These subjects had already attended two courses of Computer Science at least, and were in their very end of OOCF. 30 out of 50 advanced students participated on voluntary basis to R1. These were attending OOAD as students in the Bachelor degree (juniors) or in the Diploma or Specialized-bachelor degrees (seniors). Additionally, they had already attended courses of Software Engineering, Database, Operating systems, and so on. 10 of those participating seniors were attending ESE; many of these were also attending OOAD.

All subjects were trained. During this stage, the application that subjects applied is a game called “Pinball”; in detail, we released six versions of this application for BE, five versions for R1, each with five types of seeded faults, one version for each game-difficulty level.

For the BE experiment training, we introduced subjects both to CR and FTI techniques, and the "Pinball" game by class lectures. Subsequently, we invited subjects to visit the web site of the experiment, in order to continue the training on personal basis.

For R1 experiment training, we introduced subjects to CR and FTI techniques first in class and then in lab. Subsequently, we invited subjects to continue their visits of the web site of the experiment on personal basis, in order to get further training. For the lab-training sessions, the environment was completely in the control of three observers. The lab had eighteen PC machines available for the experiment training. Subjects were arranged randomly using statistical simple random sample method. Based on the number of subjects and available PCs, subjects were arranged as singles and couples, and the "Pinball" was briefly recalled and shown them in execution. They were hence introduced once more time, for about thirty minutes, to the experiments requirements and both the proposed techniques. Successively, they were randomly assigned to a technique, and started the training session. After one hour of training, they were requested to quit their current technique and switch to the other one. Because FTI and CR commonly share some practical stages, we assumed that the second part of the lab training would not require more than thirty minutes.

5.2.4. Apparatus/Materials

Here we will look in more detail at some points made earlier about the objects and seeded faults, infrastructure, and guidelines related to the experiment. Additional information can be found on the URM2-DISP-ESEG Web site, <http://eseg.uniroma2.it>.

Concerning objects. As already mentioned, in our experiment definition, games were designated for use as experiment and training objects. Moreover, the decision was also made to select the experiment objects among the games that we had worked with students of previous courses of OOC. Furthermore, we eventually decided to prefer Solitaire, a game known world wide, as the experiment object, and Pinball, a ball game constructor, as the training object. These objects are based on two Java games that T. Budd had previously authored and published [14], which we had already maintained - by removing some faults, extending some their logic requirements, and improving quality of user interfaces and time performance of user interaction – and reused, in order to produce new games (e.g. Flipper.)

Again during experiment definition, the additional decision was made to further adapt those games according to experiment requirements. Some changes were defined in order to introduce the concept of level of game difficulty, and hence create different versions of objects, one for each level of game difficulty. In fact, Solitaire requires players (and FTI testers in such a role) to spend a certain amount of time for reasoning on what next and what if, with respect to their actual, random selected, case. Vice versa, CR subjects are expected to enact a general reasoning by abstracting on real cases or, in the worst case, working on scenarios. This could penalize FTI subjects with respect to CR ones: in fact, working on abstractions is simpler than working on reality. The organization of the experiment object in multiple levels of difficulty allowed us, on one side, to dividing and managing the game complexity from the player point of view, on the other side, allowed for controlling and limiting the possible interactions between different failures produced by different faults. In order to have different levels of difficulty for the programs, five versions were designated to play the role of the experiment object. Following further decisions were also made: 1) Regarding experiment-game presentation, we decided to introduce a window, where players find buttons to start from level 1, advance in the game difficulty level or eventually quit. (In order to control learning effect, the return to already visited levels was not allowed.) 2) Regarding concurrency, we decided to introduce some new threads for computing and controlling score achieved by a player during the execution of each level. 3) Regarding the object documentation, we decided to improve the in-line code-documentation, and to reverse the code in order to produce class diagrams.

Concerning fault seeding. In order to compare testing techniques, we need to establish a fix point to use as reference. This can be obtained by seeding faults into objects.

Fault seeding is expected to satisfy a realistic categorization of faults. Instances of such categorizations are based on fault-nature, e.g. {Omission, Inconsistency, Ambiguity, Incorrectness, Syntax}, fault-severity, e.g. {Low, Moderate, High}, or fault-type, (see Table 18, for instance). In an experiment, for each selected categorization, faults should be seeded according to a realistic probability of occurrence. S. H. Pfleeger reported on fault classifications and distributions [54]. Table 18 shows our classification of faults for concurrent OO event driven software: it extends on fault definitions of the IBM Orthogonal Defect Classification [35]. Unfortunately, data that concerns distribution of faults in Java event-driven software are still not available, in the best of our knowledge. We hence made the decision to change faults density

through the basic experiments and its replications, mainly due to the changes in the number of type D and type H seeded defects.

Table 18: OO, Event-Driven Defect Classification. Extends IBM Orthogonal Defect Classification [35])

Type	Code	Explanation
Assignment Initialization	A	Incorrect implementation or invocation of a constructor, missing or incorrect initialization of data, wrong assignment.
Algorithm: computation Method	B	Faults that lead to incorrect calculations in a given situation or incorrect method.
Algorithm: control	C	Faults that lead the program to follow an incorrect control flow path in a given situation.
Inheritance Reflexivity	D	Faults that relate to static ancestors of a class. Faults that affect the run-time ancestor of a class-object.
Interface Message Polymorphism	E	Faults that affect quality of user interfaces, class definition of behaviors, interface with other subsystems, components or objects or classes or utilities via message exchange or parameters lists, or via calls, macros, or control blocks. Faults related with template instantiations and parameters, or binding a message to the proper port, channel, entry or method of the (variable) destination object.
Function Class Object	F	Faults that affect capability, correctness and completeness of end-user interfaces, product interfaces, interface with hardware architecture, or global data structure, class implementation of specified behaviors.
Event	G	Faults that affect event definition, detection, identification or service activation and completion.
Exception	H	Faults that affect exception raise, detection, identification, or handling activation or completion.
Concurrency	I	Faults that effect synchronization, mutual exclusion, control of non-determinism, and fairness.
Checking	L	Faults in program logic that fails to validate inputted data and value before they are used.
Relationship	M	Faults that relate to problems among objects, procedures, and data structures.

We seeded our experiment objects with previously detected and already removed faults, and with further faults. Because our objects do not have input data, we did not considered omission or commission faults of type L in Table 18.

Table 19: Faults seeded per fault-category

Fault Code & Type	BE (# %)		R1 (# %)	
A: Initialization	7	18,4	37	38,9
B: Computation	6	15,8	11	11,6
C: Control	5	13,2	8	8,4
D: Polymorphism	5	13,2	0	0,0
E: Interface	6	15,8	19	20,0
F: Function	6	15,8	0	0,0
G: Event	1	2,6	20	21,1
H: Exception	2	5,3	0	0,0
I: Checking	NA	NA	NA	NA
TOTAL	38	100,0	95	100,0

Moreover, we decided not to seed or consider faults of type M, because our basic experiment was based on sophomores, who are not expected to be aware of design and implementation of relationships between software entities (e.g. classes and objects.). Table 19 presents the different types of seeded faults for both experiments.

Concerning infrastructure. The experiment was constructed to meet the following low-level design requirements as well:

- To be conducted at an open-workshop laboratory able to guest up to 100 subject at the same time (this is the maximum number of sophomore participants that we estimated during the design of the experiment).
- To use true and up to date IC technology rather than paper and pencil.
- To be easy to replicate in any place, at any time, hence to run on the Internet, so having objects, data collection forms, experiment data, procedures, and all other documents organized to take place on a Web.
- To run in half of a day.
- To be based on the assignment of similar workstations to all participants, specifically workstations equipped by equal real and virtual machines, keyboards, and screens.

Based on those requirements, the decision was made to run the experiments extra-campus at an Internet Café. We actually conducted the basic experiment at this place as our lab L1 (see Table 17). There were PCs of the same type, ready and connected together to an internal network and via Internet to the experiment Web site.

Because in the meantime the Internet Café managers changed their mind and highly raised the half-day rental for a loft-laboratory, we had to reconfigure the experiment design, in order to let the replication be conducted at our academic lab. There were 25 PCs of the same type available for the experiment replication, connected together and, via a local network, to the lab server. We adapted those PCs to meet the experiment requirements for apparatuses. We also duplicated our experiment on the Internet-not-connected lab server.

Concerning instruments measurements. Fixed measurement instruments used were as following:

- Detected faults submission electronic forms.
- Detected faults description submission text file.
- Database for storing data submitted by subjects electronically.
- Description electronic forms.

According to fault definitions in Table 18 and the interested types of faults, eight electronic forms were provided with short descriptions. Each provided form was made of two principal parts:

- The first part leads testers to enter their ID Group Number and the relative actual time.
- The second part leads testers to answer some questions, in order to verify whether the detected fault effectively lies on one of the types of faults indicated in this form. Once the tester finds the question that matches the detected fault, the tester enters the class name and line number in which s/he detected that fault, and the detection time (in the current version of the supports, submission time is automatically inserted by the system).

Concerning guidelines. In order to execute uniformly and efficiently the experiment process, it is necessary to define steps for the techniques that testers have to carry out. There are steps that are common to both techniques and steps that are technique specific.

Common steps: subjects have to read carefully documents concerning the software application requirements and hence generate their own assertions, according with those requirements (as already mentioned, the replication package already provides assertions, hence R1 subjects were only asked to read and eventually extend assertion lists). Subjects find assertions in a text file, and write their new assertions in their local copy of this file; (in particular, this file is specified to contain the following descriptions for every assertion: Progressive ID number, Assertion state, and Assertion classification according to the fault type that this assertion addresses.) Subsequently, the subjects view the experiment object class diagrams in order to make an idea of the responsibility given to each class. Finally, the subjects enact the assigned testing technique. The current assertion (state) drives the tester. For each new level of difficulty of the game that a BE tester enters, s/he has to memorize the admission time (during the conduction of BE, we eventually

authorized subjects to use paper and pencil for writing down memo for admission time). When a tester detects a fault, s/he classifies this fault according to previously indicated eight categories of faults, and then opens a new form for specific fault type, fills out admission time, current time, and fault description, and eventually submits the form. Thereby a new test-session is defined and so forth, for next fault identification and submission (based on the experience that we gained while conducting BE, our system has been evolved to record admission and submission time automatically). Consecutively, subjects repeat the previous common steps for each further assertion and for each level in the application. Right before the expiration of the time duration assigned to the experiment run, observers require subjects to send the text files that they elaborated to an ad-hoc experiment's e-mail address.

FTI-specific steps: If there are conflicts between the current assertion and the application behavior, then the tester has to describe the wrong behavior in a local text file. Subsequently, FTI tester passes to locate the fault that caused the failure. In order to locate a fault in the application code, the tester is requested to search the software-code of the application classes that caused a failure in her or his conjecture. To have the application class-diagram available for display, can hence help the tester. As already mentioned in the Common steps above, once the tester has identified a faulted line of code, s/he has to fill out the form for this type of defect and submit the related form.

CR-specific steps: CR tester tries to identify the class, which has the responsibility of the current assertion. S/he reads carefully the class code in order to detect faults. If a fault is detected, then tester has to describe the expected wrong behavior from the user point of view in a local text file. S/he hence proceeds to fill out the form for this type defect and to submit the related form, as already mentioned in the Common steps above.

5.2.5. Procedure

In the BE experiment, subjects were requested to be present at a private Internet Café in Rome downtown in the morning of 2002 February 14, at about 8:30 A.M., where we had reserved an open-workshop for exclusive use. There were 50 PCs of the same type, ready and connected together to an internal network and via Internet to the experiment Web site. A total of 45 individuals participated to the basic experiment as subjects. No. 15 subjects were assigned for individual application of CR, and No. 15 subjects were assigned for individual application of FTI. Moreover,

No. 15 subjects were assigned for individual application of FTO, i.e. Functional Testing Only; the task of such subjects was to detect and describe failures only, without entering the code. The experiment was started at 10:30 A.M.; subjects were informed in advance that a very hard deadline of 120 minutes was in place for all the experiment procedures, plus 5 minutes for transferring text files with assertions and error descriptions by e-mail. At 12:30 subjects were requested to exit immediately the experiment web site and start transferring files by e-mail. The Internet café management logged off subjects at 12:40. Because No. 2 CR individuals, and No. 7 FTI individuals were late in their file transfer, we lost their data. In the following, only CR and FTI data will be considered; in fact, we are still working on FTO data.

The R1 experiment is a replication with variations of BE. Subject were on time, in the morning of 2002 March 27, at 9.30, taking their places in the computer lab at the URM2 DISP. There were 25 PCs of the same type, ready and connected together to an internal network and via a local net to the experiment Web mirror on the lab server. No. 13 subjects were assigned for individual application of CR, and No. 12 subjects were assigned for individual application of FTI. At 10.00 A.M. they were informed that duration time was two hours, and hence started their task. During the experiment conduction, because one of the FTI subjects did not seem enough trained, observers permitted him to continue his work but also decided to discard his data.

For both experiments, the observers provided all subjects with login name, password and link to electronic documents and instructions to apply the experiment. Common documents included: Solitaire game requirements, Solitaire class diagram, Code of classes for each level of game difficulties, Guidelines for reading and extending assertions, Assertions (for BE, an empty text file). In order to support both the comprehension of the software application, and fault reports, the lines of code of the Solitaire game had been numbered and annotated. FTI-specific documents included both FTI forms, defined according to the classification of faults, and Java Machine executable code of Solitaire (it is present for each level of game difficulty). Functional testers were allowed to access code only after they had detected a program failure. CR-specific documents included CR forms defined according to the classification of faults. CR testers were not allowed to use the Java Machine.

Interested readers can find all the documents and details related to the experiment operation by visiting the web site http://eseg.uniroma2.it/webESEG/esp_cr.htm.

5.3. Results

In this Section, descriptive statistics and hypothesis testing are presented. First of all, let us present directions that we followed in developing the analysis of quantitative and qualitative data results.

In order to investigate the impact of independent variables on the dependent variables, we evaluated projections of effectiveness with respect to some dimensions (or perspectives); these projections would explain the complete dependent variable when combining them in one. The reason for applying such an approach is to have a full govern of the analysis, and to treat the several independent variables involved with those perspectives in a fair independent way. The perspectives we used, and their explanations are discussed as follows:

- **Inspecting time:** in the present study, based on the definition of effectiveness, we are interested in observing the number of faults detected per each technique, and hence, the time available for inspecting is a part of the effectiveness' power of techniques under investigation; that is, one technique could force subjects to invest more inspecting time than another technique in order to detect an equal number of faults in the used objects.
- **Number of seeded faults:** since we are discussing an experiment with one replication, it is significant to describe the effectiveness of the techniques with respect to the number of seeded faults, especially when the number of seeded faults is significantly different in the two experiments. In particular, we are focusing on the number of detected faults out of the seeded ones.
- **Subjects' performance:** again, based on the definition of effectiveness, subjects' performance is one of the indicators of the effectiveness' power of techniques under investigation. In fact, we used two levels of subjects experience; therefore, subjects performance perspective – as we will see later on- effects on the detection process and, as a result, the effectiveness of the techniques.
- **Types of seeded faults:** faults detected by subjects were categorized according to their types; in fact, the type of seeded faults perspective is a sub-case of the number of seeded faults perspective, and hence, we are focusing on the number of detected faults out of the seeded ones for each type.

Thus, the four perspectives are explaining the same object from different points of view, and they also lead, together, to the main goal of this study. We are working for testing how strong is the correlation between those perspectives and, if so, we are going to apply some multivariate analysis [12] [32] [43].

Consider that, in order to give the ability of making comparable decisions between the results from the two experiments, data are normalized to 1 with respect to the number of participants, the number of seeded faults and, in particular, the different types of faults (Figure 11 excluded, which shows number of detected faults.) Consequently, data are presented in relative values (percentages) rather than absolute values. In particular, percentage data are presented in the range from 0.00 to 1.00 (rather than 0 and 100, respectively, because of their normalization to 1 rather than 100.)

Consider also that, in our definition, total number of *actual faults found* is different from the number of *true defect detections*; in fact, many subjects can detect the same actual fault, hence many *true defects* can be mapped to the same actual fault. Because it is reasonable to presume that the number of actual faults found tends to reach the number of (seeded or preexistent) faults when search time tends to reach infinite, actual faults found is a proper indicator for demonstrating the effectiveness of testing techniques on the inspection process.

Concerning descriptive statistics. Different types and numbers of faults were seeded in the inspected programs; actually, these faults were seeded with respect to only their different types; however, subjects interacted with these seeded faults and their types with respect to the following four different perspectives.

Inspecting time: Figure 11 presents inspection process for both techniques with respect to inspecting time. Consider that inspection process for BE and R1 were proceeded within a limited time of two hours.

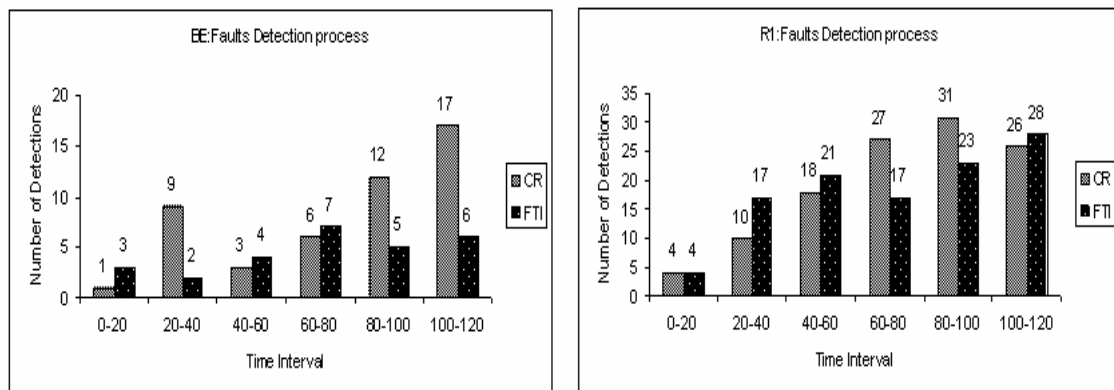


Figure 11: Inspection Process for CR (left) and FTI (right), BE and R1

Number of seeded faults: For the basic experiment, faults were seeded in an inconsistent way, and inspected faults were really small with respect to the seeded ones. At the beginning, we referred the lack of sufficient data both to subject's inexperience, and the small number of seeded faults, though thirty-eight different types of faults were seeded. Therefore, in the first replication, we seeded ninety-five different types of faults, and selected a higher level of subjects' experience.

In Figure 12, we present “how many actual faults out of the total number of the *seeded faults were detected*” in BE and R1, respectively.

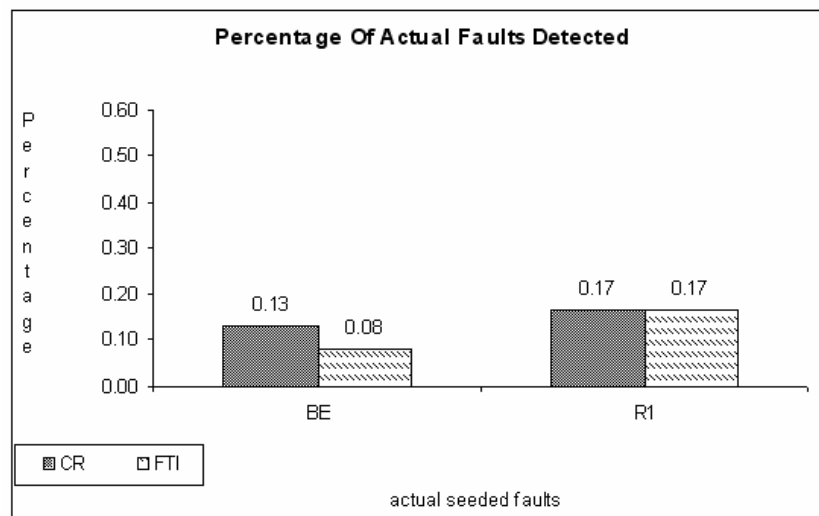


Figure 12: Percentages of actual faults detected

Figure 13 demonstrates “how many true faults were detected with respect to each testing technique”. According to our calculations, averages of detections of faults by all subjects were involved, where many subjects can detect the same fault and this count for each subject. In other words, percentage of true fault detection numbers is the overall average of the subjects' fault detection means.

Subject performance: Performance of BE and R1 subjects is not equal regarding to many reasons such as their experience and knowledge, though they were chosen from the same population, i.e. URM2-DISP university students.

Figure 14 demonstrates “how many people detected true faults with respect to the technique applied”.

