Are Unreachable Methods Harmful? Results from a Controlled Experiment

Simone Romano,* Christopher Vendome,§ Giuseppe Scanniello,* and Denys Poshyvanyk§
*University of Basilicata, Potenza, Italy
§The College of William & Mary, Williamsburg, VA, USA
Email: {simone.romano; giuseppe.scanniello}@unibas.it – {cvendome; denys}@cs.wm.edu

***Abstract*—In this paper, we present the results of a controlled experiment conducted to assess whether the presence of unreachable methods in source code affects source code comprehensibility and modifiability. A total of 47 undergraduate students at the University of Basilicata participated in this experiment. We divided the participants in two groups. The participants in the first group were asked to comprehend code base containing unreachable methods and implement five change requests in that code base. The participants in the second group were asked to accomplish exactly the same tasks as the participants in the first group, however, the source code provided to them did not contain any unreachable methods. The results of the study indicate that code comprehensibility is significantly higher when source code does not contain unreachable methods. However, we did not observe a statistically significant difference for code modifiability. From these results, we distill lessons and implications for practitioners as well as possible avenues for further research.**

I. Introduction

In software engineering, the problem with unreachable code (also named dead code, unused code, or unnecessary code) is that after awhile it starts to "smell" [1]. The older it is, the stronger and more sour the odor becomes. Unreachable code may not be completely updated when software designs change. This implies that keeping unreachable code around could be harmful. Although it seems important to deal with unreachable code, surprisingly this kind of smell has received very little research attention [2], [3], [4], [5].

The main question arises: is unreachable source code actually harmful as Martin [1] asserts? Indeed, Mäntylä [6] also states that such code hinders software comprehension and makes its structure less obvious. That is, a software engineer may spend time maintaining and documenting parts of the system that are unreachable and thus never used. In such a scenario, unreachable code would adversely impact a practitioner, since the practitioner would be wasting time maintaining and documenting these regions of code. It is also possible that a developer intentionally makes source code (i.e., methods or classes) unreachable in anticipation of future changes to the software system. Predicting the future can be difficult and often this could just add unnecessary complexity to software [6]. Although it is important to study the effect of unreachable code on program comprehension and modifiability, there have been no empirical studies with developers aimed at understanding whether unreachable code is harmful or not for developers.

In this paper, we present a controlled experiment conducted to assess whether the presence of unreachable methods is harmful with respect to source code comprehensibility and modifiability. A total of 47 undergraduate students at the University of Basilicata took part in our experiment. We divided the participants into two groups. The students in the first group accomplished tasks on source code with unreachable methods, while those in the second group carried out the tasks on source code deprived of unreachable methods. Each group of students had to perform the same two tasks. First, they had to perform a source code comprehension task and then a task modifying the system. To assess comprehension, we used a questionnaire based on the recommendations by Sillito et al. [7]. Conversely, the modification task consisted of implementing five change requests. The system on which participants accomplished the modification tasks was the same system that they used in the comprehension task. Thus, the participants had a chance to acquire knowledge of the application before we asked them to implement the change requests.

Paper Structure. In Section II, we outline motivations, background, and related work, while the design of the experiment is presented in Section III. Possible threats to validity are highlighted in Section IV, while we present results and their discussion in Section V. In Section V, we also discuss possible implications for obtained results from the practitioner and researcher perspectives. Final remarks conclude the paper.

II. Background

The first book on bad smells (shortly "smells") in object-oriented software was written by Webster in 1995 [8]. The author discussed the problems related to conceptual, political, coding, and quality-assurance issues related to bad smells. On the other hand, Riel [9] proposed heuristics to characterize good software and to possibly improve its design and implementation. Later, Fowler [10] defined 22 (source code) bad smells. He also suggested how refactoring operations should be executed to remove smells. Also, Brown et al. [11], Mäntylä [6], Wake [12], and Martin [1] suggested some bad smells. While Fowler [10] did not mention a smell for dead code, Brown et al. [11] defined this kind of smell as code frozen in an ever–changing design. Brown et al. referred to dead code as lava flow bad smell. Mäntylä [6] defined dead code as code that has been used in the past, but not used in the current version of the application. Mäntylä asserted that if

978-1-5090-1428-6/16/$31.00 ©2016 IEEE

the number of references to a method or a class is zero, then this code is likely to be such a smell. Wake [12] focused on variables, parameters, fields, methods, and classes. They are dead or unnecessary if they are not used anywhere (perhaps other than tests). On the other hand, Martin [1] focused on dead code and dead functions. Dead code is code that is not executed (e.g., the body of an if statement that checks for a condition that cannot happen), while a dead function is a method that is never called. Although there are some differences in the definitions of this smell, it seems that dead code is the code that is never executed. That is, there is no execution that reaches this code.