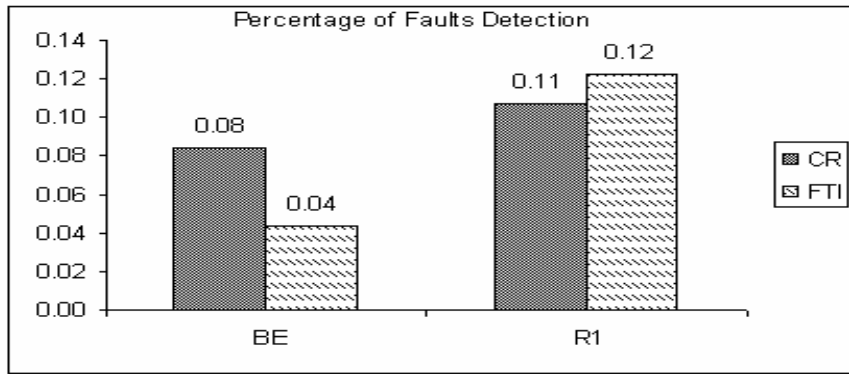


Figure 13: Average of the total number of detections

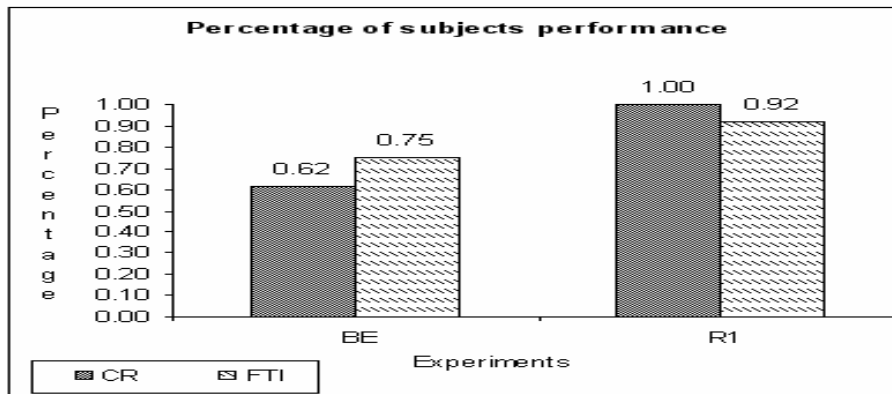


Figure 14: Percentage of subjects' performance

Types of seeded faults: This is the most important perspective for demonstrating techniques effectiveness. We would like to know "how many types of faults were detected with respect to techniques under investigation".

Let us begin with actual faults. Figure 15 shows actual faults detected with respect to fault types.

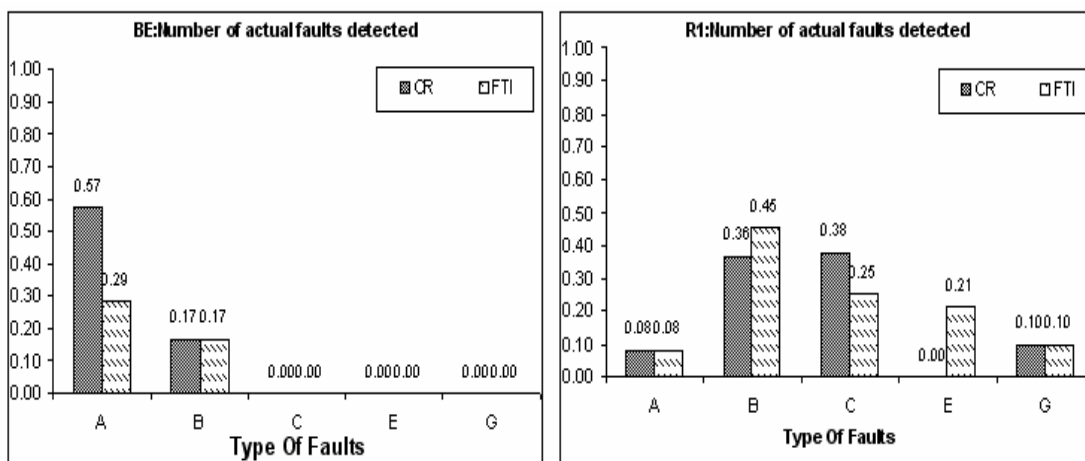


Figure 15: Actual detected faults classified by their types

Now we would like to know "which type of actual faults is the most detectable among BE and R1". Table 20 shows the detection of actual faults in a descending

order. Let us consider now true defects. Figure 16 shows true fault detections with respect to fault types.

Table 20: Actual faults detected classified by their types

LEVEL	CR		FTI	
	BE	R1	BE	R1
First	A	C	A	B
Second	B	B	B	C
Third		G		E
Forth				G
Fifth				A

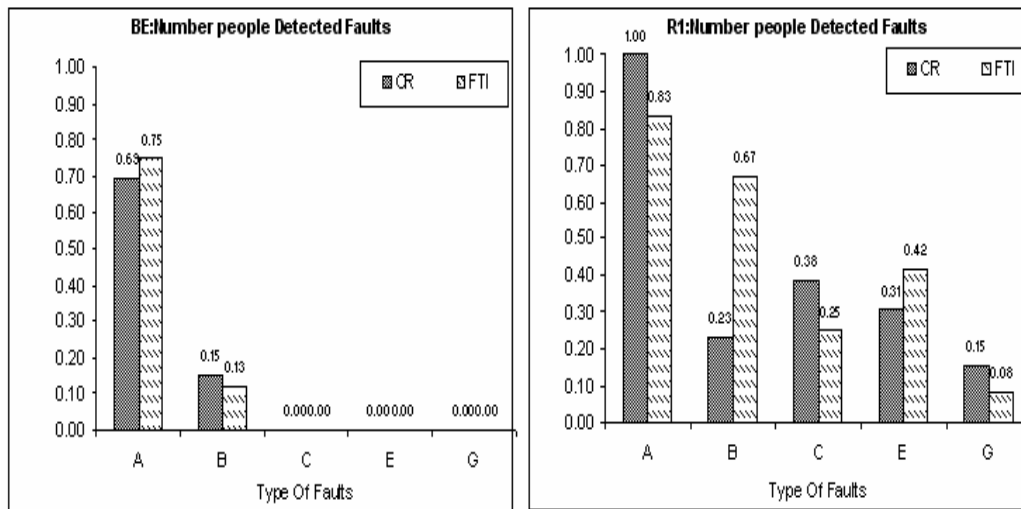


Figure 16: True defect detections classified by fault types

Now we would like to know “how many subjects detected each type of faults”. Table 21 shows true faults detected in a descending order.

Table 21: Types of faults classified by the number of subject’s detections

LEVEL	CR		FTI	
	BE	R1	BE	R1
First	A	A	A	A
Second	B	B	B	C
Third		E		E
Forth		C		B
Fifth		G		G

Concerning hypothesis testing. Let us now present the testing of our hypothesis regarding the effectiveness of CR against FTI. Again, we remind that our main hypothesis’ formal state is as the following:

H_0 (Effectiveness): CR and FTI perform insignificantly different in detecting faults.

H_1 (Effectiveness): CR and FTI perform significantly different in detecting faults.

Moreover, the research team, according to some initial suppositions and primary analysis, assumes that FTI technique would be more effective in detecting faults than CR technique, with respect to the several perspectives that we previously defined and described in this section. Notice that, for brevity reasons, we do not consider tests of normality more than presenting the final results from the two experiments. However, we can say briefly that, by applying the test of normality for the basic experiment and its first replication, results are stating that the two reading techniques can be fitted under the normal curve, which means that we can apply a parametric test for testing our hypothesis for each experiment. According with such result, we applied for t-test, which are represented by the p-values shown in Table 22 BE experiment. The test statistic for the BE experiment shows no significant difference between the two techniques concerning their capabilities of detecting and identifying faults.

Table 22: t-test for effectiveness of CR and FTI for BE experiment

Results	p-value	Test	Alfa
Accept H_0 (BE Effectiveness)	0.08	t- test	0.05

Box-and-whisker plot can be useful for handling and presenting our data values. Box-and-whisker plots allow people to explore data and to draw informal conclusions when two or more variables are present. It shows only certain statistics rather than all the data. These statistics consists of the median, the quartiles, and the smallest and greatest values in the distribution. Immediate visuals of a box-and-whisker plot are the center, the spread, and the overall range of distribution.

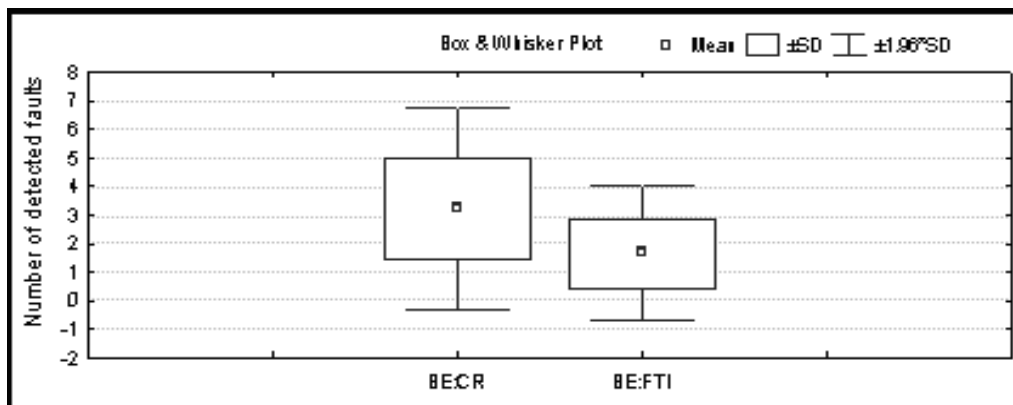


Figure 17: Box &whiskers plot for effectiveness for BE

For what concerns R1, Table 23 presents the t-test for effectiveness and, again, no significant differences are provided between the techniques with respect to their capabilities of detecting and identifying faults.

Table 23: t-test for effectiveness of CR and FTI for R1 experiment

Results	p-value	Test	Alfa
Accept H_0 (R1 Effectiveness)	0.40	t- test	0.05

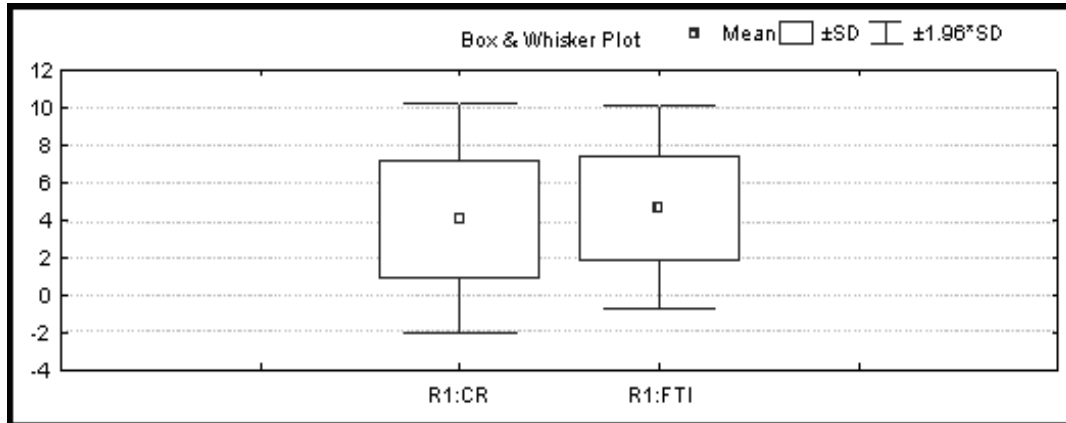


Figure 18: Box &whiskers plot for effectiveness for R1

5.4. Discussion

In this Section, we first evaluate the experiment results, and discuss their implications, especially with respect to the original hypotheses. Then we synthesize and enumerate ten results, whose further investigation is still necessary, with respect to four variables, which should be considered precisely. Finally, we consider threats to validity. Before we begin analyzing our data, let us point out that when we passed from the basic experiment to its replication, we made some changes, so that the latter faced with noticeable differences, which affected the followings:

1. Checklist-construction process
2. Experiment laboratories
3. Subjects experiences
4. Number of seeded faults
5. Applied training methods.

Reader should hence consider that, while the experiments that we are presenting are part of a family of experiments, their results should be considered separately, except for cases in which the experiments reciprocally confirm results.

We consider now the extraction from our descriptive statistics for evaluating the effectiveness of our testing techniques. We will discuss our issue from the perspectives previously assumed, and will give explanation, based on Figure 11 up to 19. Table 24 recalls those perspectives and presents their initial values.

Table 24: Different perspectives for evaluating the effectiveness of CR (left) and FTI (right)

Perspective	Explanation	BE value(s)	R1 value(s)
1 st - Inspecting time	Total detection time	120 minutes	120 minutes
2 nd - Subjects' experience	Number of subjects detecting faults	0.62 vs. 0.75	1.00 vs. 0.92
3 rd - Actual faults detection	Number of detected actual faults	0.13 vs. 0.08	0.17 vs. 0.17
4 th - Total of true detections	Frequencies of faults detection	0.08 vs. 0.04	0.11 vs. 0.12
5 th - Types of faults	Types of seeded faults found	CR	FTI
		1 st type: C 2 nd type: B	1 st types: A, B, G 2 nd type: B

5.4.1. Inspecting time

For each experiment, the inspection process for both techniques was extended to the last few minutes before the inspection's termination. This point is demonstrated by Figure 11, which implies the following:

1. The assigned testing time was not enough.

Inspecting time vs. number of seeded faults: Preexistent and seeded detected faults for R1 are around twice higher than BE, in the average. In particular, for actual faults (see Figure 12), those averages are 10.5% for CR subjects, and 17% for R1 subjects; for true defects (see Figure 13), they are 6%, and 11.5%, respectively. Based on this data, and on the number of seeded faults (see Table 19) it seems that:

2. Inspecting time needed is directly related with the number of actual faults seeded.

Inspecting time vs. level of experience: Data, which we have considered for the point 2 above, also demonstrates that R1 subjects experience indicator (Moderately expert subjects) was high enough to detect faults more than BE subjects (Inexperienced subjects). Based on this data, it also seems that:

3. Inspecting time needed is directly related with the level of experience of subjects.

Concerning again point 3 above, Figure 14 obviously confirms that R1 subjects could not discharge their capacities within the inspecting time period indicated. In fact, as Table 24 shows, R1 subjects detected an equal number of actual faults for both techniques (3rd perspective), and nearly equal true faults as well (4th perspective), with maximum subject performance (2nd perspective).

5.4.2. Subject performance

Actions were done with a very high level of subjects' performances, as shown in Figure 15, which indicates that, at the end of the experiment duration time, subjects were not yet explaining the difference between effectiveness for both techniques. In particular, for R1, all the CR subjects and almost all the FTI subjects were still positively finding faults. This also explain the high rate of the p-value concerning R1 (see Table 22, 2nd raw). For BE, with less number of seeded faults and less experience for subjects, a small difference (5%) between CR and FTI inspection can be seen (13% for CR against 8% for FTI; see 3rd perspective in Table 24).

4. Moderately expert subjects are more effective than inexperienced ones.

5.4.3. Types of faults

We can see that types of faults seeded were not all detected, even with different levels of experiences and inspecting time intervals, though the number of actual faults and their detections were higher and significant for CR against FTI in BE (see 3rd perspective in Table 24). In our interpretation, this again confirms point 1 above, by indicating that:

5. More inspecting time period is required with higher expert subjects.

Let us consider now actual fault detections. Based on Table 20, for the most types of fault detections, it seems that:

6. Inexperienced subjects emphasize on actual faults of the Initialization type.

Vice versa, moderately experienced subjects place actual Initialization faults at the lowest priority level. In fact:

7. The best performances of moderately experienced subjects relate to Computation and Control actual faults.

According to Table 20, consider also that moderately experienced subjects were able to pass further levels than inexperienced ones; in fact, they detected actual faults that depend on interaction between different software-coded entities:

8. Moderately experienced subjects were able to detect actual Interface and Event types of fault. Inexperienced subjects weren't.

Let us consider now true fault detections. Table 21 shows the most common detections.

9. Initialization faults gained the highest score of true defect detections by testers for both techniques.

Meanwhile:

10. For Moderately experienced subjects, Interface and Event faults gained the highest score of true defect detections by testers for both techniques (see 3rd and 5th levels in Table 21).

Based on points above, the following main conclusions can be carried out: further investigation is still necessary. In order to see differences in the effectiveness between CR and FTI techniques, the following variables should be considered precisely:

- A. Inspecting time. This should be much more than two hours. Since we found that medium expert subjects were still strongly active in applying both techniques after two hours, when there was still no difference between technique effectiveness.
- B. Number of seeded faults. Techniques should be tested with a small number of seeded faults, but several types of them. However, faults should be seeded according to the probability of their occurrence.
- C. Subject experience. Subjects experience seems to strongly affect on techniques' performances.
- D. Types of faults. FTI seems to perform better than CR in detecting interface, event and control faults.

Let us now explore the aspects of the study that related to validity evaluation and assessment, and hence whether our study can justify the drawn conclusions.

5.4.4. Threats to the experiment validity

Concerning threats to conclusion validity. Conclusion validity is the degree to which conclusions can be drawn about the existence of a statistical relationship between treatments and outcomes. Due to participation on voluntary basis, and because of our small population size, it was not possible for us to plan the selection of a population sample by using one of the common sampling techniques, so we decided to take the whole population of our incoming classes as our target samples, whatever that population would be. A limited number of data values were collected during the operation of the experiments, due to the limited duration time and number

of objects and subjects. Regarding the basic experiment, some of the subjects were late in e-mailing their description files, the Internet connection went down before they accomplished their process, and hence we could not give interpretation and use the forms they had submitted. Consequently, the designed balance of subjects using both techniques was lost. For what concerns the quality of data collecting, we had screen-driven automatic data collection; hence data collection is not considered being critical (in addition, starting from R1, some form fields were checked at run time, and some data, e.g. timestamps, was automatically acquired by the system.) Finally, the quantity and quality of collected data and data analysis were enough to support our conclusions, as described in the previous Sections, concerning the existence of a statistical relationship between treatments and outcomes [67].

Concerning threats to internal validity. Internal validity is the degree to which conclusions can be drawn about the causal effect of the treatments on the outcomes. The experiment was under very strict control and was managed in a way that implicitly kept further independent variables, if any, at as fixed as possible values. Consequently, the observed relationship between the treatments and the outcomes seems to be determined causally rather than the result of a factor that we could not control or had not measured. Some noticeable differences should be noted between the basic experiment and its replications. In particular: Moderately expert subjects were trained both in class, lab and through Internet; inexperienced subjects got class training, did not get lab training, were allowed to get practical training through Internet on personal basis. Moderately expert subjects received checklists as part the experiment package; inexperienced subjects had to produce checklist by themselves. The screen type changed when we passed from the basic experiment to its first replication. The room available for subjects diminished when the experiment passed from a very large Internet Café to a quite small academic lab. A further threat to internal validity is concerned with the number of participating subjects, which was very less than the minimum needed (at least 35 for each technique, in our evaluation) both for BE and R1.

Concerning threats to construct validity. Construct validity is the degree to which the independent variables and dependent variables accurately measure the concepts they purport to measure. We adopted both well-defined and appropriate measures for the attributes of the entities we wanted to measure. Threats to construct validity of our experiments are concerned with following items. Objects: one experiment object was planned for all subjects. Both the experiment and training

objects were relatively complex, compared to the time made available for training and experiment (see threats to external validity in the followings.) Subjects: we chose to ignore differences within subjects in the same sample. Moreover, the experiments were planned to be also part of academic courses in which the students are graded. While we informed students that we would not be using the experiment results for grading, we felt that many of them were not quite sure that our mind would not change. Hence, our prediction was that, on one hand, many of them could show abnormally high performance, due to both being younger and more knowledgeable about OO Programming than average practitioners, and to the pressure caused by the wrong believe of being involved with an optional part of an exam; on the other hand, they could try to find as many defects as possible, without placing the necessary attention on categorizing them correctly, in the mistaken belief that the greater the number of defects they find, the greater their exam score. In other words, our subjects could really try to cheat and bias their data. Consequently, our plan for prevention was to keep subjects under the strict continual control with respect to both the system and observers.

Concerning threats to external validity. External validity is the degree to which the results of the research can be generalized to the population under study and other research settings. The greater the external validity, the more the results of an empirical study can be generalized with regards to actual software engineering practice. Some threats to validity have been identified which limit the ability to apply generalization. There is a threat that concerns the experiment duration time. The duration of both the experiments is short with respect to the assigned tasks. We had foreseen such a threat to validity during the experiment planning. In order to cope with this threat, BE subjects and observers reached the rented lab very early morning; unfortunately, the lab personnel was upgrading the system and, because of some troubles, it was not possible for us to approach the experiment operation before 10 A.M. Moreover, according to our limited budget, it had been impossible for us to plan the basic experiment to last a full day in such a large loft-like rented laboratory. For what concerns the first replication, because we were going to change both the type of laboratory and the level of experience of the participating subjects, we did not want to change the experiment duration time as well. We hence decided to use the same amount of time of 120 minutes. Successively, in order to deal with the threat of time availability, we conducted a further replication with a longer period of duration time; data collected are still under consideration and analysis. Further threats to external validity are concerned with the materials involved. Firstly, fault seeding did not follow distributions known from literature [54]; however, data

distributions are not yet quite available, which concern concurrent OO event-driven software. Moreover, we chose an experiment object (Solitaire game), which could lead FTI subjects to spend some time with reasoning rather than testing. This object might not be properly representative of event-driven applications. In order to cope with this threat, we organized the experiment objects in multiple versions, one for each level of difficulty. However, in order to investigate this perspective, our plan for future experiments is to extend the set of experiment objects by including a game, namely Flipper game, which requires subjects to reflex, to physical react, rather than to reflect. In conclusion, results could change with students of other universities or professionals, because of their different levels of knowledge and experience.

5.5. Lessons learned

We consider now the key lessons that we have learned to date from undertaking this empirical study. We assume an internal perspective as the main point of view, so the emphasis will be on lessons learned that concern subjects, process, and materials.

Concerning training: Training is an essential preparatory phase of any experimentation on software engineering topics. Training enables practicing on the techniques to be compared. To have sufficiently prepared subjects to execute the experiment in the right way is an important aspect for the research team. Let us show now how we trained those subjects through BE and R1, respectively. Training should be conducted both in class and lab. Using Internet only to train subjects usually results into an insufficient training, whatever the level of experience and maturity of the subjects might be.

Concerning experiment objects and other material: Initially, objects, forms, and documentation include many inaccuracies. Hence, they pass through many changes. In order to improve and eventually stabilize the experiment materials, it is very important to conduct a family of experiments rather than a single or few experiments.

Concerning the experiment process: An iterative-incremental experiment process model should take place rather than Waterfall and related variants. In fact, the initial experiment process passes through many modifications, and comes to an acceptable level of stability after that misunderstandings are well explained, mistakes are detected and removed, and essential tacit knowledge is made explicit. Moreover, we cannot discover all the experiment variables and requirements in the first iteration of the experiment planning, because many of them will be seen during the experiment operation.

Concerning experiment replication: In order to make experiments easy to replicate, an appropriate way is to design the experiment for running on an electronic net with the assist of automated data collection.

Concerning capitalization of experiences: In order to manage software experimentations at a reasonable capability and maturity level, software engineering organizational concepts and best practices should be extensively applied. Lessons learned from successful and unsuccessful experiments should be packaged for future reuse. Experiment objects, guidelines and process should be placed under change and version management. Each experiment process should be placed under workflow automation. Eventually the concept of Experience Factory ([3], [5], [7]), should be specialized [50] [6] to the ESE domain [17] [20].

Discussion of the part I

In the last three chapters of this study, we have presented four experiments that share a goal of evaluating the effectiveness of verification techniques for defect detection in OO software source code. Those experiments can be considered as two packages of empirical studies that consist of one basic and one replicated experiment; for those replicated experiments, we made several changes in the experiment design, while the experiment definition and planning remained as they are during the basic experiments.

For the basic experiments in both empirical packages, we notice the followings:

1. Subjects in the basic experiments are students in their first/second graduation year.
2. Object are OO software that consists of several abstract and concrete classes, roles of inheritance, aggregation, and association are used among those classes.
3. Classification of defects was "assumed" orthogonal; each defect in the software should be classified and assigned to one and only one defect type.
4. Static verification techniques are considered as the most effective input variables (independent variables).
5. Experiments are conducted in an off-line environment.

While For the replications in both empirical packages, we could notice that:

1. Subjects in the replication are students in their third/forth and fifth graduation year.
2. Objects are OO software. They consist of several abstract and concrete classes. Relationships of generalization, aggregation, and association are used between those classes.
3. Classification of defects was "assumed" orthogonal; each defect in the software should be assigned to one and only one defect type; seeded defects were three times more as often as in the basic experiment for the second package while the same number of defects was used for the first package.

4. Static verification techniques are considered as the most effective independent variables (factor).
5. Experiments are conducted in an off-line environment.

Inputs in the replicated experiments indicate that the experiment design has changed with respect to the basic experiments.

For what concerns the experiment results, collected data indicate that the use of different techniques for defect detection cause variations in the effect; however, the significance of such variations can be verified by applying for the statistical hypotheses testing. In particular, statistical testing results provide the study with strong evidences, which help to decide upon reading techniques. Differences in the performance of reading techniques for defect detection can be significantly or insignificantly evidenced by statistical tests; this means that techniques for defect detection are indeed performing differently; however, we would like to know how significant, from statistical analysis viewpoint, that difference is.

In fact, while one of the interests in software engineering research to understand "when" and "how" to use those verification techniques for defect detection, neither rejecting nor accepting the null hypotheses could explain, or is sufficient for explaining, reasons behind any experiments' results; that is because statistical test are observable rather interpretable, quantitative rather qualitative. In our understanding, we think that statistical testing of the experiment hypotheses is not enough for providing knowledge concerning software engineering; recently, what we need is to understand overall causes behind the effect rather limit to consider the effect of the explicit treatments as an end point.

Part II

Strategies for Manipulating Empirical Data

Chapter 6

Cluster-impact: An Approach for Manipulating Empirical Data with Fuzzy Sets

Abstract

Current chapter is an attempt to investigate on experimentation in software engineering, aiming to understand the complex of variables that cause effects when empirical investigations are targeting to OO software applications, an approach, so called Cluster-impact is defined by the meaning of the experiment input/design factors. Fuzzy sets are mapped to empiricism in software engineering for qualitatively analyzing the impact of the experiment input/design factors on the study focus.

6.1. Introduction

Several techniques for detecting software defects exist nowadays, which are based on inspectional activities, as enacted by humans on software artifacts. Instances of those techniques are [68]:

- **Review:** A process or meeting during which a work product, or a set of work products is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types include code reviews, design reviews, formal qualification reviews, requirements reviews and test readiness reviews.
- **Walk-through:** A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code and the participants ask questions and make comments about possible errors, violation of development standards and other problems.

- **Inspection:** A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards and other problems. Types include code inspections and design inspections.
- **Audits:** An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements or other criteria.
- **Reading:** A technique that is used individually in order to analyze a product or a set of products. Some concrete instructions are given to the reader on how to read or what to look for in a document. Reading is embedded in methods like inspections, audits or reviews. Therefore it is used as a technique for verification in the software development process (Here reading is used to find defects).

Moreover, as already mentioned several further techniques for detecting software defects exist. We collect these techniques under the following common name:

- **Testing:** techniques that are based on the evaluation of results from static analysis or dynamic execution of code, rather than human-enacted inspectional activities (see Appendix A2).

Concerning reading, a research question still remains to answer, concerning when to use reading, and what kind of reading to enact, for detecting defects in software artifacts. In fact, some reading techniques are traditional, such as Checklist-based reading, CBR, while other are specialized for OO software such as Systematic-based reading, SBR, and Functionality-based reading, FBR, for software frameworks.

In order to answer that question, we empirically verify software reading techniques; experimentation are one means to evaluate, explore, and compare the use of techniques such as reading or testing; this however requires an understanding mechanism of all those factors, not only the intended ones, that affect outcomes when enacting controlled experiments.

This chapter aims to investigate on software reading techniques for defect detection in OO software; however, this investigation is complicated by the facts that: **OO software studies require a body of knowledge.** The availability of laboratory packages for experiments can encourage better replications and complementary studies [58]; a body of software knowledge is a framework that mainly depends on the credibility, reliability, and reusability of the collected empirical data rather than the number of empirical packages that we might involve in the study; in other words, collecting data with low quality cannot be recovered by increasing the number of empirical packages in the framework.

- **Empiricism in software engineering is difficult.** While there are many benefits to experimentation, there are also some drawbacks, especially when human subjects are involved. Software developers' time is expensive, so the cost of running a software engineering experiment is high [23]. In empirical software engineering we deal with an artificial environment, which can be easily affected by large amount of variables that, in most cases, cannot be controlled in an extensive manner. Some of the variables that mainly affect software experimentations relate to the human factor; in fact, because software engineering is a human-based development process, software engineering experiments are eventually human-based processes too, whatever the level of technology supports might be; so variation in human ability tends to obscure experimental effects. Human factors tend to increase the costs of experimentation while making it more difficult to achieve statistical significance [9] [58].

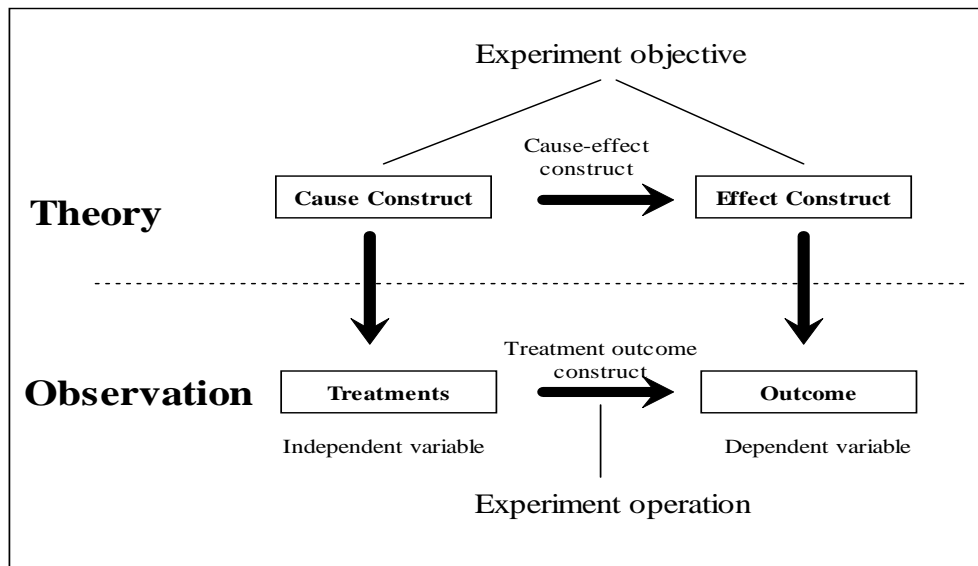


Figure 19: Principles of experimentation

6.1.1. Related Work

Several empirical papers exist that are related to software reading for defect detection ([1], [2], [4], [10], [15], [9], [10], [18], [27]). However, few are those studies that aimed at integrating empirical studies for building a common framework of knowledge ([9]), while few are those studies that aimed to propose related solutions ([48],[58]); and the same for those studies that aimed at understanding the specific impact on results of the experiment's different variables ([19],[23]).

The integration of empirical studies: In 1998, the Fraunhofer Center in Maryland, with the collaboration of the University of Maryland's Department of Computer Science (UMDCS ESEG), and the University of Bari Department of informatics, have proposed the idea of building a body of knowledge concerning empirical software engineering; their study [9] argues for the necessity of a framework for organizing sets of related studies. As a motivating example, a framework of empirical studies on reading techniques was presented, the goal was to determine if beneficial experience and work practices could be distilled into procedural form, and used effectively on real projects. The study was triggered from the hypothesis that reading techniques were of relevance and value to the software engineering community, since reading software documents (such as requirements, design, code, etc.) is a key technical activity; and so, research into reading could provide a model for how to effectively write documents as well: by understanding how readers perform more effectively, it may be possible to write documents in a way that facilitates the task. Several sets of reading techniques were experimentally involved for doing a specific reading task during the development process; Defect-Based Reading (DBR), and Perspective-Based Reading (PBR), which focused on defect detection in requirements, Use-Based Reading (UBR), which focused on anomaly detection in user interfaces, Scope-Based Reading (SBR), which consisted of two reading techniques that were developed for learning about an OO framework in order to reuse it.

The impact of the experiment variables: In 2003, in the Empirical Software Engineering Group at the University of Roma "Tor Vergata", UoRMTV-ESEG, we have conducted several versions of strictly controlled empirical studies [19] that compares the effectiveness of static vs. dynamic testing techniques for defect detection once applied for OO event driven Java software; no changes were made neither in the study definition, nor in the experiment planning, while the experiment design was changed for each version: levels of subjects' experience, numbers of seeded defects, and/or inspection time; the experiment objects were the only artifact that kept unchanged and fixed for OO even-driven Java software (realistic rather real or toy games). Some last minute changes occurred in the first experiment at conduction time, hence at the very beginning of the study; we had to adapt our design to those contingencies, in the hope that those changes on the experiment variables would cause negligible effect on the results from the planned family of experiments (basic experiment and replications); instead, however, because of those changes, we faced with a lot of difficulties when trying to gather those empirical studies and build an integral conclusion. One of the greatest challenges in such

experience was the fact that results for each version were quite different from the last; we finally figured that each experimental version of this study should be mainly considered as separated experiment (each requiring replications) rather than basic experiments and their replications.

Based on results above, the following Sections are related to those empirical studies which share a common goal, and are intended to argue on (i) Points of view that should be considered when running software experiments, and (ii) How data resulting from a single experiment should be evaluated and eventually integrated with data coming from other empirical studies.

6.1.2. Study motivations

Missing data benchmark: One of the main challenges in software engineering research nowadays is the lack of enough formal base of knowledge concerning software verification for defect detection. Researches in software engineering have established several techniques, methods, and tools for black-box and white-box testing, however, the question still remains concerning "what" and "when" to use among those testing techniques, tools and methods. In our understanding, what we need in software engineering research is a mechanism for gathering empirical data and building a reusable body of software engineering knowledge; with such a framework base of knowledge, experiments can be viewed as a whole rather than being isolated events.

Limited manipulation & analysis of data: In most of empirical studies in software engineering, we collect empirical data based on controlled experiments, we usually apply for some statistical analysis methods aiming to describe, and eventually analyze, the experiment data; results of using the statistical data analysis are of two phases: (i) Some experiments failed to decide upon the study hypotheses when statistical testing fails; (ii) Other experiments succeed in providing statistical significant evidence for the study hypotheses, however, results unexpectedly change when replicating the same experiment with some limited variants in design. In both cases, and for the sake of software understanding, we need to point at reasons behind such results; in fact, statistical analysis methods are not quite detailed to describe the experiment results; vice versa, in order to generate new hypotheses, more detailed should be needed, because software development, similarly to other human-based processes, requires deeper understanding of causes and impacts.

6.1.3. Study view and goal

Based on such motivations, we need a body of knowledge in software engineering; as shown in Figure 20, this can be achieved by (i) Conducting several controlled empirical studies that share a goal in order to observe the effect (Cause - Effect Construction), which is traditional, and eventually generate a benchmark of software knowledge; and (ii) Manipulating the empirical data in order to understand the impact of the explicit and implicit experiment factors on the effect (Cause-impact - Effect Construction).

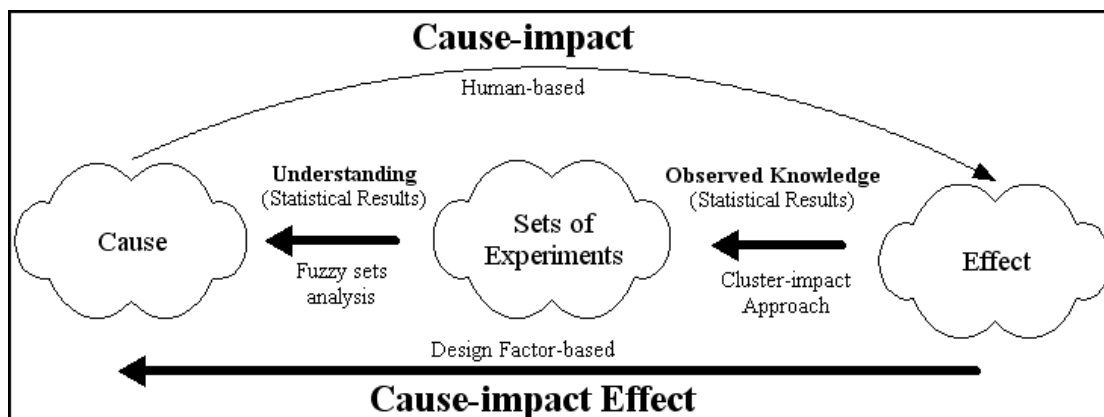


Figure 20: Building a body of knowledge

In other words, we reason on two points of view.

The one, which is traditional, is aimed to confirm the existence of a causal relationship between input and output for each experiment, i.e. to evaluate if, and in what measure, the outcome consequential to different levels of factors (treatments or input states) respects a hypothesized cause-effect construct.

The other one is aimed to investigate as a whole the experiments that share a common goal; essential for this view is data integration and manipulation from quantitative into a qualitative form. This would enable to consider clusters of interrelated input variables (e.g. those clusters concerning the "human-factor", like experience, subjects' understanding, team size etc.), to investigate the impact of those clusters on the output variables, and finally evaluate the interaction between a cluster and remaining experimental entities, like objects, infrastructures, utilized techniques, defects classification, and so on.

Concerning the latter point of view, Fuzzy sets might help to explain issues that statistical analyses methods fail to interpret; with respect to statistical analyses, Fuzzy sets require less rigorous conditions since this theory is based on qualitative, rather than quantitative inputs; however, Fuzzy sets are not quite reliable, and hence,

our decision is for using Fuzzy sets to support statistical analyses results rather replacing them.

The present study is an attempt to investigate the state of the art of experimentation in software engineering, aiming to understand the integral set of variables that cause effects when empirical investigations are targeting to OO software applications. The study goal is to build a body of formal software knowledge by adopting Fuzzy sets for contributing to an approach, so called Cluster-impact, that relates between the experiment's independent variables (the intended factors, on one side, and parameters and design variables like blocking-variables, on the other side) and the experiment outcome; the purpose is to understand the impact of input variables on the experiment outcome when the impact of some design variables are previously measured and benchmarked, in the context of several empirical studies that share the goal with variant changes in the experiment design, from the viewpoint of the researcher.

6.2. Understanding Cluster-Based Factors

Traditional experiments focus the attention on the set of intended factors, so called cause-constructs, and effect-constructs. The intended factors are independent sets of variables that are designed to change in the experiment; they hence build the effect-construct, which, as hypothesized, causes the dependent outcomes. Remaining input variables are parameters, intended by the experiment design to be kept at selected constant value, and design variables, e.g. blocking-variables; the latter are expected to change between elementary experiments; however, these variations are designed to statistically compensating each other, while all the elementary experiments are merged and considered as a whole.

With respect to such a traditional view, input factors are presented to enact the experiment and realize the effect. We propose here a different schema of understanding the experiment operation; it is necessary for this study to formalize the interaction between the input and output variables.

Based on the experimental hypotheses, during the traditional planning phase of the experiment, the experimental researcher define some independent variables ("Factors") as the experiment's inputs (Input Construct) in order to observe the output (Effect Construct); while during the experiment design, the experiment designer define other, hopefully all the remaining, independent variables that contribute to build the experiment (Design Construct).

Let us place here a first consideration concerning the traditional experiment planning and design: the independence of the “independent variables” is often a conjecture, which comes from the empirical relational systems of the scientist, and is assumed without further proof or verification: in other word, often we do not know if a relationship exists between some variables, i.e. whether some “independent” variable depends, in effect, from some other ones. For instance, in an controlled experiment aimed to evaluate relationship between some programming attribute, say productivity of programmers, and the programming language used, we assume language, development environment, experiment starting time and duration as independent variables: however, we really do not know whether each “standard” development environment has a preferred programming language. As an extreme example, we also ignore whether a preferred daytime exists for each programming language; consequently, when replicating an experiment, we do not mind to the starting time, in the assumption that starting time does not affect behaviors of programmers.

Concerning again the traditional approach to the experiment analysis and design, factors are called those independent variables, which impact on outcomes the designers aim to verify during the experiment; these variables are valorized by treatments (or levels), which are values assigned by design to those intended independent variables during the experiment. Let us recall that in experimental software engineering factors are usually qualitative variables, assuming a quite limited number of values (e.g., for programming languages: C++ and Java; for experience of subjects: Low, Moderate, High; etc.)

Beside those factors, there are other independent variables, which are those variables that, by design, are not intended to contribute to realize the effect. Many of these other variables are qualitative too; however some of them are quantitative: e.g. the experiment duration. Because we cannot assume linear relationship between variables and outcomes, we have to expect that values of all those other independent variables generally have an effect on outcomes too, e.g. they affect the average value, the form, and amplitude of the response. Moreover, the (intended) inter-experiment variations of those other independent variables can also produce different results. Finally, unintended intra-experiment variations and inter-experiment variations of those other independent variables (“noises”) can affect outcomes.

So we prefer to call “Input Factors” the former (i.e. the intended independent variables), and “Design factors” the latter, including noises. Consequently, we call “Experiment factors” the union of input and design factors.

Of course, similarly to input factors, also design factors are kept in control during the experiment conduction through measurements.

For instance, in a basic experiment, designers make decision about the input factors, their levels to use in the experiment, and response variables to investigate; they also identify design factors, and fix each of these to some constant value (“parameter”). Outcomes of the experiment and related conclusions, including statistical significance or insignificance of results, are hence valid for that set of constants. Such a simple scenario becomes infeasible when, in an experiment, some design factors cannot or should not be controlled at constant value through individual experiments, e.g. to prevent threats to the validity of the experiment results. In other words, in order to cope with intended or unintended (“noises”) variations in some design factors, designers might requested to introduce redundancies in their experiments, in the aim of having those variations to compensate one each other. Finally, designers may want to change values of parameters through many replicated studies, aiming to observe the impact of those changes on the experiment output. Again, our conclusion is that design factors can directly or indirectly contribute to the experiment output because they act during the experiment operation.

As shown in figure 21, input factors are presented once the experiment operates; input factors start interacting with the experiment design factors; and so, interactions that involve the input and design factors produce the effect; however, at this point, based on traditional approach to software experimentation, while we are already able to evaluate impact of input factors on outcomes, we are still unable to formally define interaction between input and design factors, if any, as well as to measure the impact of design factors on the input factors, and the impact of design factors on the output.

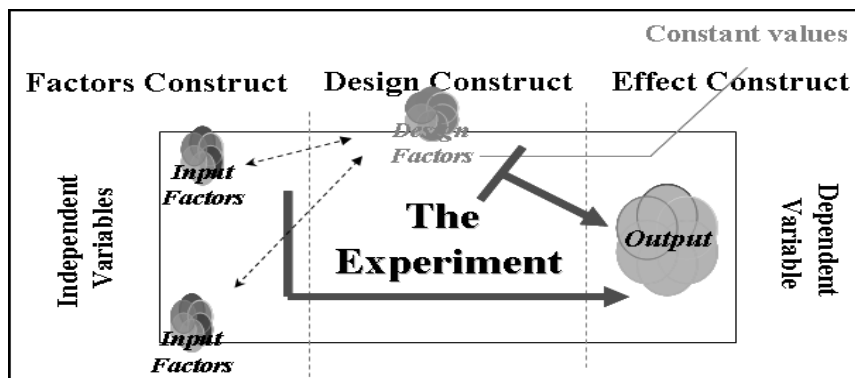


Figure 21: The impact of input/design factors on the output

6.2.1. Cluster-impact approach

Cluster-impact is concerned with investigating a cluster of cause factors on the objective of the empirical study.

Because our expectation is that understanding the integral set of experiment factors would help to understand several issues concerning the study objective, through Cluster-impact, we focus on the experiment factors as often as the study focus.