In the programming language field, dead code is well known and indicates source code that never gets used [13]. For example, a statement where a local variable is initialized and never used is considered dead even if it belongs to an execution trace. Almost any compiler tries to detect this kind of code to optimize generated object code and to minimize its size. As for dead methods, they are removed from byte-code, and not from source code, only for performance reasons [14]. When working on source code, optimization approaches may make source code hard to understand for software engineers. This could be unavoidable when optimizing software and it is, in a nutshell, why unreachable methods are different in software maintenance and optimization. In software maintenance, code is modified for clarity, flexibility, maintainability, and readability reasons.

To avoid confusion, we prefer to use the term unreachable code (e.g., method or class) with respect to dead code in the rest of the paper. Although there are slight differences in the definition of this kind of smell, there is a consensus on the fact that unreachable code is harmful [1], [6], [11], [12]. However, there is a lack of empirical evidence regarding the potential impact (harm) of this smell. To bridge this gap, we conducted the controlled experiment described in this paper.

A. Related Work

Olbrich et al. [15] analyzed the historical data of two large open source software systems: Lucene and Xerces. The study investigates two smells, god class and shotgun surgery, by considering several years of development. The results suggested that god classes and classes subjected to shotgun surgery have a higher change frequency than other classes. In addition, the authors observed the presence of different phases in the evolution of smells during the system development and that smell-infected components exhibit a different change behavior. This information could be considered useful to identify risky areas of a software system that need refactoring operations.

Chatzigeorgiou and Manakos [16] studied the evolution of four kinds of smells (i.e., long method, feature envy, state checking, and god class) throughout successive versions of two open source systems (JFlex and JFreeChart). The authors found that these smells persist in systems and that their removal is often a side effect of adaptive maintenance rather than refactoring activities. The findings also suggested that, in most cases, the design problems persist up to the latest examined version accumulating as the project matures. They also performed a survival analysis on the four kinds of smells studied. The results showed that smells live for a large number of versions. This suggests that smells are a permanent problem once introduced in a software system. On the other side, Tufano et al. [17] presented the results of a large empirical study on 200 open source projects from different software ecosystems. The results contradict common wisdom stating that smells are being introduced during evolutionary tasks.

Khomh et al. [18] studied the impact of classes with smells on change-proneness and the particular impact of certain smells. In particular, they analyzed ArgoUML, Eclipse, Mylyn, and Rhino and detected 13 smells in 54 releases of these systems. The results indicated that in almost all releases of the four systems, classes with smells are more fault-prone than others. Finally, structural changes affect more classes with smells than others.

Previous work raised the awareness of the community towards the impact of smells on software development. In these studies, the detection of smells during the evolution of the systems was performed by the application of automated tools. Surprisingly, unreachable code has never been studied. We can speculate that this is due to the lack of well-established tools for detecting unreachable code. Recently, Romano et al. [3] proposed a static approach to detect unreachable methods. Since this kind of smell is difficult to statically identify due to late binding, multithreading, and reflection, the authors partially dealt with these issues by simulating late binding and multithreading. The approach also allowed the authors to overcome the issues related to the application of dynamic detection of unreachable methods; it is practically impossible to execute all the usage scenarios to detect unreachable methods [6].

In the last decade, several catalogues (e.g., [1], [6], [10], [11], [12]) have been proposed to characterize bad smells. Additionally, methods and tools have been suggested to detect and/or remove smells. There is an ongoing debate regarding the extent to which developers perceive bad smells as design problems. Yamashita and Moonen [19] performed an exploratory survey aimed at investigating developers' knowledge about smells. Their results indicate that 32% of professional developers do not know (or know little) about smells, and those developers, who were aware about bad smells, asserted that in many cases their removal is not a priority due to time constraints or lack of adequate tool support. Successively, Palomba et al. [20] conducted an empirical study aimed at providing empirical evidence on how developers perceive bad smells in which participants were asked to indicate whether the code contained a potential design problem, and if any, the nature and severity of that problem. The results provide useful insights into characteristics of bad smells.

Deligiannis et al. [21] proposed the first quantitative study on the impact of smells on software development and maintenance activities. The authors set out to better understand to what extent a god class contributes to the quality of designs developed. The experiment was conducted with 20 undergraduate students as participants. The results of their

study suggested that god classes affect the evolution of design structures and the subjects' use of inheritance. However, the study did not assess the impact of this kind of smell on the ease of participants to understand the software nor the participants' ability to execute successful comprehension tasks on this software.