The goal of Cluster-impact approach, therefore, is to define a construction of cause factors for understanding their impact on the focus of an empirical study.

Cluster-impact approach is based on following definition and characterizations.

- **Clusters:** in our definition, a cluster is any part of the integral set of experiment factors. The following point “Layers” explains how experiment factors are partitioned. Because factors impact on outcomes, also clusters have an impact on response variables.
- **Layers:** Concerning partitioning of the universe of Experiment Factors (EF), the level of indirection with which factors impact on the experiment focus constitutes an Equivalence Relationship on EF. In other words, we can proceed by clustering first (L_1) the factors of EF that directly impact on the focus of the study, hence the experiment outcomes. The remaining factors, $(\{EF\} - \{L_1\})$ indirectly impact on study focus and outcomes: i.e. their impact on study focus is explained through factors in L_1 . We can recursively apply to factors in $\{EF\} - \{L_1\}$ the partitioning criterion that generated L_1 , so having factors partitioned in nested layers L_2 , L_3 and so on, up to reach the innermost layer, which includes slightly influent factors only. We can transitively close the above relationship by defining as L_0 the experiment focus that we are investigating, which is part of software knowledge. In conclusion, the layer $L_{n \geq 0}$ has deepness “n”; $L_{i > 0}$ impacts directly on L_{i-1} only, and the more deep is a layer, the less relevant is its impact on the study focus and experiment outcomes.
- **Strata:** Our Cluster-impact approach is based on the further assumption that, for each layer, we can always identify one representative factor: this is the layer’s main factor; it defines and strictly controls some specific impact on each of the other factors in the cluster. Eventually, a layer can be divided in two strata: one lower stratum (S_1), which includes the main factor; and one upper stratum (S_0), where remaining factors are located in the layer.
- **Dependency on study focus:** Partitioning of a given universe of factors depends on the study focus: if we change the study focus then the partitioning does change. For instance, when studying code reading with respect to functional testing, some main design factors that impact on the effectiveness might differ

from those concerning efficiency (say the types of defects seeded, and the experiment duration time, respectively).

Finally, let us note that, for any stratum of type S_1 , the elements (design factors) might interact on each other. For instance, if objects of the type Fortran function or subroutine are chosen by design, then defects cannot be seeded, which can be classified as Class, Inheritance, and Concurrency, or typed as Polymorphism.

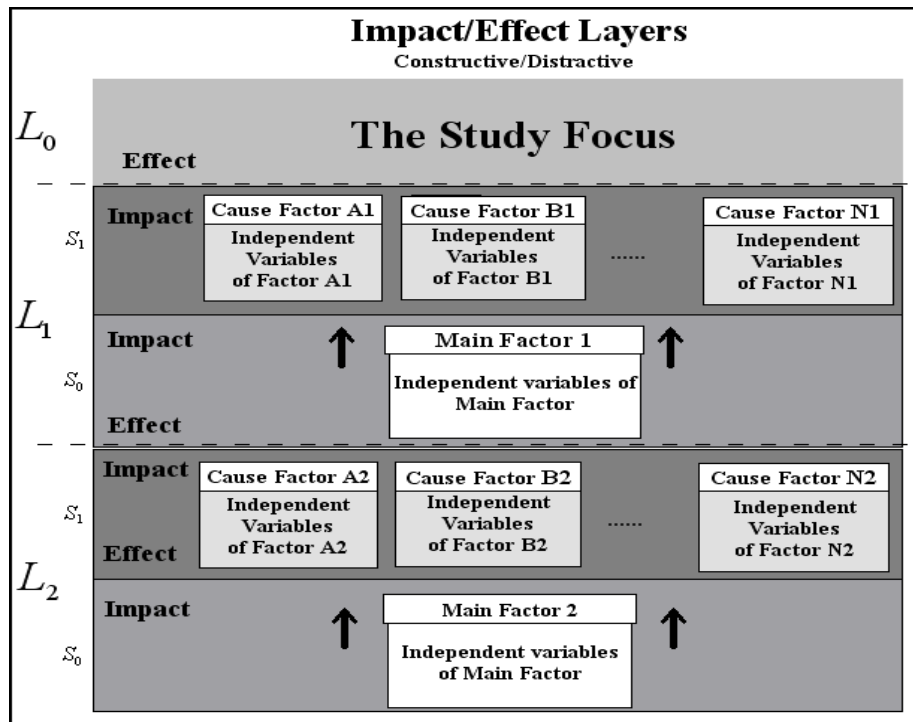


Figure 22: Gathering and claustring of independent variables

6.3. Fuzzy sets for data manipulation

Prof. Lotfi A. Zadeh of UC/Berkeley introduced Fuzzy sets theory in 1965 [43]; it is an extension of the conventional ("crisp") set theory. It handles the concept of partial truth (i.e. truth values between completely true and completely false) as one means to model the vagueness and ambiguity in complex systems.

In other words, when data cannot be fitted into a known distribution, we can use Fuzzy sets to get some understanding from those data.

Such as between any two real numbers there exists an infinite set of numbers, the idea of Fuzzy set theory is based on assuming that between 0 (the False logical value) and 1 (the True logical value) there exists an infinite number of partial fractions of truth that might be used to indicate some kind of membership grading of the

elements in a given Fuzzy set A ; this number in the range from 0 to 1 is a Fuzzy measure and is called Membership grade or simply Grade.

Basically, a Fuzzy set is thus a set - which belongs to the universal set under investigation - that contains elements each with a predefined Grade of membership to this Fuzzy set; this membership grade lies between truthfulness (represented by 1) and falsity (represented by 0) of certainly being a membership of this set.

Fuzzy set theory has been implemented in many scientific areas such as natural life and social sciences, engineering, medicine, management and decision-making, etc.; several researching studies were conducted with Fuzzy set theory's [43]; therefore, we think that Fuzzy sets also can be applied in empirical software engineering as well.

Let us recall the concept of Fuzzy set theory. Once given:

- A phenomenon X under investigation.
- A universal Fuzzy set A of X .
- A Fuzzy relation \mathfrak{R} presenting more advanced information about A . \mathfrak{R} is also an equivalence relation on A , which generates the crisp Fuzzy sets $A = \{A_1, A_2, A_3, \dots, A_n\}$, i.e. classes of A .

The Fuzzy set B of all possible elements of A that are relative to X by the given \mathfrak{R} can be represented as in the following:

$$B = A \circ \mathfrak{R} \quad \text{Equation [1]}$$

The Equation in Equation [1] explains the logical term "if A then B by \mathfrak{R} ". In other words, B represents the Membership grades of the observed elements of the Fuzzy set A by the Fuzzy relation \mathfrak{R} , which is the relationship grade between elements of the Fuzzy sets $A = \{A_1, A_2, A_3, \dots, A_n\}$ in X .

6.3.1. Membership grades of Fuzzy sets

Let us consider crisp subsets of the universal Fuzzy set A , and a function that returns a value between 0 and 1 for each element in that universal set. Such a function is a grading function; because such a function can be used for discriminating between members and non members of the Fuzzy set under consideration, that function is a characteristic function of the Fuzzy set membership. In conclusion, each membership

function maps elements of a given universal set X , which is always a crisp set, into real numbers $[0, 1]$.

Let A be a Fuzzy set on X , x an element of Universal set U of the phenomenon X ($x \in U$), and $A(x)$ a Membership grade function, i.e. it is interpreted as the degree to which x belongs to A [43]. Membership grade functions are continuous function, and their boundaries lie between 0 and 1 (see Appendix A1).

Moreover, let cA denote the complement of A to the Universal set U , $cA(x)$ be a Grade function of cA . Then, $cA(x)$ may be interpreted not only as the degree to which x belongs to cA , but also as the degree to which x does not belong to A . Similarly, $A(x)$ may also be interpreted as the degree to which x does not belong to cA .

Furthermore, functions c can be defined, which assigns a value $c(A(x))$ to each Membership grade $A(x)$ of any given Fuzzy set A . c is a Membership grade function of a Membership grade function ($A(x)$); any value $c(A(x))$ is interpreted as the value of $cA(x)$.

Finally, any Membership grade function is equilibrium functions with respect to $A(x)$, that is: $\forall x \in U(X), c(c(A(x))) \sim A(x)$ (see Appendix A1).

6.3.2 Fuzzy Membership Function

In empirical software engineering studies, and mainly those studies which use human resources as subjects, most of the experiment dependent variables (where we can see the effect of change in the independent variables) are distributed in the continuous form $f(x)$, that is, the change on the dependent variable is always related to some certain changes in one of those, known or unknown, independent variables, whether increasing or decreasing.

As already mentioned, we can use Fuzzy sets when we are unable fit the true distribution $f(x)$ of X . In fact, one of the advantages in Fuzzy sets is that Membership grade of Fuzzy sets ignores the true distribution of X , which is, probably, continuous function. Instead, Fuzzy sets apply for a function $c(A(x))$ to realize Membership grade of the Fuzzy set.

The function $c(A(x))$ is both defined, and assumes value, in the real range $[0...1]$. Hence, it does not depend directly on the observations x . Consequently, while working on c we may ignore x and $f(x)$.

However, in order to use $c(A(x))$, we need to keep track of elements x , in order to eventually obtain the Membership grade $A(x)$. The fact is that we are unable to get the Membership grade $A(x)$ from an observation x . However, we already know that

$A(x) \cong c(c(A(x)))$. So what we can try is to get $c(A(x))$ from x . To achieve such a result, we can apply for the Sugeno Class [43] in Equation 2, which defines membership grads of the Fuzzy sets.

$$c(A(x)) = \begin{cases} \frac{(\sqrt{1+x}) - 1}{x} & \text{for } x \neq 0 \\ 0.5 & \text{for } x = 0 \end{cases} \quad \text{Equation [2]}$$

6.3.3 Evaluation of Fuzzy Membership Grades

Finalizing the construction of membership grading requires manipulation of data that resulted by applying equation [2]. We first complement those data to 1, so having values quite equal to Fuzzy Membership grades. In order to reason on differences between these Fuzzy memberships, we proceed to classify Fuzzy membership grades in a qualitative manner.

Table 25: Formalization of membership grads

Interval (Membership Grade)	Measure of Occurrence
0.00 to 0.10	Never (N)
0.10 to 0.30	Very Seldom (VS)
0.30 to 0.60	Seldom (S)
0.60 to 0.70	Often (O)
0.70 to 0.85	Very Often (VO)
0.85 to 1.00	Always (A)

As shown in Table 25, grade ranges are mapped onto an ordinal scale, and those scale values are eventually assigned to elements of Fuzzy sets. Figure 23 shows another view of such an ordinal classification.

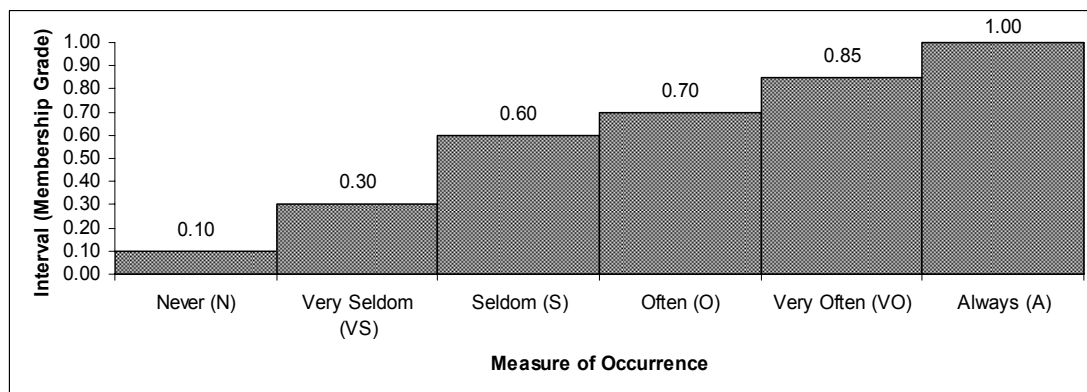


Figure 23: Formalization of the membership degree classification

Membership grades, therefore, are now defined in a qualitative manner. In order to distinguish between any two memberships grades in the same interval, when due, we can use the term "More than".

6.4. Organizing experiment-based study in software inspection techniques for defect detection

Building empirical knowledge requires a large number of empirical studies which, in turn, can be achieved either through simulating an empirical study for many turns or, fiscally, by conducting several empirical studies with several replications and few changes in design factors. However in our understanding, any kind of data simulation is strictly related to the need of, at least, data benchmark, and so, using data simulation before achieving an integral and formal definition of the interaction between the experiment factors, could limit the validity of integration and formalization of the resulted knowledge.

Building a body of knowledge, therefore, requires proceeding with two qualitative mechanisms: integration and formalization; data which are collected from different, but related, empirical studies can be integrated by tracing those collected data up to reach a certain level of conformity among those data (i.e. the average, median, mode etc.); eventually, we need to qualitatively formalize those integrated data in a way that permit to reuse them to support any other similar studies, without need of proving their validity for the specific new context.

According to discussion above, building empirical knowledge from a family of empirical studies requires designing and data manipulation phases. Cluster-impact approach is used, in the definition and design phases, for designing the interactions between/within the experiment factors, while empirical data from several empirical studies are manipulated with Fuzzy sets; eventually, a process of qualitative formalization is built to obtain results.

6.4.1. Definition, planning, and design

We aim here to adopt Fuzzy set theory for qualitatively manipulating the empirical data in software engineering; this requires determining the following:

- **Study focus:** which empirical knowledge we want to improve. In particular, in this study, we are focused on investigating the effectiveness of several reading techniques for defect detection in OO software frameworks.
- **Infrastructure:** A benchmark, hence data from a number of replicated studies of controlled experiments that share the focus. Two different, but related in goal, empirical packages of replicated experiments are utilized; the first package resulted from evaluating the effectiveness of reading techniques for defect detection in OO C++ software frameworks, while the second package resulted from comparing the effectiveness of functional testing and checklist-based reading for defect detection in OO event driven Java software.
- **Input and design factors:** the independent variables, which impact on the study focus. In particular, for the present study, we discuss four main factors that have an impact on the study focus: subjects, objects, defects, and three reading techniques for software defect detection.

6.4.2. Operating with Fuzzy sets

In a certain phenomenon X that represent an experiment for a certain objective, given that:

- T and \check{D} are any two different experiment factors, in the universal Fuzzy set of the experiment X (Example: T values are reading techniques, and \check{D} values are observed true defects.)
- Q is the observed impact of T on \check{D} in X : $Q = \check{D}(T)$. (Example: Q is the set of true defects that a certain technique observed.)
- S and D are any two different experiment-factors, and $(S \bullet D)$ is their Cartesian; the impact relationship of S on D is known (Example: S is the Subjects' experience and D includes categories of true detected defects.)
- $\mathfrak{R}(S \bullet D)$ is the Fuzzy relation, which presents the impact of S on D ; it indicates the Membership grade of known relationship between each $s \in S$ with each $d \in D$ (see Figure 22 for Main Factors).

Then:

- According to Equation [1], we can construct Fuzzy set $B \subset T \bullet S$, which presents the degree in which members of T are related with members of S :

$$B = Q \circ R$$

or, equivalently (see Figure 24):

$$B(s \in S, t \in T) = \max(\min(Q(t), \mathfrak{R}(s))) \quad \text{Equation [3]}$$

where $Q(t)$ is the Q 's row $Q(t, \text{any } \tilde{d} \in \tilde{D})$, and $\mathfrak{R}(s)$ is the \mathfrak{R} 's column $\mathfrak{R}(s, \text{any } s \in S)$.

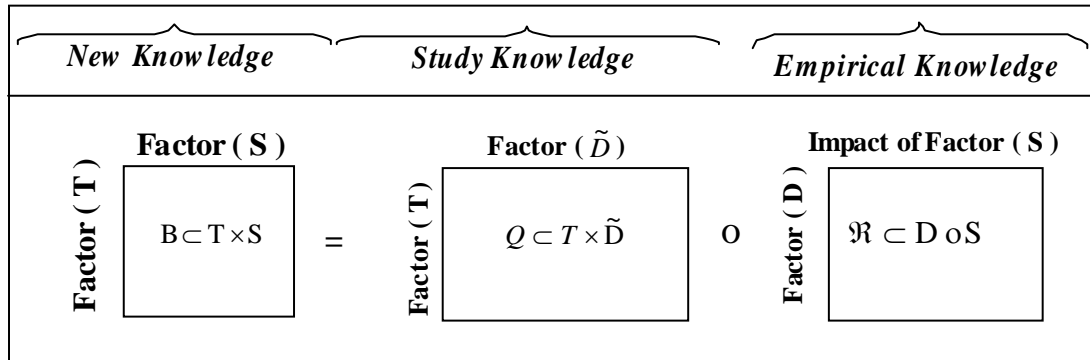


Figure 24: Fuzzy sets and Cluster-impact

In Equation [3] and Figure 24, the set \mathfrak{R} denotes a Fuzzy relation that presents the impact of S on D , with S and D any two different but related factors; those factors have had interaction with each other for many several replicated experiment, and showed to have relations among them. \mathfrak{R} becomes a main relation when considering an interaction that involves the Main Factor (S in our case) for a certain layer, which presents some available empirical knowledge (see Figure 25).

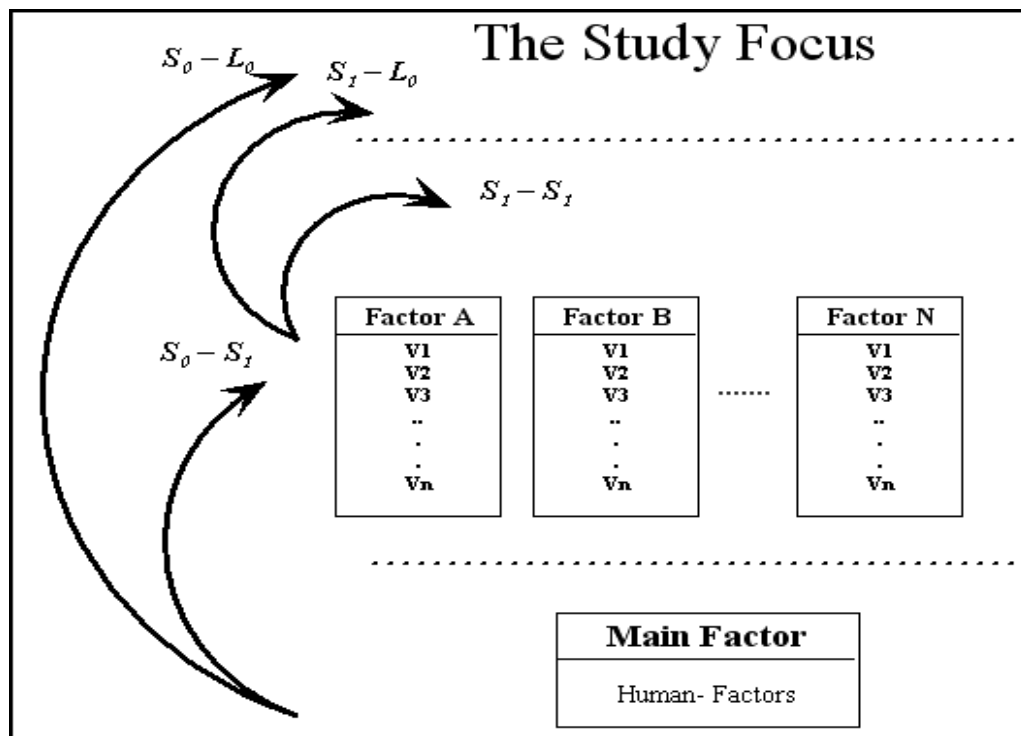


Figure 25: Fuzzy relations between the experiment factors

In addition, Q is a Fuzzy set as well; the set Q presents the observed impact of the experiment factor T on the experiment factor \check{D} for a certain empirical study.

In other words, membership grades of the Fuzzy relation R is built, or decided, based on previous evaluations, experience and knowledge, while membership grades of the Fuzzy set Q is observed from a certain study or experiment. Manipulation of data is made based on the following steps:

1. **Data integration:** which can be made by collecting data from previously conducted empirical studies concerning: S and D , in previous reasoning and Figure 24. Involved experiments should be valuable and have an acceptable level of significance; data that are related to both S and D are collected individually from each experiment as a first step; a second step is to normalize those data based on the maximum value of S , the Main Factor (e.g. the greatest number of participant subjects; that is, upgrading experiment results based on the highest number of subjects involved in the experiments); finally, data integration is made by presenting averages of the collected data concerning S , D , T , and \check{D} for each experiment.
2. **Data Transformation:** after that we had integrated our empirical data, the next step is to transform those integrated data from being quantitative to explorative, so having Membership grades of Fuzzy sets; Sugeno Class [43] in equation [2] is used to realize such explorative data.
3. **Data Manipulation:** this step aims to obtain several sets of input/design factors that intersect with each other in order to evaluate the impact. Cluster-impact approach emphasizes on human-factor as one of the Main Factors in experiments; the relationships of the experiment factors with human-factor, therefore, are necessary for understanding the impact of the latter on the former (Figure 26- manipulation levels S_0 - S_1 and S_0 - L_0). In addition, the impact of non-Main Factors on the focus of the study is also essential for understanding the effect (Figure 26- manipulation levels S_1 - L_1). Finally, the impact of non-Main Factors on each other can be also obtained but with less value of reliability, that is because there is no Main Factor involved in such operation (Figure 26- manipulation internal to level S_0).
4. **Data Formalization:** Fuzzy membership function is used in order to give our new obtained data a degree to which we can easily assign them to a qualitative measure; by using Table 25 above, we may turn data from being explorative to qualitative (membership grades taking values between 0 and 1).

Chapter 7

Cluster-impact Evaluation on the Effectiveness of Reading Techniques for Defect Detection

Abstract

This chapter is a test case that aims to validate the ability to map Fuzzy sets theory in empirical software engineering, and, in particular, (i) to investigate on factors that contribute on optimizing defect detection when reading techniques are used for inspecting OO software; and (ii) to compare the impact of those factors on three reading techniques for defect detection with regard to OO software from different viewpoints.

7.1. Introduction

As we described in chapter 6, there are many factors that cause impact on the experiment results (see section 6.1), each factor presents a set of independent variables; these interact with each other and with other variables of other factors. Research is trying to understand the effect of changing the independent variables of those factors by focusing on one or two of them, while many other factors are considered as side effects, though that the effect should be considered from different viewpoints.

7.2. Building a Body of knowledge from families of empirical studies

In order to employ Fuzzy sets in the area of experimental software engineering research, we need to trace the collected data and build some common and formal empirical knowledge. Based on the requirements stated in Section 6, we define the followings.

7.2.1. Study focus

The study focus of this Chapter is to evaluate the ability of using Fuzzy sets theory in empirical software engineering studies, focusing on different factors that cause changes on software reading for defect detection, from the viewpoint of the researcher, with respect to defects of different types, real OO software, and subjects with variant levels of experience and knowledge.

7.2.2. Infrastructure

In order to apply for Fuzzy sets, we need to expertly manipulate data collected from several controlled experiments with similar objectives; in such manipulation, the more number of empirical studies we involve is, the more reliable results we have.

In our case study however, we will be considering two packages of empirical studies, which thought to have an acceptable level of quality, similarity, and reliability.

First Package: this package consists of two experiments ([1], [2]), the basic experiment, FPK_BE, and a replication of the basic experiment, FPK_R1. The empirical study of such two experiments aims to investigate on the effectiveness and efficiency of reading techniques for defect detection with regard to C++ OO software frameworks.

Table 26: Variables, factors, and results of the first empirical package

Objective	Type	FPK_BE	FPK_R1
Variables	Dependent	Effectiveness and Efficiency	
	Independent	Reading techniques (CBR,FBR,SBR)	
Factors	Subjects	<ul style="list-style-type: none"> • Junior and sophomore German students • Different programming skills • Arranged into groups of 2 subjects • Number of groups: CBR(8),FBR(10),SBR(7) 	<ul style="list-style-type: none"> • Third and fourth year Italian students • Different programming skills • Arranged into groups of 2 subjects • Number of groups: CBR(9),FBR(9),SBR(11)
	Objects	C++ OO application for Telecommunications, more than 1000 LOC, with 3 abstract and one concrete class, and one abstract interface, cabling between 2 and 7, 53 method.	
	Defects	Defects were seeded according to the following: A: Initialization (4), B: Computation (3), C: Control(3), D: OO-Messages or -Relationships (Polymorphism) (1)	

Details concerning the experiment process, data analysis, discussions and conclusions are located in chapter three of this dissertation. However, we present in

Table 26 some important notes concerning the basic experiment and the first replication.

Second Package: the second package of empirical studies consists of two controlled experiments, the basic experiment SPK-BE and SPK-R1, the empirical study of those two experiments aims to investigate on software testing strategies comparing the effectiveness and the efficiency between code reading and functional testing for defect detection for OO event driven Java software; details concerning the experiment process, data analysis, discussions and conclusions are located in chapter five of this dissertation. However, we present in Table 27 some important notes concerning the basic experiment and the first replication.

Table 27: Variables, factors, and results of the second empirical package

Objective	Type	SPK_BE		SPK_R1	
Variables	Dependent	Effectiveness and Efficiency			
	Independent	Software Testing Strategies (Code Reading, CR, and Functional Testing, FT)			
Factors	Subjects	<ul style="list-style-type: none"> • First year Junior students • Same programming skills • One subject for the same object • Number of subjects: CR(12),FT(8) 		<ul style="list-style-type: none"> • Third year students • Same programming skills • One subject for the same object • Number of subjects: CR(13),FT(12) 	
	Objects	Toys made by Java OO driven software, more than 384 LOC, 13 different relations between classes and 33 methods.			
	Defects		A: Initialization B: Computation C: Control D: Polymorphism E: Interface F: Function G: Event H: Exception	7 6 5 5 6 6 1 2	A: Initialization B: Computation C: Control D: Polymorphism E: Interface F: Function G: Event H: Exception

7.2.3. Factors and variables

As shown in Figure 1, there are many empirical factors that have an impact on the objective of the study; based on Cluster-impact approach, our cluster under investigation is defined by the Main Factor of type "Inspectors for Defects" (see also Figure 26). The set of independent variables that belong to this factor are mostly fixed to some constants, while the effect level contains objects of the experiment, techniques for defect detection, and the classification of defects.

Our hypotheses are:

- Subjects' experience affects the number of true detected defects.
- Subjects' experience affects the effectiveness of reading techniques.
- Subjects' understanding of objects affects detecting defects in complex OO classes, in particular (abstract and concrete classes).

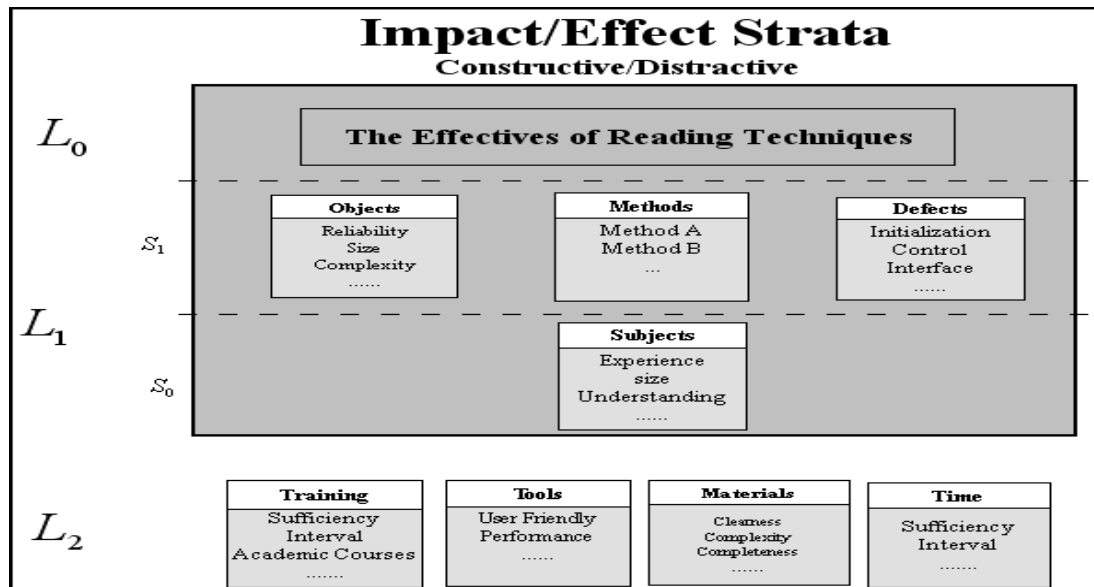


Figure 26: Cluster-impact Approach is used to define a case study. The cluster {Subject, Method, Object} is selected as the case study for defect detection in OO software.

7.2.4. Study Materials

In order to operate on Fuzzy sets, several materials are need to be defined and made available. Resources of those materials are the available empirical packages which will be used to build a base of knowledge on the cluster of factors under study.

a. Readers (Subjects): are the key factors for the goodness of any empirical data. The attempt to formalize the impact of such factor is important for the success of any future empirical efforts [55]. Software developers' time is expensive, so the cost of running a software engineering experiment is high [21].

In order to synthesis on subjects of the two packages, four experiments, we can classify them as in the following:

Table 28: The interaction between different empirical factors

Empirical Package		Classification (Sp: Sophomore, Jr: Junior, Sr: Senior)	Category
Impact	Assignment		
FPK	BE	Sp,Jr	Basically Experienced (BExp)
	R1	Sr	Average Experienced (AExp)
SPK	BE	Sp	Basically Experienced (BExp)
	R1	Jr,Sr	Average Experienced (AExp)

b. Defect Types: Our investigation with Fuzzy sets requires interacting with a certain defect categorization. Instances of such categorizations are based on fault-nature, e.g. {Omission, Inconsistency, Ambiguity, Incorrectness, Syntax}, fault-severity, e.g. {Low, Moderate, High}, or fault-type. This study is concerned with a categorization of defects based on their type: in fact, a stable and quite standard categorization of defects exists, which is based on types [35]. In an experiment, objects are expected to be seeded with a certain number of defects that are corresponding to their realistic possibility of occurrence.

Treated defects are taken from the first empirical package, while some data in the second package concerning defect types will be considered as well. Types of defects on which the study is concerned are of four types: A: Initialization, B: Computation, C: Control (see Table 29). Choosing those types of defects is referred only to the lack of data concerning the other classification of defect types.

Table 29: OO Event-Driven Defect Classification. Extends IBM Orthogonal Defect Classification [35])

Code	Category	Description
A	Assignment Initialization	Incorrect implementation or invocation of a constructor, missing or incorrect initialization of data, wrong assignment.
B	Algorithm: Computation Method	Faults that lead to incorrect calculations in a given situation or incorrect method.
C	Algorithm: Control	Faults that lead the program to follow an incorrect control flow path in a given situation.

For each reading technique in the first package, Table 30 presents the collected true defects for both experiments, FPK_BE and FPK_R1, classified by, defect types as classified in Figure 28, and the number of groups who detected those defects.

Note that, groups of subjects were not equivalent. In order to eliminate the difference in groups between the two, a simple mathematical calculation is made, and resulted values are presented in Table 30 and Table 31.

Table 30: The number of true defects classified by the first package, defect types, and the number of groups who detected those defects for each reading technique

Techniques	FPK_BE				FPK_R1			
	Groups	A	B	C	Groups	A	B	C
CBR	9	2	0	2	10	7	1	3
FBR	9	2	5	5	10	8	9	19
SBR	9	3	1	1	10	13	10	11

Table 31: The number of true defects classified by the second package, defect types, and the number of groups who detected those defects for each reading technique

Techniques	SPK_BE				SPK_R1			
	Groups	A	B	C	Groups	A	B	C
CBR	9	25	5	1	10	26	15	29

c. Techniques: Empirical studies for defect detection on source code employ several reading techniques that aim to ease the mechanism of navigation through source code of software. The goal is to compare the effectiveness of those navigation techniques in detecting defects. Technique's definition and mechanism might vary from one experiment to the other; we will be focusing on three reading techniques for defect detection (see Chapter 3 for more details concerning reading technique), Checklist-Based Reading, CBR, Functionality-Based Reading, FBR, and Systematic-Based Reading, SBR.

In order to realize Equation [3], we need to define the occurrence Fuzzy relation set $Q \subset T \times \check{D}$ (see Chapter 6), which relates both reading techniques with detected defect types; this requires to treat subjects' experience by means of some constant values (qualitative or quantitative).

d. Objects: objects are needed when conducting any experiment in empirical software engineering studies; in particular, the intention of this case study is directed to those objects of type "software", which are developed using OO design and programming.

Table 32: Summary statistics of classes to be inspected

Classes	Type	LOC	# Methods	Relations
Class A	Abstract	209	15	5
Class B	Abstract	117	11	3
Class C	Concrete	718	13	7
Class D	Abstract	71	14	2
Total		1115	53	17

As shown by Table 32, FPK object is a C++ software framework, which is developed by Intel. This framework is known as Telephony Devices Connector, which simply supports telecommunication domain [36]. A partial hierarchy of this framework is considered as the experiment object; this hierarchy (see shadowed parts in Figure 27) is made by 4 classes, three abstract and one concrete class; lines of code are more than 1115 LOC with 53 different methods (overridden and fully-implemented), and 17 different types of OO relationships.

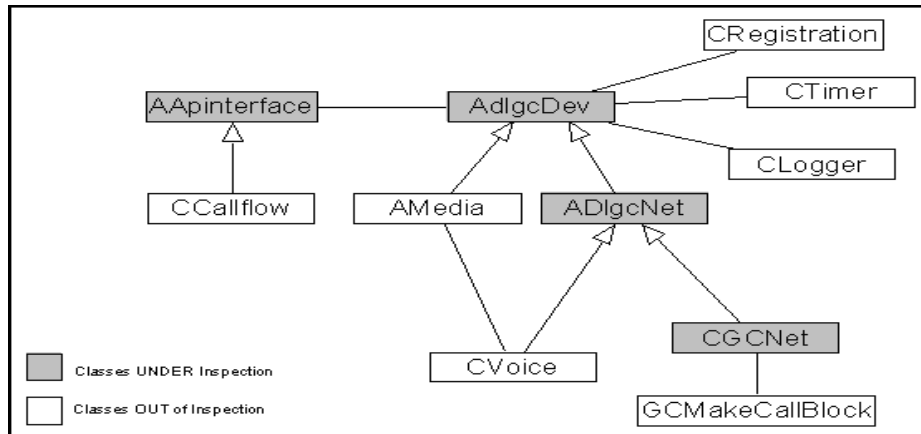


Figure 27: Class diagram for FPK

In order to build a body of knowledge, the approach we intend to enact is to collect data from several empirical packages. However, collected data that are relative to objects in the targeted empirical packages are different or insufficient; for instance, in the first package, the experiment object is real, and developed in C++, while in the second package the experiment object is a few more than a toy software, and developed in Java; in addition, the first package is a software framework which is a real product, while the second package is an event driven OO software which was made for teaching issues.

7.3. Preparation and data management

According to Figure 26, we need to investigate on some interactions between the experiments factors; those interactions are shown in Figure 28.

The descriptions (see Figure 28) S, T, D, and O are factors to which evaluation will be made with respect to P (as headers of Subjects, Techniques, Defects, Objects, and Positives, i.e. true defects, respectively). The relation between S and P (interaction denoted by No.1 in Figure 28) is obtained from the researcher experience and the previously collected empirical data by reasoning on a process of constructing a body of knowledge; this in turn can be achieved first by defining the study Materials (see Section 7.2.4), and second by proceeding with a qualitative mechanism of a generalization and formalization, which comes later in this chapter. Moreover, relations between/within other factors and the study focus can be achieved by the meaning of interaction No.1 in Figure 28. That is, whenever the Fuzzy relation between S and P is known, there are several other Fuzzy relations that can be defined and eventually established. Such Fuzzy relations define both, the impact between some factors and the study focus (interaction denoted by No.2 in Figure 28),

and the impact within the experiment factors themselves (interaction denoted by No.3 in Figure 28).

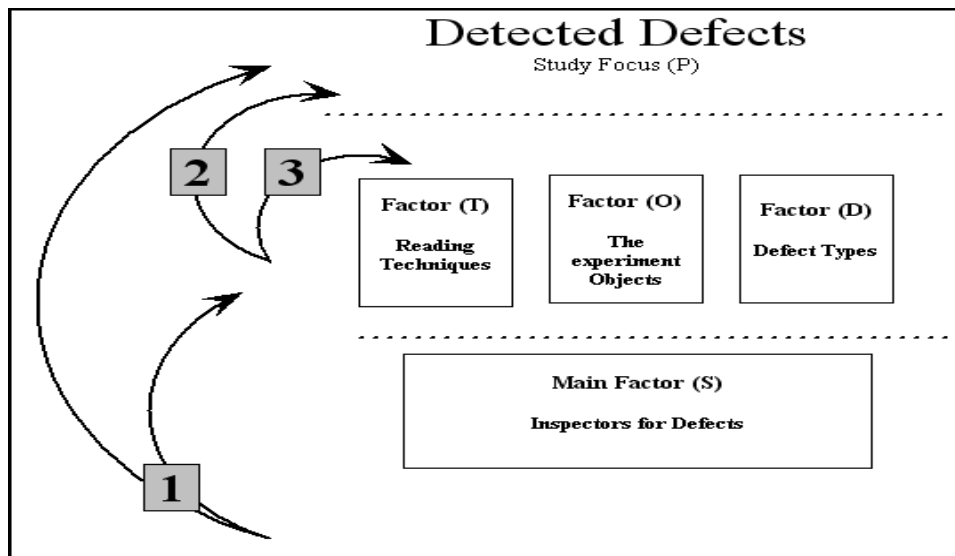


Figure 28: Interactions between the experiment factors

Therefore, our attention is to establish a specific set of Fuzzy relations concerning the interactions shown by Table 33.

Table 33: The interaction between different empirical factors

The Impact of A on B	Operation		Fuzzy description
	Affect	Effect	
A. Level of subjects' experience B. True defects Detected	Subjects (S)	True defects (P)	$R = C_{SP}$
A. Level of subjects' experience B. Inspection techniques	Subjects (S)	Techniques (T)	C_{ST}
A. Level of subjects' experience B. Objects per Class type	Subjects (S)	Objects (O)	C_{SO}
A. Level of subjects' experience B. Types of defects	Subjects (S)	Defects (D)	C_{SD}

7.3.1. Membership grads of Fuzzy sets

As we described in Chapter 6 (see Section 6.4.2), we can reason upon the collected empirical data in Table 30 and Table 31, and the continues Fuzzy set membership grades function $c(A(x))$ to build several Fuzzy relations that would help to understand the impact of the experiment factors on the study focus. Fuzzy relations are obtained by gathering empirical data (i.e. Table 30, and Table 31), and applying for the continues membership function $c(A(x))$ in Equation [2].

Generalization 1: Fuzzy Set (C_{SP}). The Fuzzy relation, C_{ps} , can be defined based on Equation [3]; C_{ps} relates both true defect detections (D), and the different levels of subjects' experience (S) (see Table 28).

Generalizing the relationship between subjects' experience and detected true defects can be obtained by defining P, S such that:

- P represents the number of true defects.
- S represents the classification of subjects' experience $S = \{BExp, AExp\}$.

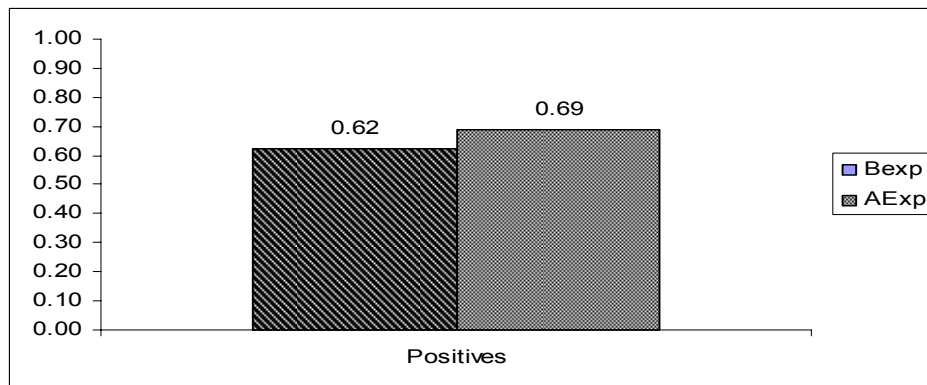


Figure 29: Membership grades of Fuzzy relation between true defects (P) and subjects (S)

C_{ps} is called the occurrence Fuzzy relation between the number of true defects and subjects' experience; C_{ps} is obtained as in the followings:

$$C_{PS} = \begin{matrix} & \xrightarrow{\text{Experience}} \\ \begin{matrix} \downarrow \text{Positives} \\ \left[\begin{array}{cc} BExp & AExp \\ R11 & R12 \end{array} \right] \end{matrix} & = & [0.62 \quad 0.69] \end{matrix} \quad \text{Equation [4]}$$

Note that, we have been using the collected data in both FPK and SPK with ignoring facts that: (i) experiments were conducted within three hours in the first package, while experiments were conducted within two hours in the second package; (ii) differences both in objects, and CBR's guidelines in both packages. Along this study however, in order to avoid such technical differences, manipulated data were taken in averages and number of inspection groups (subjects) was normalized to 9 inspection groups, in BExp, and 10 inspection groups in AExp.

Generalization 2: Fuzzy Set (C_{TS}). Presenting a Fuzzy relation between the subjects (S) and reading techniques (T) is necessary for understanding the impact of

reading techniques on the number of true defects (P). Such a relation can be obtained by generalizing data in FPK and SPK (see Equations [5] and related graphical representations in Figure 30).

$$C_{TS} = \begin{matrix} & \xrightarrow{\text{Inspectors}} \\ \begin{matrix} \downarrow \text{Techniques} \\ \text{CBR} \\ \text{FBR} \\ \text{SBR} \end{matrix} & \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{bmatrix} & = & \begin{bmatrix} 0.59 & 0.65 \\ 0.68 & 0.72 \\ 0.60 & 0.69 \end{bmatrix} \end{matrix} \quad \text{Equation [5]}$$

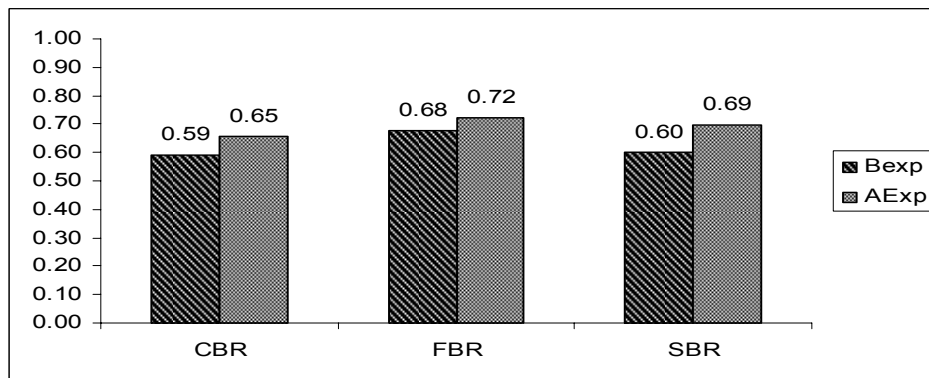


Figure 30: Membership grades of Fuzzy relation between subjects (S) and techniques (T)

We should also consider that CBR is the only technique that we can generalize; that is because the other reading techniques were not used in the FPK. In fact, conducting more families of empirical studies would help to understand the impact of reading techniques on the objective of the study.