Du Bois et al. [22] conducted a controlled experiment with 63 students to study the effect of decomposing god classes into several collaborating classes on source code comprehensibility. Collaborating classes were obtained by applying well-known refactoring operations. The participants were asked to perform tasks on god classes and their decompositions. The results suggested that students had more difficulty understanding the original god class than certain decompositions.

Abbes et al. [23] designed and conducted three controlled experiments, which each had 24 participants. In each experiment, participants were divided into three groups and asked to accomplish two basic tasks related to code comprehension. The level of comprehension was assessed through a questionnaire. Depending on the group, the participants were provided with source code containing an occurrence of god class, one occurrence of spaghetti code, and two occurrences of both smells. The authors gathered data regarding the NASA task load index, time to accomplish comprehension tasks, and percentage of correct answers given to the questions of the comprehension questionnaire. The results suggested that the occurrence of one of the considered types of bad smells did not significantly affect source code comprehensibility, while the combination of both types of bad smells had a significant (negative) impact on code comprehension.

There are several differences between our controlled experiment and those introduced before: we studied a kind of smell considered relevant [1], [6], [11], [12], but not adequately studied, and we analyzed the effect of unreachable methods on the modifiability of source code by asking the participants to effectively implement changes.

B. Unreachable Methods

It has been observed that the presence of unreachable methods is common in object-oriented software [3]. The identification of this kind of smell in source code might be problematic, because object-oriented languages (e.g., Java, C++, and C#) have sophisticated rules for late binding and class loading. Also, the presence of methods with similar names or parameters (i.e., overloading) could make the identification of unreachable methods difficult especially for humans. The use of reflection, development frameworks, and libraries can make the identification of unreachable methods even worse. Therefore, the identification of this kind of smell requires a deep knowledge of programming language rules and the knowledge of the source code of the software at hand. The larger the software, the harder the identification and management of unreachable methods is. Unreachable methods are likely to hinder code comprehension and make the structure of the code less obvious [6]. To demonstrate how difficult it is to deal with unreachable methods and how the presence of such a smell could affect code comprehensibility and modifiability, we report a fragment of Java code with an unreachable method in Figure 1 and discuss here a few related concerns to comprehensibility and modifiability.

```
public class GeneralDepository {
  ...
  public void removeSmallBags(Beverages bevs, int num){
    switch(bevs){
      case COFFEE:
        coffee = coffee - num;
        break;
      case TEA:
        tea = tea - num;
        break;
      case CAMOMILE:
        camomile = camomile - num;
        break;
    } }
}
public class Depository extends GeneralDepository{
  ...
  public void removeSmallBags(Beverages bevs, int num){
    try  {
      PreparedStatement stmt = conn.
      prepareStatement("UPDATE DEPOSITORY SET QUANTITY =
           ? WHERE PRODUCT = ? ");
      int tot = numberOfSmallBagsDB(bevs) - num;
      stmt.setInt(1, tot);
      stmt.setString(2, bevs.toString());
      stmt.executeUpdate();
      stmt.close();
    } catch (SQLException ex) {
      ex.printStackTrace();
    } }
}
public class Purchase {
  ...
  private GeneralDepository depository;
  public void setDepository(GeneralDepository d){
    this.depository = d;
  }
  public void purchaseVisitor(Beverages bevs, int num) {
    if (depository.areThereSmallBags(bevs, num)) {
      Euro amount = smallBagsPrice.multiply(num);
      cash.addMoney(amount);
      depository.removeSmallBags(bevs, num);
      System.out.println("done");
    } else {
      System.out.println("error");
    } }
}
public class Main {
  ...
  public void purchase() {
    ...
    purchase.setDepository(new Depository());
    ...
    purchase.purchaseVisitor(beverages, num);
  }
}
```

Fig. 1. Source code sample

Figure 1 shows four classes: `GeneralDepository`, its subclass `Depository`, `Purchase`, and `Main`. The class `GeneralDepository` represents a depository that contains small bags of beverages. It declares a method `removeSmallBags()` that removes small bags from the depository without updating the database. The class in charge of updating the database is `Depository`. Its method `removeSmallBags()` removes small bags in a persistent way. The class `Purchase` has a property `depository` and a method `purchaseVisitor()` that calls `removeSmallBags()`. It is worth mentioning that the implementation of `removeSmallBags()` called during the execution of the software depends on the dy-