Generalization 3: Fuzzy Set (C_{OS}). In order to present Fuzzy relation between the subjects (S) and the experiment objects (O), C_{OS} , we reason on the collected data from the two packages to obtain the following Fuzzy set shown by Equation [6], which results are graphically presented by Figure 31.

$$C_{OS} = \begin{matrix} & \xrightarrow{\text{Inspectors}} \\ \begin{matrix} \downarrow \text{Objects} \\ \text{Abstract} \\ \text{Concrete} \end{matrix} & \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} & = & \begin{bmatrix} 0.67 & 0.76 \\ 0.67 & 0.70 \end{bmatrix} \end{matrix} \quad \text{Equation [6]}$$

Note that, data presented are considering only the second package; this refers to the lack of homogenous and comparable data concerning the experiment objects in the two packages.

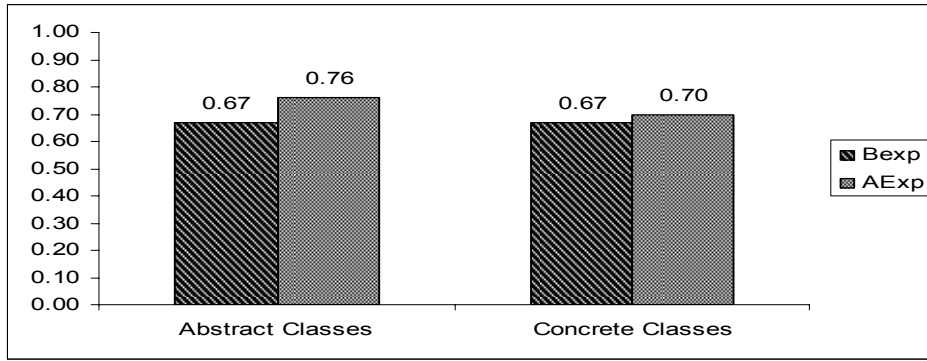


Figure 31: Membership grades of Fuzzy relation between subjects (S) and objects (O)

Generalization 4: Fuzzy Set (C_{DS}). Let us now consider the Fuzzy relation between defect types (D) and subjects' experience (S). According to Table 29, three types of defects are taken under consideration (A, B, and C); their corresponding Fuzzy relation with subjects' experience is as in the following:

$$C_{DS} = \begin{array}{c} \text{Defects} \\ \downarrow \\ \begin{array}{cc} A & \begin{array}{cc} \xrightarrow{\text{Inspectors}} & \\ C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{array} \end{array} \end{array} = \begin{bmatrix} 0.63 & 0.67 \\ 0.59 & 0.69 \\ 0.64 & 0.72 \end{bmatrix} \quad \text{Equation [7]}$$

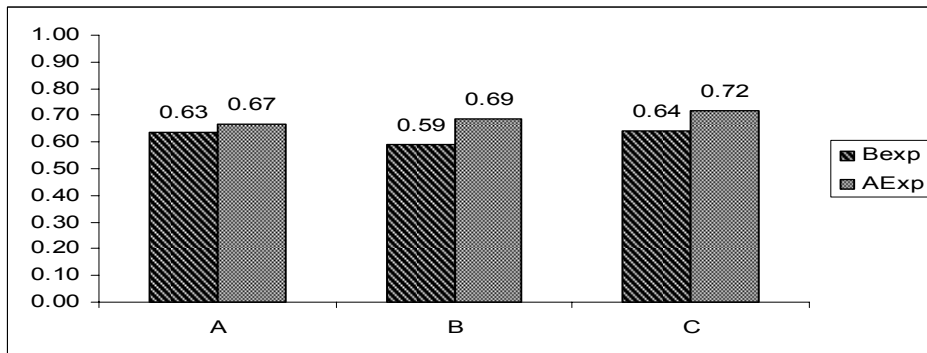


Figure 32: Membership grades of Fuzzy relation between subjects (S) and defects (D)

We finally note that, in order to build a solid body of knowledge in software engineering, a family of empirical studies is required. In particular for this study, we should say that Equations [1], [2], and [3] require more quality and reliability improvements by replicating the studies in both FPK, and SPK. However, evaluating the process of Cluster-impact Approach can be considered as one of the goals behind organizing this Chapter.

7.4. Manipulation of Fuzzy Sets

At this point of the study, we will be considering the previously established Fuzzy sets in table 33, Based on such data, we could investigate on a set of interactions between factors in a specific Cause/Effect layer (see impact lines 2, and 3 in Figure 28); this can be achieved by using two different views:

- **Impact on FoCus Interaction (ICI):** Presents the Impact of the experiment factors on the experiment focus (see impact line 2 in Figure 28); by this meaning, Table 34 shows the impact of Defect Types, C_{DP} , Objects, C_{OP} , and Techniques, C_{TP} , on the experiment goal when C_{SP} .

Table 34: the impact of Main Factor on the other factors

The Impact of A on B	Operation		Fuzzy description
	Cause	Effect	
A. Inspection techniques B. True defects Detected	Techniques (T)	True defects (P)	$C_{TS} \cdot C_{SP} \Rightarrow C_{TP}$
A. Objects per Class type B. True defects Detected	Objects (O)	True defects (P)	$C_{OS} \cdot C_{SP} \Rightarrow C_{OP}$
A. Types of defects B. True defects Detected	Defects (D)	True defects (P)	$C_{DS} \cdot C_{SP} \Rightarrow C_{DP}$

- **Impact on Factor Interaction (IFI):** presents the impact among the experiment factors; by this meaning, Table 35 shows the impact of Defect Types, Objects, and Techniques upon each other when the impact of subjects is known (see line 3 in Figure 28).

By the meaning of Fuzzy sets and Cluster-impact approach, IGI and IFI are two Fuzzy sets implementations in experimental software engineering, which help to extract issues related to the study objective by understanding the effect of change on the experiment factors when the impact of the Main Factor is known.

Table 35: the impact of Main Factor on the other factors

The Impact of A on B	Operation		Fuzzy description
	Cause	Effect	
A. Inspection Techniques B. Objects per Class type	Techniques (T)	Objects (O)	$C_{TS} \cdot C_{SO} \Rightarrow C_{TO}$
A. Types of defects B. Objects per Class type	Defects (D)	Objects (O)	$C_{DS} \cdot C_{SO} \Rightarrow C_{DO}$
A. Types of defects B. Inspection Techniques	Defects (D)	Techniques (T)	$C_{DS} \cdot C_{ST} \Rightarrow C_{DT}$

7.4.1. Impact on FoCus Interaction (ICI)

Generalization 5: Fuzzy Set (C_{TP}). Based on Table 34, our aim here is to establish a Fuzzy relation between reading techniques (T) and true defect detections (P) when the impact of the Main Factor (Subjects) is known, the interaction between techniques and subjects (C_{TS}) given by Equation [5], while the interaction between subjects and true defect detections, or positives, (C_{SP}) is previously established in Equation [4], Equation 7.5 shows, and Figure 33 graphically presents such a Fuzzy relation.

$$\begin{array}{c}
 \begin{array}{c} \text{Techniques} \\ \downarrow \end{array} \\
 \begin{array}{c} \text{CBR} \\ \text{FBR} \\ \text{SBR} \end{array} \\
 \begin{array}{c} \text{Positives} \\ \rightarrow \end{array} \\
 \left[\begin{array}{c} C_{11} \\ C_{21} \\ C_{21} \end{array} \right] \\
 = \\
 \begin{array}{c} \text{Techniques} \\ \downarrow \end{array} \\
 \begin{array}{c} \text{CBR} \\ \text{FBR} \\ \text{SBR} \end{array} \\
 \begin{array}{c} \text{Subjects} \\ \rightarrow \end{array} \\
 \left[\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{21} & C_{23} \end{array} \right] \circ \\
 \begin{array}{c} \text{Subjects} \\ \downarrow \end{array} \\
 \begin{array}{c} \text{BExp} \\ \text{AExp} \end{array} \\
 \begin{array}{c} \text{Positives} \\ \rightarrow \end{array} \\
 \left[\begin{array}{c} C_{11} \\ C_{21} \end{array} \right] \\
 \\
 C_{TP} = \begin{bmatrix} 0.65 \\ 0.69 \\ 0.69 \end{bmatrix} \qquad \text{Equation [8]}
 \end{array}$$

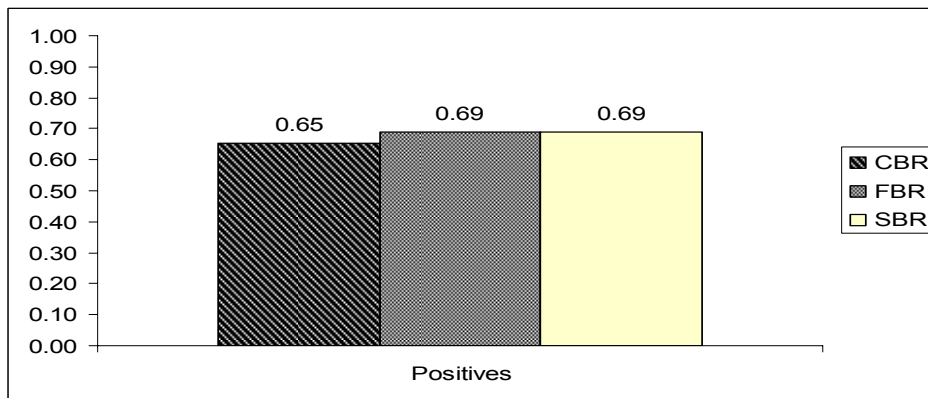


Figure 33: Fuzzy relation between the true defects (P) and reading techniques (T)

Generalization 6: Fuzzy Set (C_{OP}). This Fuzzy relation shows the impact of the experiment objects (O) on true defect detection (D) when the impact of the Main Factor (Subjects) is known; abstract and concrete classes are separated and previously evaluated (see Equation [6]), based on the known impact of the Fuzzy relation (C_{SP}), Fuzzy set (C_{OP}) can be established as in the followings (see also Table 34):

$$C_{OP} = \begin{array}{c} \text{Objects} \\ \downarrow \\ \begin{array}{c} \text{Abstract} \\ \text{Concrete} \end{array} \end{array} \begin{array}{c} \xrightarrow{\text{Positives}} \\ \left[\begin{array}{c} C_{11} \\ C_{21} \end{array} \right] \end{array} = \begin{array}{c} \text{Objects} \\ \downarrow \\ \begin{array}{c} \text{Abstract} \\ \text{Concrete} \end{array} \end{array} \begin{array}{c} \xrightarrow{\text{Subjects}} \\ \left[\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{23} \end{array} \right] \end{array} \circ \begin{array}{c} \text{Subjects} \\ \downarrow \\ \begin{array}{c} \text{BExp} \\ \text{AExp} \end{array} \end{array} \begin{array}{c} \xrightarrow{\text{Positives}} \\ \left[\begin{array}{c} C_{11} \\ C_{21} \end{array} \right] \end{array}$$

$$C_{OP} = \begin{bmatrix} 0.69 \\ 0.69 \end{bmatrix} \quad \text{Equation [9]}$$

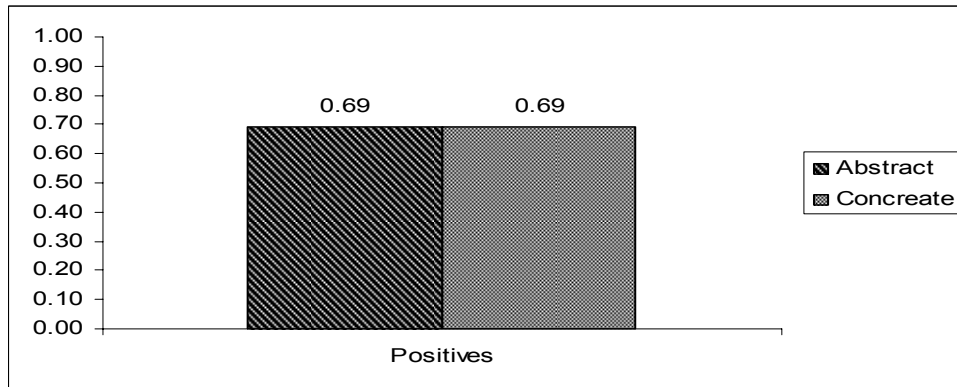


Figure 34: Fuzzy relation between the true defects (P) and Objects (O)

Generalization 7: Fuzzy Set (C_{DP}). According to Table 34, A Fuzzy set of the interaction between defect types (D) and true defects detected (P) is important to understand the impact of defect types on defect detection process; we intend to explore a Fuzzy set that describe the impact of defect types on true defect detection (C_{DP}) when the impact of the Main Factor (Subjects) is known (see Equation [4]). Fuzzy set C_{DP} can be established as in the following Equation [10] and Figure 35.

$$C_{DP} = \begin{array}{c} \text{Defects} \\ \downarrow \\ \begin{array}{c} A \\ B \\ C \end{array} \end{array} \begin{array}{c} \xrightarrow{\text{Positives}} \\ \left[\begin{array}{c} C_{11} \\ C_{21} \\ C_{31} \end{array} \right] \end{array} = \begin{array}{c} \text{Defects} \\ \downarrow \\ \begin{array}{c} A \\ B \\ C \end{array} \end{array} \begin{array}{c} \xrightarrow{\text{Subjects}} \\ \left[\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{array} \right] \end{array} \circ \begin{array}{c} \text{Subjects} \\ \downarrow \\ \begin{array}{c} \text{BExp} \\ \text{AExp} \end{array} \end{array} \begin{array}{c} \xrightarrow{\text{Positives}} \\ \left[\begin{array}{c} C_{11} \\ C_{21} \end{array} \right] \end{array}$$

$$C_{DP} = \begin{bmatrix} 0.67 \\ 0.69 \\ 0.69 \end{bmatrix} \quad \text{Equation [10]}$$

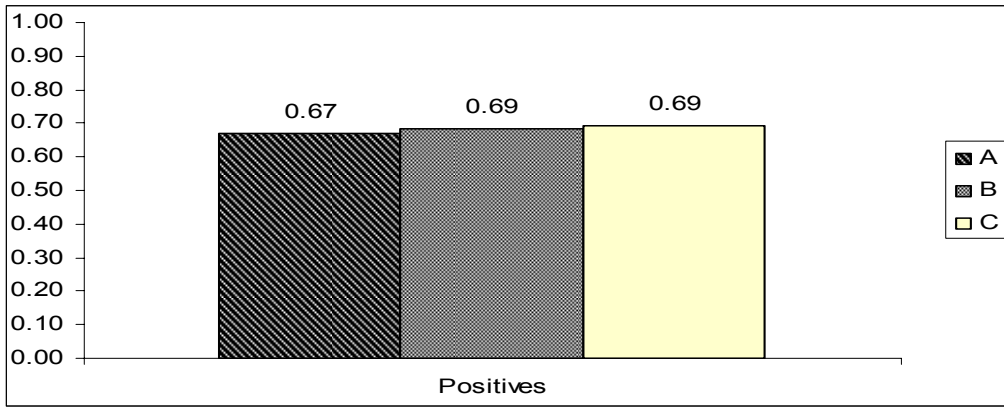


Figure 35: Fuzzy relation between the True defects and defect types

7.4.2. Impact on Factor Interaction (IFI)

Generalization 8: Fuzzy Set (C_{TO}). Let us now consider the relation between reading techniques (T) and the experiment objects (O) from the viewpoint of abstract and concrete classes; when subjects' experience is known, i.e. C_{SO} in Equation [6], the Fuzzy relation C_{TO} can be established as in the following Equation [11] shows and Figure 36 graphically presents.

$$\begin{aligned}
 C_{TO} = & \begin{array}{c} \text{Techniques} \\ \downarrow \\ \text{CBR} \\ \text{FBR} \\ \text{SBR} \end{array} \begin{array}{c} \xrightarrow{\text{Objects}} \\ \left[\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{21} & C_{23} \end{array} \right] \\ = \end{array} \begin{array}{c} \text{Techniques} \\ \downarrow \\ \text{CBR} \\ \text{FBR} \\ \text{SBR} \end{array} \begin{array}{c} \xrightarrow{\text{Subjects}} \\ \left[\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{21} & C_{23} \end{array} \right] \\ \circ \end{array} \begin{array}{c} \text{Subjects} \\ \downarrow \\ \text{BExp} \\ \text{AExp} \end{array} \begin{array}{c} \xrightarrow{\text{Objects}} \\ \left[\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{21} & C_{23} \end{array} \right]
 \end{array}
 \end{aligned}$$

$$C_{TO} = \begin{bmatrix} 0.65 & 0.65 \\ 0.72 & 0.70 \\ 0.69 & 0.69 \end{bmatrix} \quad \text{Equation [11]}$$

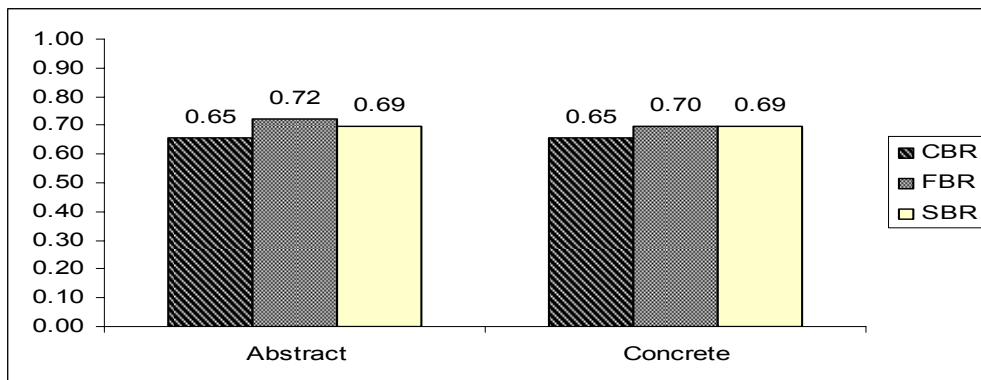


Figure 36: Fuzzy relation between the objects (O) and reading techniques (T)

Generalization 9: Fuzzy Set (C_{DO}). Based on Table 35, A Fuzzy set between the experiment objects (O) and defect types (D) when subjects' experience is known could help to understand the impact of defect types on OO software hierarchy; for instance, the impact of those types of defects on abstract and concrete classes can be established as in the following Equation [12] shows and Figure 37 graphically presents.

$$C_{DO} = \begin{matrix} & \xrightarrow{\text{Objects}} \\ \begin{matrix} \downarrow \text{Defects} \\ \text{A} \\ \text{B} \\ \text{C} \end{matrix} & \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{21} & C_{23} \end{bmatrix} \end{matrix} = \begin{matrix} & \xrightarrow{\text{Subjects}} \\ \begin{matrix} \downarrow \text{Defects} \\ \text{A} \\ \text{B} \\ \text{C} \end{matrix} & \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{21} & C_{23} \end{bmatrix} \end{matrix} \circ \begin{matrix} & \xrightarrow{\text{Objects}} \\ \begin{matrix} \downarrow \text{Subjects} \\ \text{BExp} \\ \text{AExp} \end{matrix} & \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{21} & C_{23} \end{bmatrix} \end{matrix}$$

$$C_{DO} = \begin{bmatrix} 0.67 & 0.67 \\ 0.69 & 0.69 \\ 0.72 & 0.70 \end{bmatrix} \quad \text{Equation [12]}$$

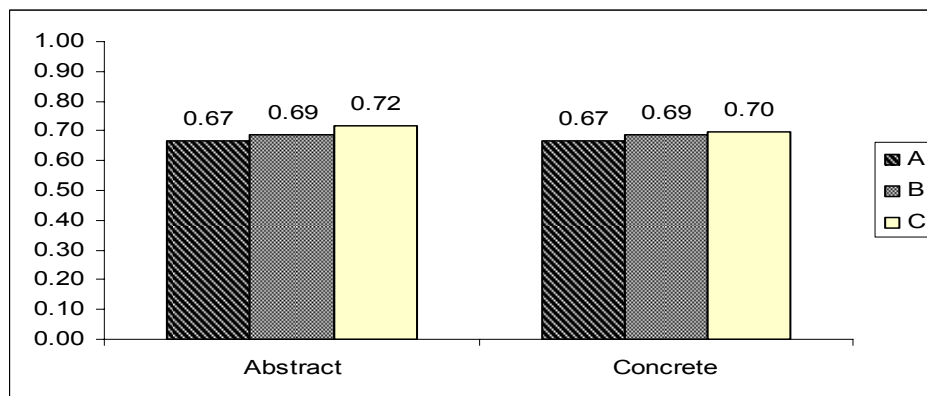


Figure 37: Fuzzy relation between the Defects and the Experiment Objects

Generalization 10: Fuzzy Set (C_{DT}). A Fuzzy set between the experiment objects (O) and reading techniques (T) when subjects' experience is known can be established as in the following Equation [13] shows and Figure 38 graphically presents.

$$C_{DT} = \begin{matrix} & \xrightarrow{\text{Techniques}} \\ \begin{matrix} \downarrow \text{Defects} \\ \text{A} \\ \text{B} \\ \text{C} \end{matrix} & \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \end{matrix} = \begin{matrix} & \xrightarrow{\text{Subjects}} \\ \begin{matrix} \downarrow \text{Defects} \\ \text{A} \\ \text{B} \\ \text{C} \end{matrix} & \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{bmatrix} \end{matrix} \circ \begin{matrix} & \xrightarrow{\text{Techniques}} \\ \begin{matrix} \downarrow \text{Subjects} \\ \text{BExp} \\ \text{AExp} \end{matrix} & \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \end{bmatrix} \end{matrix}$$

$$C_{DT} = \begin{bmatrix} 0.65 & 0.67 & 0.67 \\ 0.65 & 0.69 & 0.69 \\ 0.65 & 0.72 & 0.69 \end{bmatrix} \quad \text{Equation [13]}$$

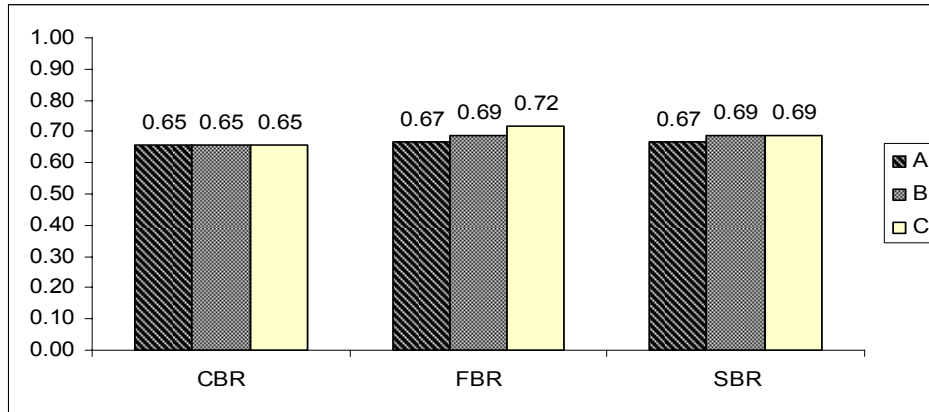


Figure 38: Fuzzy relation between the Defects and the Reading Techniques

7.5. Formalizing results

Finalizing the construction of building a body of knowledge from empirical studies requires formalizing resulted data (see Section 6.3.3). In order to reason on differences between Fuzzy membership values, elements of Fuzzy sets need to be classified in a qualitative manner.