namic type of the property `depository`. If we do not observe the class `Main`, it is not possible to identify which implementation of the method `removeSmallBags()` will be executed. This implies that some knowledge of the application is needed to identify unreachable methods. The class `Main` uses an object of type `Depository` (i.e., dynamic type) to set the property `depository` of `Purchase`, then it uses this object to call the method `purchaseVisitor()` of `Purchase`. By applying the rules of late binding, the method `removeSmallBags()` of `Depository` is reachable whereas `removeSmallBags()` of `GeneralDepository` is unreachable. The presence of this unreachable method could lead to potential problems comprehending this source code. The developer could wrongly assume that the method `purchaseVisitor()` of `Purchase` calls `removeSmallBags()` of `GeneralDepository`. Thus, the operation of removing small bags from the depository does not update database when actually the database is updated. The developer driven by inaccurate comprehension of the code could also modify the method `removeSmallBags()` of `GeneralDepository` in order to implement a given change request. Since the method is unreachable, each modification in its body does not have any effect on the application behavior (unless someone makes this method reachable). The programmer could build a test case to exercise `removeSmallBags()` of `GeneralDepository`. Although the implementation of the change request was correct, the behavior of the software during the execution would remain the same. Without this unreachable method the code would be more obvious, and the developer would spend less time trying to understand and modify this code.

III. Experiment

We carried out our experiment by following recommendations provided by Juristo and Moreno [24], Kitchenham et al. [25], and Wohlin et al. [26]. We reported experimental design according to the guidelines suggested by Jedlitschka et al. [27]. For replication purposes, we made the experimental material (e.g., raw data) publicly available[1].

A. Goal

Considering the Goal Question Metric (GQM) paradigm by Basili *et al.* [28], the goal of our experiment can be defined as follows: **Analyze** the presence of unreachable methods in source code **for the purpose of** evaluating their effect **with respect to** comprehensibility of unknown source code and **with respect to** modifiability of familiar source code **from the point of view** of researchers and practitioners **in the context of** novice developers/students and object-oriented software implemented in Java.

The use of GQM ensured that important aspects of our experiment were defined before the planning and the execution of the experiment took place [26]. According to our goal, we have defined and investigated the following research questions:

RQ1: Does the presence of unreachable methods penalize correctness of understanding source code if software engineers are not familiar with source code?

RQ2: Does the presence of unreachable methods penalize the effort to comprehend source code if software engineers are not familiar with source code?

RQ3: Does the presence of unreachable methods penalize correctness of modifying source code if software engineers are familiar with source code?

RQ4: Does the presence of unreachable methods penalize the effort to maintain source code if software engineers are familiar with source code?

B. Context Selection

We used LaTazza application as an experimental object. It is a coffee maker support application to manage the sales and supply of small-bags of beverages. The application supports two kinds of clients: visitors or employees. Employees can purchase beverages by paying cash or credit, while visitors can only pay by cash. The secretary can sell small-bags to clients, buy boxes of beverages, manage credit and debt of employees, check the inventory, and check the cash account. LaTazza was implemented in Java and its domain can be considered a good compromise between generality and industrial application. LaTazza was not implemented by the authors of this paper and was used in a number of empirical studies as experimental object (e.g., [29], [30], [31]). The original version of this application contained 18 Java classes with a total of 1,291 Lines of Code (LOCs). Comments were removed before computing LOCs. The number of methods is 116.

The original version of LaTazza contained both reachable and unreachable methods. The first author of this paper identified unreachable methods and then removed them in order to create a new version of LaTazza without this kind of smell. To detect unreachable methods, this author used the DUM tool [32]. After having applied this tool, the LaTazza source code was analyzed (by both inspecting and executing it) to determine whether the methods detected as unreachable were actually unreachable and to identify unreachable methods the tool was not able to detect. The inspection of the results revealed that all the methods that DUM recovered were actually unreachable. However, this tool was not able to identify four unreachable methods (e.g., never used methods in the class hierarchy). We performed refactoring operations to remove unreachable methods from the code [1]. To verify that refactorings did not modify LaTazza behavior and did not introduce syntactic errors, we removed one unreachable method at a time. We performed regression testing each time. The test cases were obtained from the original version of LaTazza [29]. The execution of these refactoring operations produced a new version of LaTazza without unreachable methods containing 18 Java classes, 1,078 LOCs, and 72 methods. The number of unreachable methods seems to confirm that this smell is quite common in code base [3].

The participants in our experiment (subsequently referred to as UniBas) were 3rd-year undergraduate students in Computer

[1] www2.unibas.it/sromano/UM.html

Science from a course on the design and implementation of information systems. The experiment was conducted as an optional exercise of this course. Participants had passed all the exams related to the following courses: Procedural Programming, Object-Oriented Programming I, and Databases. In these courses, participants gained significant experience with C/C++ and Java. The participants had sufficient level of technical maturity and knowledge of software design, development, and refactoring. To get demographic information, we asked the participants to fill out a questionnaire. Because of space limitations, further details on this questionnaire and on gathered data are not provided. The students were informed that their grade in the course in which the experiment was conducted would not be affected by their performance in UniBas.

C. Variable Selection

The control group comprised participants who were given the source code of LaTazza with only reachable methods, while treatment group included the participants who were given source code with both reachable and unreachable methods. Thus, "method" is the main factor (also named manipulated factor) in our experiment. It is a nominal variable and assumes values: UM (source code with reachable and unreachable methods) and NoUM (source code deprived of unreachable methods).

To quantify the construct "correctness of understanding," we defined a comprehension questionnaire. To assess the correctness of the answers given to this questionnaire, we used two approaches. The first was based on the information retrieval theory [33]. We called: A_s as the set of string items provided as answer to the questions in the comprehension questionnaire by the participant s and C as the correct set of items expected for the questions in the comprehension questionnaire (i.e., the oracle). We computed precision and recall as follows:

$$precision(s) = \frac{|A_s \cap C|}{|A_s|} \quad recall(s) = \frac{|A_s \cap C|}{|C|} \qquad (1)$$

Precision (i.e., the fraction of items in the answers that are correct) and recall (i.e., the fraction of correct items in the answers) measure accuracy and completeness of the answers to the comprehension questionnaire, respectively. To get a trade-off between accuracy and completeness, we used a balanced harmonic mean of precision and recall:

$$F_1(s) = \frac{2 \cdot precision(s) \cdot recall(s)}{precision(s) + recall(s)} \qquad (2)$$

This (ratio) metric takes values in between 0 and 1 and estimates the correctness of understanding construct. A value close to 1 means that the participant achieved a correct comprehension because s/he answered rather well to all the questions in the comprehension questionnaire. Conversely, a value close to 0 means that source code comprehension was bad. The measures that were utilized have been largely adopted in software engineering (e.g., [34]).

In the second approach, to assess correctness of understanding, we computed the number of correct answers divided by the number of questions in the comprehension questionnaire (also "Avg") [23]. Since questions in the comprehension questionnaire could admit more than one string items, we consider an answer as correct if and only if all the correct items are given for that question. For example, the questions "*What are the kinds of users that can buy small-bags of beverages?*" admitted as correct string items: *visitors* and *employees*. Therefore, the answer is correct if and only if a participant provides exactly both these string items. The answer is incorrect otherwise. Also in this case, the used metric is a ratio metric that assumes values in the interval $[0, 1]$, where 1 indicates the best value possible. This is the second metric used to assess the construct correctness of understanding.

To quantify the construct "correctness of modification," we provided five change requests to the participants. Implemented modifications were assessed with respect to a balance between the accuracy and completeness of implemented change requests. To this end, we used:

$$precision(s) = \frac{|M_s \cap CM|}{|M_s|} \quad recall(s) = \frac{|M_s \cap CM|}{|CM|} \qquad (3)$$

where M_s is the set of modifications the participant s implemented; CM is the correct set of implementations. $|M_s \cap CM|$ indicates the number of correctly implemented change request. A change request was considered correctly implemented if and only if all the test cases in the associated suite passed. Before the experiment took place, we defined a test suite for each defined change request. This suite has been only used to evaluate the modifications the participants implemented. To get a trade-off between precision and recall (Equation 3), we used a balanced harmonic mean as that shown in Equation 2. The used metric is a ratio metric that assumes values in the interval $[0, 1]$. The ideal value is 1, whereas the worst value is 0. For example, the value of 1 indicates that the participant implemented all five change requests correctly.

To determine the "effort" to accomplish comprehension and modification tasks ("understanding effort" and "modification effort" constructs, respectively), we used the overall time (expressed in minutes) to accomplish each of these tasks. The higher the value, the higher the effort to accomplish the comprehension tasks. We consider the time as an approximation for effort. This means that aspects related to the effort (e.g., cognitive effort of participants) were not measured. This is customary in the literature and it is compliant with the ISO/IEC 25000 standard [35] definition that effort is the productive time associated with a specific project task.

D. Experiment Design

We used one factor with two treatments design [26]. This is a simple experiment design for comparing two treatment means. The design setup uses the same experimental object (i.e., LaTazza) for both treatments (i.e., UM vs. NoUM). Each participant used only one treatment on one object. We used a completed randomized variant, where each participant was randomly assigned to each group. At the end, groups were not balanced since we assigned 20 participants to NoUM and 27 to UM, respectively.

E. Hypotheses

We have defined and tested the following null hypotheses:

- Hn0: The mean value of the correctness of understanding (assessed by applying F_1 and Avg) for NoUM is the same as the mean value of the correctness of understanding for UM.
- Hn1: The mean value of the effort to accomplish a comprehension task when using NoUM is the same as that when using UM.
- Hn2: The mean value of the correctness of modifications for NoUM is the same as the mean value of the correctness of modifications for UM.
- Hn3: The mean value of the effort to accomplish a modification task when using NoUM is the same as that when using UM.