As we stated in Chapter 6, elements of any Fuzzy set are defined, evaluated and eventually assigned to values that lies between [0, 1]. Such values describe the degree to which those elements belong to that Fuzzy set. Membership grades, therefore, can be logically defined qualitatively by translating those membership grades to several qualitative intervals, which in turn, can be treated as qualitative measurements, as shown by Table 25 in Chapter 6, membership grads are classified into 6 logical intervals. Figure 23 in Chapter 6 shows the degree of Fuzzy set elements to which we can assign the corresponding logic term. Recall that we use the term "More than" for distinguishing between any two memberships grad in the same interval.

In Section 7.3 above, the experiment data was based through a generalization progress and several Fuzzy sets were established, by the meaning of the categorization in Table 25; we note that, with respect to the available data , while some results expose the significance of cluster-impact Approach, by deeply providing knowledge concerning the effectiveness of reading techniques in detecting true defects, some other results are presenting common knowledge concerning software

verification in general (such as results 5,6, and 7). We formalize the resulted Fuzzy sets in a textual form, as in the followings:

Formal Result 1:

With basically and advanced expert subjects, true defects can be detected *often*; however, advanced expert subjects detect true defects *more than often*.

Formal Result 2:

a. With basically expert subjects, true defects are detected *seldom* when using CBR inspection technique, while they detect true defects *more than often* when using FBR technique, and *often* when using SBR techniques.

b. With advanced expert subjects, true defects are detected *more than often* for CBR and SBR inspection techniques, but *very often* with FBR inspection technique.

Formal Result 3:

a. With basically expert subjects, true defects, which are located in abstract and concrete classes, are detected *more than often*.

b. With advanced expert subjects, true defects are detected *more than very often* in abstract classes; but *more than often* in concrete classes.

Formal Result 4:

a. Defects of type (A: Initialization) are detected *often* by basically expert subjects, and *more than often* by advanced expert subjects.

b. Defects of type (B: Computation) are *seldom* detected by basically expert subjects, but *more than often* by advanced expert subjects.

c. Defects of type (C: control) are detected *often* by basically expert subjects, but *very often* with advanced expert subjects.

Formal Result 5:

In OO software, it is *more than often* that reading for defect detection supports software readers in detecting defects.

Formal Result 6:

In OO software frameworks, it is *often* that readers detect true defects in abstract as well as in concrete classes.

Formal Result 7:

In OO software, it is *more than often* that reading for defect detection tends to true defects of type initialization, control, and computation.

Formal Result 8:

- a. For what concerns CBR technique for defect detection in OO software frameworks, it is *more than often* that reading for software defects tends to detect true defects of type initialization, control, and computation in abstract as well as in concrete classes.
- b. For what concerns FBR technique for defect detection in OO software frameworks, it is *very often* that reading for software defects tends to detect true defects of type initialization, control, and computation in abstract as well as in concrete classes.
- c. For what concerns SBR technique for defect detection in OO software frameworks, it is *more than often* that reading for software defects tends to detect true defects of type initialization, control, and computation in abstract as well as in concrete classes.

Formalization 9:

- a. In object-oriented software, it is *more than often* that reading software framework tends to detect true defects of type initialization and control in abstract as well as in concrete classes.
- b. In object-oriented software, it is *very often* that reading software framework tends to detect true defects of type control in abstract as well as in concrete classes.

Formal Result 10:

- a. In object-oriented software, it is *more than often* that using CBR technique for defect detection tends to detect defects of type initialization, computation, and control.
- b. In OO software, it is *more than often* that using FBR technique for defect detection tends to detect defects of type initialization and computation; but it is very often that using FBR technique for defect detection tends to detect defects of type control.

- c. In object-oriented software, it is *more than often* that using SBR technique for defect detection tends to detect defects of type initialization, computation, and control.

Discussion of Part II

Along with Chapters 6 and 7, we defined and eventually implemented a mechanism for evaluating the use of Fuzzy sets in experimental software engineering:

- i) We defined an approach for better understanding the impact of cause variables on the effect variables. In the former, we included both the traditional factors, which we denoted as the “Input factors”, and the traditional parameters, blocking-variables etc., which we denoted as the “Design factors”. By the meaning of Cluster-impact approach, we partitioned those independent cause variables into several equivalent classes of factors (“Layers”), each part consisting of several domain-related independent variables.
- ii) In families of experiments sharing the goal, we adopted the interaction between the experiment factors by defining a Main Factor, as a factor able to represent its equivalence class, i.e. its layer; a Main Factor is a factor that interacts with other factors in the same class (“Strata”), and is the only factor that directly affects the Main factor in the upper contiguous layer, eventually the topmost layer, that is the study focus. All the independent variables that belong to a Main Factor must be identified in the planning phase of the experiment and fixed into a constant; this enables to easily measure the impact of Main Factors on the other factors as well as the study focus (the response and dependent variables).

Our implementation on Fuzzy sets is performed in two main directions, with respect to interactions among some different factors: Impact on FoCus Interaction (ICI), which considers the impact of the experiment factors on the study objective; Impact on Factor Interaction (IFI), which considers the interaction among the experiment factors themselves, etc.

Based on the analysis of Fuzzy sets that we generated, which enables to derive 10 Formal Results in our test case, we will be qualitatively discussing these formulated results (see Section 7.5).

Formal results 5, 6, and 7 are common results; they basically emphasize on the feasibility and importance of reading for software defect detection; this holds for OO software (Formal result 5), and for OO software frameworks (Formal result 6) with special emphasis.

It is clear that readers, while reading software artifacts, are indeed complicated with learning issues; the more a reader understands software code, the more s/he exposes defects; this in turn requires more time and more clear and detailed software specifications, which may explain the software different constructs. What we have presented so far, are some general views that strongly emphasize on several facts when inspecting software for defects; this implies that:

1- As long as we inspect object oriented software for defects we may find some, this applies also for object oriented software frameworks.

Based on point 1 above, there is no specific point where we could stop testing; however, it seems that the inspection process is affected by many variables that could optimize (Constructive Affect), sometimes reduce (Destructive Affect), the significance of software inspection effects, and therefore:

2- In experimental software engineering, understanding the impact of the experiment factors must be an essential goal, focusing on the study focus is not enough to understand effects.

Point 1 above requires more details when we consider software defects, the challenge at this point is to determine what type of software defects do we expect from testers to detect, Fuzzy sets methodology emphasizes on the fact (formalization 7) that:

3- With object oriented software, defects of type initialization, control, and computation can be detected so frequent by testers at any level of software knowledge and experience.

For what concerns defect detection, Fuzzy sets methodology emphasize on the fact that software reading for OO software tends to detect defects of type: Initialization, control, and computation (formalization 7); that is, those three types of defects may easily, so far, be exposed and detected in OO software.

4- In object oriented software, the number of detected defects is proportional to testers' experience.

To be more specific, let us take a closer look at the nature of relationship between testers and the level of experience, knowledge in software programming and design analysis; our study emphasize on the fact (formalization 4) that:

5- Detecting defects of type Initialization and control increases when increasing testers' experience; while defects of type Computation require the highest level of expertise.

When considering OO software mechanism, first thing we note is the mechanism of message passing between software's different constructs; an OO software is complicated with a large number of message passing between classes, methods and

variables; a full functional behavior (each functionality presents a single property in the software) requires, at least one, initialization, control and computing constructs, defects might be located in each of those three constructs; however, computing constructs are the most complicated with respect to those, that is because faults of computing are tailored with the implementation of the functionality which builds the behavior and realize a functionality (concrete classes).

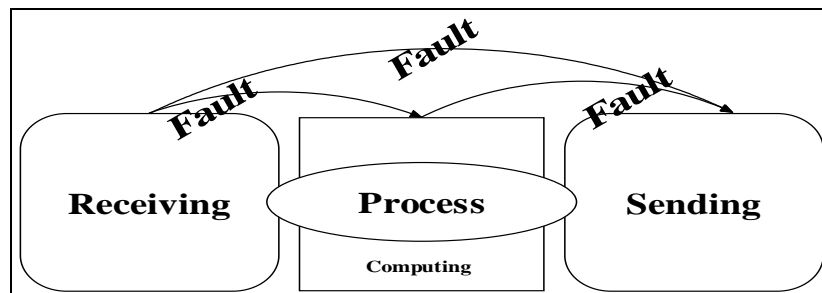


Figure 39: message passing mechanism and computation faults

Computing faults might be caused during the computing process itself, this is more easy to be detected and eventually isolated; however, computing faults might be also caused by receiving wrong messages, which in turn cause faults when sending wrong messages. We intend to emphasize on the fact that:

6- Detecting defects of type computation requires more inspecting effort with concrete classes as often as abstract classes, because computation process are often implemented in concrete classes.

Point 6 is a millstone for understanding several issues concerning the relationship between faults and OO mechanism, in particular, the impact of implemented and non implemented constructs in OO software, Fuzzy manipulation emphasize on the fact that:

7- Abstract and concrete classes present a strong impact on software inspection for what concerns computation faults.

Now, for what concerns reading techniques for defect detection, three reading techniques were used in the first package for the purpose of comparing their effectiveness in detecting defects, our manipulation with Fuzzy sets emphasize on the fact (formalization 2) that:

8- CBR technique requires more expert subjects than FBR and SBR techniques, while FBR technique works more effectively than SBR with any level of experience.

Again, point 7 emphasize on the importance of understanding software functionalities (see point 2); that is because software functionalities must be understood well before inspection takes place when considering computation faults.

The interaction between software reading techniques and OO constructs varies from one reading technique to another; however, for what concerns concrete classes, Fuzzy sets methodology (formalization 8) emphasize on the fact that:

9- The impact of abstract and concrete classes on the effectiveness of CBR technique is insignificant.

Point 8 tells that abstract and concrete classes have weak impact on the performance of CBR technique; this is simply because CBR technique doesn't pay enough attention to systematic inspection as often as FBR or SBR techniques, testers in CBR technique are given a list of what could possibly present defects, so the impact of implemented or non-implemented classes is trivial.

When reading software by using CBR technique, testers suspect on a large amount of code lines in the software; that is because, though that checklists documents cover most of possible test cases that could lead to a defect, CBR mechanism doesn't force the inspector to follow a certain mechanism of navigation through the code, instead, navigation through the code is made randomly and therefore, many of CBR testers were not able to gain a complete understanding of the software functionalities (formalization 10); that is:

10- CBR technique is in lack of a mechanism for navigating through object oriented software's different constructs, as well as in software code lines, including software frameworks' constructs.

In particular, for what concerns FBR technique, Fuzzy sets emphasize on the fact (formalization 2) that:

11- With respect to CBR and SBR techniques, FBR is more capable to guide testers through the code for true defects, either with basically or advanced expert testers.

And therefore,

12- With respect to CBR and SBR techniques, FBR technique is usually more effective in detecting true defects.

In general, in order to inspect any software for defects, we should take in count software design, architecture, functionalities, and programming method (procedural, modeling, OO); point 6 also emphasize on the fact that:

13- Inspecting any software for defects requires a systematic mechanism of code navigation; such mechanism must be tailored to software programming methods.

Functionality based reading technique is a mechanism of navigation through the code as well as an approach for understanding software's different functionalities. Moreover, Functionality based reading technique is practical for human nature; especially when most of the technologies and theories in software engineering are human-based [9].

14- During software inspection for defects, a first step is to understand the software's different constructs by the meaning of their functionalities (what software functionalities actually do?); a further step is to decide where to start inspecting (how to inspect what software functionalities do?).

In OO software, and by the meaning of software functionalities, inspection starting point is usually, if not always, an abstract class where most of definitions and initializations are located,

15- The strategy of navigation through functionality-tracing is an effective mechanism for defect detection.

Point 15 above is an axiom of the fact [9] that:

16- Experimentation on software engineering is complicated with several learning issues in a way that create difficulties when predicting results.

Chapter 8

Conclusions

In this study, an attempt has been presented to investigate on the ability to better understanding software quality principles, focusing on different factors that affect software during software development; in particular, the study focus on those factors that impact on the effectiveness and efficacy of software reading techniques for defect detection, aiming to improve the quality of an existing software (i.e. software is in the implementation phase). However, because similar impacts of factors can occur in all phases of software development life cycle, this study, in general, contributes to efforts that aim to understand some of the experimentation principles and characterizations in software engineering ([65], [48], and [55]).

The present study is based on empirical data, as collected from four controlled experiments; those empirical studies have had a quite acceptable level of reliability ([1], [2], and [16]); by manipulating results from those controlled experiments, we have proposed and eventually evaluated a new technique, so called Cluster-impact Approach for organizing the experiment factors based on the study focus. By the meaning of the study focus, several layers of factors are established, while each layer consists of one Main Factor. The more external is the layer, the more it impacts on the experiment focus. In particular, for this study, we considered the known impact of the level of subjects' experience on a certain number of defect categories. The study is, therefore, a complex of two proportional activities:

- I. Software benchmark: in this activity, bases of knowledge are established through observation; several empirical investigations are conducted with an acceptable level of quality and reliability; a software base of knowledge that can be achieved through observation can be considered as a study benchmark.
- II. Integration Approach: An approach that is able to map empirical studies in software engineering to the required amount of software knowledge, and eventually software understanding.

With respect to the study goal, and by formally enacting those activities, we finally figured that these are simply presenting some, or the minimal degree of, requirements that are necessarily needed when empirically investigating the different units and individual areas in software engineering.

For what concerns software quality, several standardizations for defining software quality exist and defined by several organizations, researching on the possible approaches for understanding “when” and “how” to achieve those standardizations is one of the important efforts in software engineering [65],[9]; the challenge here is that software engineering individual areas are complicated by a large amount of factors that seriously impact on efforts for achieving those standardizations of software quality; these factors in fact, cause many impact-causes, where it is difficult for a researcher to control [9] in full.

In particular, human factor plays an important role in software engineering [55]; most of software development phases are “handmade”, and so, investigating on software requires a particular consideration of the impact of such factor. Because not all problems are placed logically, humans have the ability to understand the problem and eventually define several scenarios that characterize solutions. Empirical investigations are involved to verify those solutions and evaluate their efficiency and effectiveness ([45], [9]).

1. Empiricism is difficult in software engineering, because of human factor contribution to empirical studies.

Moreover, in empirical studies, providing significant/insignificant statistical evidences are insufficient without considering how to measure the observed differences between the experiment factors and their possible changes at experiment conduction time. Moreover, software benchmarks are different from manufacturing benchmarks; in the latter we build the product over and over to meet a particular set of specifications, while software is developed and each product is different from the last [9]. This emphasize on the fact that most of the technologies and theories in software engineering are human-based, and so variation in human ability tends to obscure experimental effects. Human factors tend to increase the costs of experimentation while making it more difficult to achieve statistical significance ([9], [48]). Present study emphasize on such fact. We also conclude that:

2. In software engineering research, statistical power is too strict when confirming or rejecting hypotheses; some requirements for enacting statistical hypotheses testing are difficult to be achieved in software engineering.

That is because statistical methods for hypotheses testing provide basic explanations concerning the impact of the independent input/design factors on the

dependent response, while experiments in software engineering are, or should be, conducted for understanding factors (i.e. the experiment input variables) that impact on an effect (i.e. the experiment response variables), indeed :

3. *In software engineering, statistical power-based conclusions are not final; measuring the impact (i.e. the degree of a factor's contribution to a specific response) of the experiment independent variables (i.e. input / design factors) on the effect (i.e. the effectiveness, the efficiency, etc.) is necessary for finalizing any study.*

In fact, one of the results of this study demonstrated one strategy that maps the understanding of software with the input-design factors – response variables in experimental software engineering; that is:

4. *In order to gain understanding in software engineering, we need to provide a base of knowledge concerning software by empirically focusing on all possible factors that might contribute in building such base of knowledge.*

Problems that we considered in this study as related to software development, like verification (defects), human contribution (subjects), development methods (processes, plus languages and their objects), are a few of the factors that impact on software quality; and therefore:

5. *In order to achieve an acceptable level of software understanding, sets of factor-based software knowledge are needed.*

As a further result of the present study:

6. *Focusing on the impact of factors that contribute to software development must be a basic goal when researching in software engineering.*

The goal of Cluster-impact Approach, which we proposed in the present study, is to map software engineering empirical data to software understanding needs. One of the essential needs for the Experimental Software Engineering (ESE) community is to reach agreement on, and eventually establish, an organized working framework that sets out the families of empirical studies that grows in an ordered and organized manner, ruling out the now uncontrolled growth [48]. By defining the experiment factors as input/design factors, Cluster-impact Approach contribute to efforts concerned with the need to establish such an approach to standardize SE empirical works, that is:

7. *Organizing the experiment variables into several, input/design factors, impact-based order of clusters for each set of factors, and layers for each cluster with one Main Factor, may contribute to define an organized schema for empirical studies in software engineering.*

Chapter 9

APPENDICES

Appendix A1: Fuzzy sets

Theorem [43]: Let A be a Fuzzy set on X , x an element of Universal set U of the phenomenon X ($x \in U$), and $A(x)$ a Grade function, i.e. it is interpreted as the degree to which x belongs to A . Moreover, Let cA denote the complement of A to the Universal set U , $cA(x)$ be a Grade function of cA . Then, $cA(x)$ may be interpreted not only as the degree to which x belongs to cA , but also as the degree to which x does not belong to A . Similarly, $A(x)$ may also be interpreted as the degree to which x does not belong to cA .

Function c can be defined:

$$c : [0,1] \rightarrow [0,1]$$

which assigns a value $c(A(x))$ to each membership grade $A(x)$ of any given Fuzzy set A . $c(A(x))$ is a function of grade function ($A(x)$); any value $c(A(x))$ is interpreted as the value of $cA(x)$. That is

$$c (A (x)) = cA (x)$$

For all $x \in$ to X . and therefore, Fuzzy sets must satisfy the following two conditions:

- (i) $c(0) = 1$ and $c(1) = 0$
- (ii) For all $a, b \in [0, 1]$, if $a \leq b$, then $c(a) \leq c(b)$

This means that the membership function of a Fuzzy set A must lie between $[0, 1]$; and that the membership grid of the elements of the Fuzzy set A must be increasing when the elements themselves increase, in other words, membership function is continues, and its boundaries lies between 0 and 1.

Axiom: the following axioms can be established:

(i) $c(0)$ is continuous function

(ii) $c(c(a)) = a$ for each $a \in [0, 1]$

Now, if those two conditions are satisfied, then we can say that the membership function is an equilibrium function with respect to $A(x)$, that is.

Finally, $\forall x \in U(X), A(x) = 1 - cA(x)$

APPENDIX A2: Software Verification

Verification denotes the process of evaluating or proving an attribute or capability of a program or system and determining that it meets its required results [32]. Verification includes any activity that aims to evaluate a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [37]. Verification therefore aims to answer the question: Have we built the system right? It is different from Validation, which aims to answer the question: Are we building or have we built the right system?

The history of software verification is as long as the history of software development itself. Verification is an integral part of the software life cycle; effectiveness of methods and techniques for software verification need to be classified with respect to the type of product, environment, and language used. Verification techniques are essential for improving the quality of software products during software development life cycle. Improving software quality would present troubles to software designers, analysts, software programmers, and eventually customers, in the absence of feasible and cost-effective methods and techniques for verifying software artifacts.