When a null hypothesis (i.e., $\mu_1 = \mu_2$) is rejected, it is possible to accept the alternative ones that can be easily derived ($\mu_1 < \mu_2$ or $\mu_1 > \mu_2$), namely mean values of a dependent variable for UM an NoUM are not the same. It is worth mentioning that Hn0 will be tested by using both of the previously defined metrics. The use of two metrics for the same construct allows reducing construct validity threats.

F. Experiment Tasks

We asked the participants to perform the following tasks:

- *Comprehension task*. Independently from the treatment, the comprehension questionnaire was composed of five open questions. These questions were chosen on the basis of the recommendations by Sillito et al. [7]. Some questions focused on expanding points in the source code viewed to be relevant and related to software comprehension, often by exploring relationships among entities (e.g., classes and method). Other questions were concerned with understanding concepts in the source code that involved multiple relationships and software entities. Answering these questions required understanding the overall structure of a subgraph.[2] All the five questions in the questionnaire were formulated using a similar form/schema. Once the comprehension task was completed, we asked the participants to return all the materials and answer a post-experiment survey. This questionnaire had to be filled out online; we used Google forms to create it. The goal of this questionnaire was to get feedback about participants' perceptions of the experiment execution. There were also questions on how participants dealt with the comprehension task (e.g., used regular expressions).
- *Modification task.* We defined five change requests (the same for both UM and NoUM) and asked the participants to implement them. An example of a change request is: "Change the cost of the box of small-bags of beverages. The new cost must be 28 €." Once a modification task was accomplished, participants had to return experimental materials (i.e., printed textual descriptions of change requests, where start and stop time needed to be reported) and the source code of LaTazza (i.e., the version modified by the students) to the experiment supervisors. Similar to the comprehension task, we asked participants to fill out a post-experiment survey online. The goal of this questionnaire was to obtain feedback about participants' perceptions of the experiment execution and used tools.

To accomplish comprehension and modification tasks, we did not impose any rules on the participants. For example, a participant could start answering a question from the comprehension questionnaire and then he/she could pass to another question and finally he/she could return back to the prior question. It is worth mentioning that because of space limitations we do not provide the details on the mentioned post-experiment questionnaires.

G. Experiment Procedure

Before the experiment, all the participants had to fill out a pre-questionnaire. The gathered information allowed us to better characterize the experimental context. Just before the experiment, the experiment supervisors (the first and the third authors) highlighted the study goal without providing details on its hypotheses. The experimental tasks were carried out under controlled conditions in two subsequent laboratory sessions. A break of 10 minutes was allowed between the comprehension and modification tasks to reduce fatigue effect in the second task. We monitored participants to prevent their communication with each other both in the laboratory sessions and in the 10-minute break. Experiment supervisors were the same in each session. No training session on tasks similar to the ones in the experiment was carried out because of the following reasons: *(i)* they should have adequate experience in performing maintenance operations on source code written by others, *(ii)* they performed homework[3] on legacy code written in Java before the experiment, and *(iii)* time and logistical constraints made the execution of a training session infeasible.

We asked all the participants to use Eclipse and the following steps (experimental procedure) for the comprehension task: *(i)* writing down their name and start-time; *(ii)* answering the questions in the comprehension questionnaire executing or not source code; and *(iii)* writing down the end-time. The steps *(i)* and *(iii)* were the same for the modification task, while the step *(ii)* was different. In particular, in the step *(ii)* the participants had to implement five change requests mentioned before. It is worth mentioning we did not provide participants with any Eclipse plug-in to detect unreachable methods.

We allowed all the participants to use the Internet to accomplish the tasks because actual developers usually exploit this medium as support for their daily work activities. We clearly forbid the participants to use of the Internet to communicate

[2]Sillito et al. [7] saw a code base as a graph of entities (classes, methods, and fields, for example) and relationships between those entities (references and calls, for example).

[3]The lecturer asked them to add some functionality using the Test-Driven-Development approach to real software they did not know before the homework. Thus, participants acquired familiarity with bad smells, refactoring, and source code written by others.

with each another. In addition, participants needed the Internet to fill out post-experiment tasks.

H. Analysis Procedure

To perform data analysis, we used the R environment (www.r-project.org) with the following steps:

1) We computed descriptive statistics of the dependent variables;
2) To test null hypotheses, we applied an unpaired (or independent samples) t-test when data were normally distributed. We used the Shapiro-Wilk W test [36] (Shapiro test, in the following) to perform the normality analysis of the data. If the normality assumption did not hold, we applied the Wilcoxon rank-sum test [37] (also known as the Mann-Whitney U test). This is a non-parametric test alternative to the unpaired t-test;
3) We studied the magnitude of differences between two groups. In the context of parametric analyses, we opted for the Cohen's d [38] effect size, while for the Cliff's d [39] in the context of non-parametric analyses;
4) We also took into account statistical power. We determined statistical power post-hoc [40]. In this case, if a null hypothesis is rejected statistical power represents the probability that a statistical test rejects a null hypothesis when it is actually false. The highest value is 1, while 0 is the lowest. The higher the statistical power value, the higher the probability to reject a null hypothesis when it is actually false. The statistical power is computed as 1 minus the Type-II-Error, also named β-value. This type of error indicates that a null hypothesis is false, but the statistical test erroneously fails to reject it. We use β-value when a statistical test was unable to reject a null hypothesis. The higher the β-value, the better it is. A standard value for adequacy for both statistical power and β-value is 0.80 [41].
5) To summarize and analyze raw data and to support their discussion, we relied on boxplots [26].

In all performed statistical tests, we decided (as it is customary) to accept a probability of 5% (i.e., α value) of committing Type-I-error [26].

IV. Threats to Validity

To better understand the strengths and limitations of our experiment, we present and discuss threats that could affect the results and their generalization. Despite our efforts to mitigate as many threats as possible, some of them are unavoidable. We discuss the threats using the guidelines by Wohlin et al. [26].

A. Internal Validity

- *Maturation.* The adopted experimental design avoided the presence of carry-over effects [26] in the execution of the comprehension task.
- *Diffusion or imitation of treatments.* This threat concerns information exchanged among the participants within each experiment. Experiment supervisors monitored the participants to prevent their communication to each another. As an additional measure to prevent diffusion of material, we asked participants to return back material at the end of each task.
- *Selection.* The effect of letting volunteers take part in the experiment may influence the results, since volunteers are generally more motivated.

B. External Validity

- *Interaction of selection and treatment.* Relying on students as participants may affect the generalizability of the results with respect to practitioners [42], [43]. However, the tasks to be performed did not require a high level of industrial experience, so we believe that relying on students as participants could be considered appropriate, as suggested in literature [44]. Working with students also implies various advantages, such as the fact that the students' prior knowledge is rather homogeneous, there is the possible availability of a large number of participants [45], and there is the chance to test experimental design and initial hypotheses [46].
- *Interaction of setting and treatment.* In our study, the kind of experimental object (i.e., LaTazza) and its complexity may affect the result validity. Also, the size might affect external validity. The difference in the size of the experimental object with and without unreachable methods might also affect the validity of results. The use of Eclipse might have impacted the result validity; it could be possible that a few participants in the experiment were more familiar with Eclipse than others.

C. Construct Validity

- *Interaction of different treatments.* To mitigate it, we adopted one factor with two treatments design [26].
- *Confounding constructs and level of construct.* We randomly assigned the participants to the treatments.
- *Evaluation apprehension.* We mitigated this threat because the participants were not evaluated on the results they achieved in the tasks. The participants were also unaware of the objectives and the experimental hypotheses.
- *Experimenters' expectancies.* Experimenters can bias results both consciously and unintentionally based upon expectations of an experiment. We mitigated this kind of threat in several directions. We downloaded the application used in the study from the web. Smells in this application were identified by applying the DUM tool [32] and manually validated. The researchers who analyzed participants' artifacts and analyzed raw data were different.
- *Mono-method bias.* We mitigated this kind of threat because we used different kinds of measures to assess the same construct.
- *Reliability of measures.* This threat is related to how response (dependent) variables were measured. Regarding task completion time, we asked the participants to write start and stop times (e.g., [47]).

TABLE I
DESCRIPTIVE STATISTICS

Construct	Metric	NoUM		UM	
		Mean	St. Dev.	Mean	St.Dev.
Correctness of understanding	F_1	0.8678	0.0606	0.7586	0.1004
	Avg	0.68	0.1196	0.5481	0.1718
Understanding effort	Time	35.05	6.9772	33.7	7.9895
" Correctness of modification	F_1	0.5447	0.211	0.5424	0.2338
Modification effort	Time	52.95	8.153	51.67	12.7189

TABLE II
RESULTS FOR HN0, HN1, HN2, AND HN3 ($^+$ INDICATES THE CASES IN WHICH WE PERFORMED PARAMETRIC ANALYSES, $p - values$ IN BOLD INDICATE THAT THE CORRESPONDING NULL HYPOTHESIS WAS REJECTED)

Hypothesis	Metric	p-value	Cohen/Cliff's d	Perc. difference	Stat. Power/β-value
Hn0	F_1 (+)	$<$ **0.0001**	large (1.317)	14.395%	0.925
	Avg	**0.008**	medium (0.428)	24.065%	0.835
Hn1	Time (+)	0.542	negligible (0.179)	4.006%	0.946
Hn2	F_1	0.956	negligible (0.011)	0.424%	0.955
Hn3	Time	0.575	negligible (0.094)	2.417%	0.937

- *Reliability of treatment implementation.* A possible threat concerns the fact that we did not impose any time limit to perform the tasks.