Software verification is a trade-off between budget, time and quality. Different verification methods behave differently with different products and in different development contexts. There are theoretic verification methods, which goal is to prove the correctness of software artifacts, and also aim to apply formal transformations to upper level software specifications in order to get lower level artifacts. This study does not consider proving methods. Other approaches that can be used to verify software are inspection and testing; these will be considered in deep by this study; in fact, we assume software verification as the complex of knowledge, experiences and practices, which are enacted for checking software artifacts against their requirement specifications, user needs, internal standards if any, and detecting and identifying defects. Once excluded approaches to software verification that are based theorem proving, we have to note that remaining software engineering theories are still not enough to provide practitioners with mechanistic models capable of evaluating of methods – languages and processes – techniques and tool independently from the when, where, on what, and who is using those methods, techniques and tools. Consequently, some empirical evaluation is

necessary to understand which mix of verification techniques to use, for what software, by whom, in which organization.

When empirically evaluating software verification techniques, there are always a large number of context variables. So, creating a cohesive understanding of experimental results requires a mechanism for motivating studies and integrating results. Anyway, there are quite large number of causes that must be understood in software engineering experimentations [9], which makes those experimentations quite difficult; in fact, most of the empirical studies pawn themselves to statistical analysis results, and an empirical study is doomed to fail if the experiment results do not confirm the study hypotheses with enough statistical significance.

Software inspection

Software inspection was formally defined by Michel E. Fagan [30] and introduced at IBM in the middle of 70s. The technique is based on structured teams of people, who meet for identifying and removing defects as early and effectively as possible from software analysis and design artifacts.

While the concept of software inspection changed in time by subsuming points of view and perspectives [56], and including software artifacts of any kind, included code, it is still characterized by requiring that humans enact some ruled accesses to software artifacts in the aim of detecting defects. Those humans can work in team or individually, and those teams can be structured or not.

Code inspection for defect detection is a kind of inspection. Because it is enacted on code and its goal is to find software defects, in particular software faults, it is also considered a static testing technique.

Software testing

Software testing denotes any activity that aim to evaluate an attribute or capability of a program or system and determining that it meets its required results [32]. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well; correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality.

Testing software code is an act, quite final in any development process iteration, aimed to ensuring that developed software does everything it is supposed to do.

Some testing efforts extend the focus to ensure the code does nothing more than it is supposed to do; consequently, this cannot be applied in full to framework-based software since frameworks are supposed to include all possible functionalities that an application domain might have. In general however, testing makes a significant contribution to guarding users against software faults and failures that can result in a loss of time, property or customers.

Goals behind software testing are [48, 36]:

- To help to expose those hidden number of defects which are created during the definition of software specification, design and coding stages of the development.
- To provide a confidence that failures do not occur.
- To reduce the cost of software faults and failures over the life of a product.

Figure 40 shows a traditional approach to software analysis and construction, based on customer requirements. User needs are brought by the marketing group; the project feasibility is studied by the engineering group; this works together the software quality assurance group during the product analysis and design, and generates software specification; developers start realizing design and eventually implementing the product, while the software quality assurance group are working for the software testing plans, which will be applied when the developers are ready with the code.

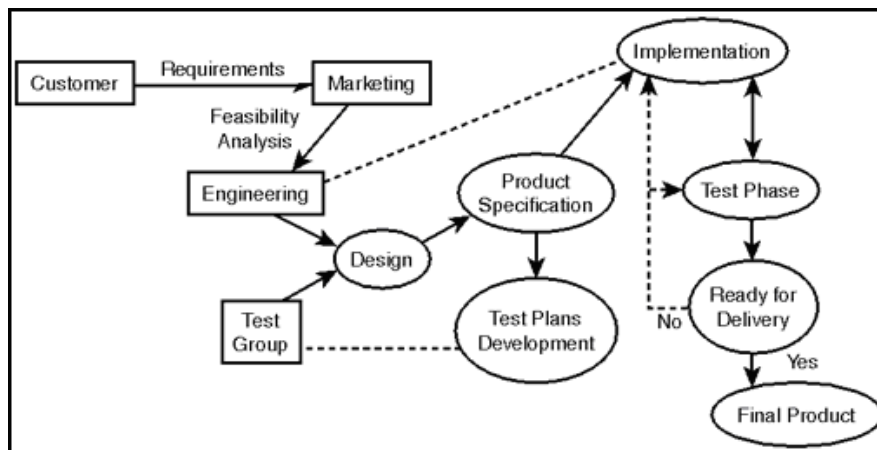


Figure 40: From customer need to software products in the traditional approach to software development

Based on such a traditional approach, testing comes at the end, so no amount of testing will be enough to improve the overall quality of the software, as determined by analysis and design; in fact, testing helps only in identifying failures and faults; so, when development faults are detected, the product returns to developers who can fix

and eventually remove faults, while the product should return to analysts and designers for working on analysis and design faults, respectively. In conclusion, the more testing we do of a system, the more convinced we might be of its correctness; yet testing cannot in general prove a system works 100% correctly [38].

There are further software process models and supporting tools, which aim to remove limits of the previous mode, e.g. the IBM-Rational Unified Process [61], which is an iterative-incremental process; a few of user needs are classified for risk and priority, and few of them are assumed in the first iteration of development. Further user needs will be included for the next and next iteration. As shown in Figure 40, an iteration might proceed again; however, because a limited set of additional features are developed in an iteration, inspection of artifacts and eventually testing will help to discover analysis, design, and implementation defects earlier than in the traditional approach. Finally, few critical paths can be selected very early in a software application and prototyped by using interpretable languages. In such a case, testing analysis and design ideas can be made quite early in the development process, even before code is written. [37]

Software testing strategies

There are some approaches for a software to be tested: static and dynamic testing, where the former works on software documents, and the latter is based on software execution; white box testing, which can be enacted statically or dynamically, and black box testing, which is a dynamic technique. Each of those approaches has its own characteristics, advantages and disadvantages.

Black box testing

Black-box is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure ([53],[11]); it is also termed data-driven, input/output driven or requirements-based testing [34]. In black box testing, the more we have covered in the input space, or the more the test cases are significant sample of the input space, the more problems we find and the more good quality we gain; only the interface behavior of the software module is concerned. A question in black box testing concerns how to maximize the effectiveness of testing with minimum cost; it is generally impossible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. Based

on reported experience [37] [55], characteristics of black box testing can be collected and summarized as in the following:

- Although only a small number of possible inputs can actually be tested, black box testing is more effective on larger units of code than in white box testing; in addition, black box testing may leave many program paths untested.
- The tester needs no knowledge of implementation, including specific programming languages.
- Tester and programmer are independent of each other; however, without synchronizing between the programmer and the tester, there might be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried.
- Black box tests are done from the point of views of users.
- Black box testing helps to expose any ambiguities or inconsistencies in the specifications; however, without clear and concise specifications, test cases are hard to design.
- Test cases can be designed as soon as the specifications are complete.

White box testing

In white-box testing, the structure and flow of the software under test are visible to the tester; testing plans are made according to the details of the software implementation such as programming language, logic, and styles; white-box testing is also called glass-box testing, logic-driven testing, or design-based testing [34], [51]. White box testing is interested to examine the coverage of program control flow; this type of software testing requires the intimate knowledge of the program internals. Test cases can be similar to the functional ones, or randomly realized, or eventually derived from the control flow of the program. With respect to their focus, white box testing approaches can be classified as in the followings:

- i. Static Testing: does not necessitate the execution of the software; formal-based techniques and tools can be used to evaluate the coverage that should correspond to an input state. Also code-reading is a kind of white box testing [4].
- ii. Dynamic Testing: is what is generally considered as ``testing``, i.e. it involves running the system.

For each of those classifications, these types of coverage can be applied:

- a. Code Coverage: the approach is performed in the aim of ensuring that every statement is covered at least once.
- b. Branch Coverage: the approach is performed in the aim of ensuring that all branches are covered at least once. Branch Coverage includes Code Coverage.
- c. Path Coverage: the approach is performed in the aim of ensuring that all control-paths are covered at least once. Path Coverage includes Branch Coverage.
- d. Bound Coverage: the approach is performed in the aim of ensuring that all loops are covered at least once in their lower bound number of iterations, and upper bound number of iterations, respectively. Bound Coverage includes Path Coverage.
- e. All-definition-use-path Coverage: the approach is performed in the aim of ensuring that all data-paths between the definition of a variable and the use of that definition are covered.

Those Instruction testing approach, control-flow testing approach, loop testing approach, and data-flow testing approach, all maps the corresponding flow structure of the software into a directed graph. While functional or random test cases can be utilized, eventually all those approaches can tend to ask for carefully selecting test cases, based on the criterion that all the nodes or paths have to be covered or traversed at least once. Because such a selection process is not decidable, often also not feasible, by doing so the analyst may discover that some code had never been covered at all during testing, and eventually proving that some of that code is of no use, which can not be discovered by black box testing.

In general, the characteristics of white box testing can be collected and summarized as in the followings:

- White box testing forces test developer to reason carefully about the implementation; this also forces the tester to trace software code toward understanding the implementation, at least its control flow, before start testing.
- White box testing limits software partitioning as often as functional testing; that is because the tester needs to trace and eventually understand the whole functionality of the software.
- White box testing reveals defects in "hidden" code.

However, concerning static white box testing, it is extremely expensive and still missing some test cases; dynamic white box testing, in turn, calls for including probes into the code and outputting their states during the test phase: as a

consequence, we will not be testing the target software, less than leaving probes in the released software; anyway, we would not be again able to trace coverage when outputs of probe states are disabled.

Chapter 10

References

[1] Abdelnabi Z., Cantone G. "Inspecting Software Frameworks for Defect Detection Effectiveness: A Replicated Experiment", 17th International Conference in Software & Systems Engineering and their Applications, ICSSEA04, December 2004, Paris, France.

[2] Abdelnabi Z., Cantone G., M. Ciolkowski and H.D. Rombach, "Comparing Code Reading Techniques Applied to Object-oriented Software Frameworks with regard to Effectiveness and Defect Detection Rate", Proceedings of ISESE04.

[3] Basili V. R., "Quantitative evaluation of software methodology". Proceedings of the First Pan Pacific Computer Conference. Melbourne, Sept. 1985.

[4] Basili V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies", IEEE Transactions on Software Engineering, CS Press, 10 (12), pp 1278-1296, December 1987.

[5] Basili V. R., G. Caldiera and H.D. Rombach, "Goal Question Metric Paradigm," in J. J. Marciniak (ed.), Encyclopedia of Software Engineering 1, New York: John Wiley & Sons, pp. 528-532, 1994.

[6] Basili V. R., G. Caldiera G., and G. Cantone, "A Reference Architecture for the Component Factory", ACM TOSEM, Vol. 1, No. 1, January 1992.

[7] Basili V. R., G. Caldiera, and H. D. Rombach, "Experience Factory", Encyclopedia of Software Engineering, ed. J. J. Marciniak, Vol. I, pp. 469-476, Wiley, 1994.

[8] Basili V. R., Weiss D. M., "A Methodology for Collecting Valid Software Engineering Data", IEEE-TSE, Nov. 1984, pp. 728-738.

- [9] Basili, V., F. Shull, and F. Lanubile, "Building Knowledge through Families of Experiments", IEEE Transactions on Software Engineering, Vol. 25, No. 4, pp. 456-473, July 1999
- [10] Basili, V., G. Caldiera, F. Lanubile, and F. Shull, "Studies on reading techniques", Proc. of the Twenty-First Annual Software Engineering Workshop, SEL-96-002, Greenbelt, MD, December 1996.
- [11] Beizer B., "Black-box Testing: techniques for functional testing of software and systems", New York: Wiley, c1995. ISBN: 0471120944 Physical description: xxv, 294 p.: ill. ; 23 cm.
- [12] Briand L. C., Wust J. "Modeling Development Effort in Object-Oriented Systems Using Design Properties", IEEE Transactions on Software Engineering, Vol. 27, N° 11, 963-986.
- [13] Brooks A., et others, "Replication's Role in Experimental Computer Science", EfoCS-5-94 (RR/94/171), 1994.
- [14] Budd T. A., "Introduction to Object Oriented Programming with Java", Wiley, 2001.
- [15] Cantone G., and S. Celiberti: "Evaluating efficiency, effectiveness, and other indices of code reading and functional testing for concurrent event-driven OO Java software: Results from a multi-replicated experiment with students of different level of experience", Proceedings of 2002 IEEE ISESE Symposium, Vol. II, Nara, JP, 3-4 October 2002, Pgs. 23-24.
- [16] Cantone G., and Z. A. Abdulnabi, "Effectiveness and Fault Detection Rate of Code Reading and Functional Testing with Event-driven OO Java Software: Results from a Multi-replicated Experiment with Students of Different Level of Experience", URM2-DISP-ESEG 01.03 T.R., Rome, April 2003.
- [17] Cantone G., L. Cantone, and P. Donzelli, "Models, Measures and Learning Organizations for Software Technologies" in Global Semiconductor Technology 2001, published by World Markets Research Centre (WMRC) in association with SIA, SISA, Nepcon WEST, FSA, pp. 136-148, January 2001.

- [18] Cantone G., L. Colasanti, Z. Abdulnabi, A. Lomartire, and G. Calavaro, "Evaluating Checklist-Based and Use Case-Driven Reading Techniques as Applied to Software Analysis and Design UML Artifacts", in [29] pp. 142-165.
- [19] Cantone G., Z. Abdulnabi, A. Lomartire, and G. Calavaro, "Effectiveness of Code Reading and Functional Testing with Event-Driven Object-Oriented Software", in [29] pp. 166-192.
- [20] Cantone, G., "Measure-driven Processes and Architecture for the Empirical Evaluation of Software Technology", *Journal of Software Maintenance: Research & Practice* 12(1): pp. 47-78, 2000.
- [21] Carver J., "The Impact of Background and Experience on Software Inspections", PhD Thesis, 2003, University of Maryland, USA.
- [22] Carver J., L. Jaccheri, S. Morasca, and F. Shull, "Using Empirical Studies during Software Courses", in [29] pp. 81-103.
- [23] carver J., VanVoorhis J., Basili V. R., "Understanding the Impact of Assumptions on Experimental Validity", In *Proceedings of International Symposium on Empirical Software Engineering*, October 2004.
- [24] Chernak, Y., "A Statistical Approach to the Inspection Checklist Formal Synthesis and Improvement, *IEEE Transactions on Software Engineering*", pp. 866-874, 1996.
- [25] Chroust G., "Systems Engineering and Automation: SEA", www.sea.unilinz.ac.at/research/projects/cosimis/html/inspection/rt.htm (02.01.2004)
- [26] Dunsmore A. M. Roper, and M. Wood, "Practical Code Inspection of Object-Oriented Systems", *Proceedings of WISE'01, Paris, 2001*. www.cas.mcmaster.ca/wise/
- [27] Dunsmore A., "An Empirical Investigation of Three Reading Techniques for Object-Oriented Code Inspection". EFOCS-44-2001, DCS, University of Strathclyde, 2001.

- [28] E. Gamma R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design", Proceedings of the 7th European Conference on Object- Oriented Programming, Kaiserslautern, Germany, 1993
- [29] ESERNET EC Project, Empirical Methods and Studies in Software Engineering: Experiences from ESERNET, Eds. R. Conradi and A. I. Wang, LNCS2765, Springer, 2003.
- [30] Fagan M. E., "Design and code inspections to reduce errors in program development," IBM Systems Journal, vol. 15, no. 1, pp. 182-211, 1976. [2] Tom Gilb and Dorothy Graham, Software inspection, Addison Wesley, 1993.
- [31] Fielding A.," Research Methods", Manchester Metropolitan University, [www.149.160.199.144/ resdesgn /1stframe.html](http://www.149.160.199.144/resdesgn/1stframe.html), 23.12.2003
- [32] Genero M., M. Piattini, E. Manso, and G. Cantone, "Building UML Class Diagram Maintainability Prediction Models Based on Early Metrics", Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS'03), 2003 (to appear.)
- [33] Hetzel, William C., "The Complete Guide to Software Testing", 2nd ed. Publication info: Wellesley, Mass. QED Information Sciences, 1988. ISBN: 0894352423.
- [34] http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/reference#reference
- [35] IBM Corp., "S/390 Orthogonal Defect Classification Education". www-1.ibm.com/servers/eserver/zseries/ odc/ nonshock/odc8ns.html, 4.04.2003
- [36] Intel Networking & Communications, [www.resource .intel.com/telecom/ support/solutions/c_framework.htm](http://www.resource .intel.com/telecom/support/solutions/c_framework.htm), 4.04.2003
- [37] John D. McGregor, David A. Sykes,"Introduction to Testing Object-Oriented Software", Addison-Wesley professional web site, <http://www.awprofessional.com/articles/article.asp?p=167907&seqNum=3,1-10-2004>.
- [38] John D. McGregor, David A. Sykes," A Practical Guide to Testing Object-Oriented Software"
- [39] Johnson R. E., "Components, Frameworks, Patterns", SSR 1997: 10-17

[40] Juristo N., and A. Moreno, "Basics of Software Engineering Experimentation", Kluwer AP, February 2001.

[41] Juristo N., and S. Vegas, "Functional Testing, Structural Testing and Code Reading: What Fault-Type Do They Each Detect", in [28] pp. 289-232.

[42] Kamsties E., and C. M. Lott, "An Empirical Evaluation of Three Defect-Detection Techniques", University of Kaiserslautern, ISERN-95-02 T.R., Germany, 1995

[43] Kiler G. j., Folger T. A. "Fuzzy Sets, Uncertainty, and Information" 1988 ISBN: 0-13-345984-5

[44] Laitenberger O., DeBaud, J-M, "An Encompassing Life-Cycle Centric Survey of Software Inspection", ISERN-98-32.

[45] Lott C., H. D. Rombach, "Repeatable Software Engineering Experiments for Comparing Defect Detection Techniques", Empirical Software Engineering Journal, 1(3): 241-278, 1996

[46] Mendes E.I, Watson I., Mosley N. and Counsell S., "A Comparison of Development Effort Estimation Techniques for Web Hypermedia Applications" Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS'02), 2002.

[47] Myers, Glenford J., "The art of software testing", Publication info: New York : Wiley, c1979. ISBN: 0471043281 Physical description: xi, 177 p. : ill. ; 24 cm.

[48] N. Juristo, A.M. Moreno, S. Vegas, "Towards Building a Solid Empirical Body Of Knowledge In Testing Techniques", Proceedings of ISESE 2004.

[49] N. Juristo, and S. Vegas: "Functional Testing, Structural Testing and Code Reading: What Fault Type do they Each Detect?", Universidad Politécnica de Madrid, ESERNET book v3.9, 2003 (see Chapter 12 of this book).

[50] Neighbors J. M., "Draco: A method for engineering reusable software systems", in Software Reusability, Vol. 1: Concepts and Models, pp. 295-319, ACM Press, New York, 1989.

[51] Pan J. , "Software Testing" ,http://www.ece.cmu.edu/~koopman/des_s99/sw

_testing, Carnegie Mellon University, Pittsburgh, PA

[52] Pasetti A., "Software Frameworks and Embedded Control Systems", Springer-Verlag, Berlin, Heidelberg, (2001).

[53] Perry W. Boston E., "A standard for testing application software", Auerbach Publishers, 1991

[54] Pfleeger S. L. "Software engineering: Theory and Practice", 2nd edition, Prentice Hall, 2002.

[55] Risley C. "Glass Box Testing", http://www.cse.fau.edu/~maria/COURSES/_CEN4010-SE/C13/glass.htm, 03.04.2005

[56] Russo M. P., Z. Abdelnabi, M. Ciolkowski, and A. Lomartire, "Comparing Code Reading Techniques for C++ OO Software Frameworks", Proceedings of ISESE 2003 Poster & Demo Session, Rome, September 2003.

[57] Schmidt, D. C., and Fayad, M. E., "Lessons Learned Building Reusable OO Frameworks for Distributed Software," Communications of the ACM, vol. 40, no. 10, October 1997, pp. 85-87.

[58] Shull F., Basili V. R., Carver J., Maldonado J. C., Travassos G. H., Mendonca M., and Fabbri S., "Replicating Software Engineering Experiments: Addressing the Tacit Knowledge Problem", In Proceedings of International Symposium on Empirical Software Engineering, October 2002, pp. 7-16

[59] Shull F., Lanubile, L., and Basili V., "Investigating Reading Techniques for Object-Oriented Framework Learning", Technical Report CS-TR-3896, UMCP Dept. of Computer Science, UMIACS-TR-98-26, UMCP Institute for Advanced Computer Studies, ISERN-98-16, International Software Engineering Research Network, May 1998.

[60] Solingen (van) R., and E. Berghout, "The Goal/Question/Metric Method: A Practical Guide for Quality Improvement and Software Development", McGraw-Hill Intl. Editions, 1988.

[61] The IBM rational software web site, <http://www-306.ibm.com/software/rational/>

- [62] Thomsett .R, Hall P., "Radical Project Management", 2002 ISBN:0-13-009486-2
- [63] University of Kaiserslautern. www.wagse.informatik.uni-kl.de/teaching/se1/ws2003/ (29.04.2004)
- [64] University of Roma Tor Vergata. [ese.uniroma2.it/ esperimentoFramework / English%20Version/ default.htm](http://ese.uniroma2.it/esperimentoFramework/English%20Version/default.htm) (29.03.2005)
- [65] Vegas S, Juristo N., Basili V. R., "Identifying Relevant Information for Testing Technique Selection: An Instantiated Characterization Schema", Springer 2003, ISBN: 1-4020-7435-2
- [66] Von Mayrhauser A., and A. M. Vans, "Industrial Experience with an Integrated Code Comprehension Model", Software Engineering Journal, September 1994.
- [67] Wohlin C. et others: "Experimentation in Software Engineering. An Introduction", Kluwer A. P., 2000.
- [68] www.cs.umd.edu/projects/SoftEng/ESEG