D. Conclusion Validity

- *Low statistical power.* The power of a statistical test concerns its ability to reveal a true pattern in the data. If the power is low, there is a high risk that an erroneous conclusion is drawn. We computed post-hoc statistical power. In addition, the number of participants in our experiment was large enough.
- *Random heterogeneity of participants.* We drew a fair sample and conducted our experiment with participants belonging to this sample.
- *Fishing and the error rate.* Our experimental hypotheses have been rejected considering proper p-values. It is also worth mentioning that we involved 47 participants.
- *Statistical tests.* We used a parametric test to verify null hypotheses only when the assumptions for its application were verified. We used non-parametric tests otherwise.

V. EXPERIMENTAL RESULTS

In this section, we present the results of our data analysis following the aforementioned procedure.

A. Descriptive Statistics and Exploratory Data Analysis

In Table I, we report the values of mean and standard deviation for the used metrics. The distributions of the values are graphically summarized by the boxplots in Figure 2. The descriptive statistics and boxplots indicate that there is no significant difference between NoUM and UM with respect to the time to accomplish comprehension and modification tasks in all the experiments. There is also no difference for correctness of the modification task. On the other hand, there is a difference between NoUM and UM for correctness of understanding regardless of the used measure (see Figures 2(a) and 2(b) for F_1 and Avg, respectively).

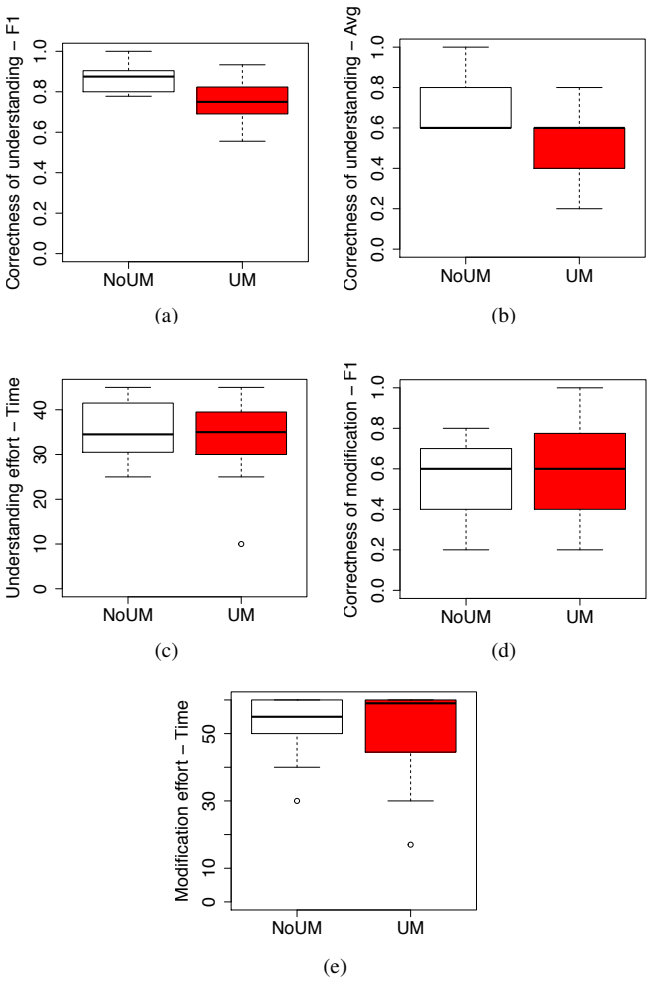

Fig. 2. Boxplots for correctness of understanding ((a) and (b)), understanding effort (c), correctness of modification (d), and understanding effort (e) grouping observation by method.

B. Hypotheses Testing

1) Hn0: Correctness of Understanding: The results for Hn0 are summarized in Table II. We rejected the null hypothesis using both of the metrics (F_1 and Avg) exploited to assess the construct correctness of understanding. That is, for correctness of understanding, a statistically significant difference was observed in favor of NoUM. As for F_1, we applied parametric analyses (e.g., an unpaired t-test to test the null hypothesis) as the results of the Shapiro test suggested (p-value was 0.079 for NoUM and 0.644 for UM). Effect size is large for F_1 and medium for Avg. The mean percentage difference values[4] indicates that the correctness of understanding is 14.395% greater for NoUM as compared with UM with respect to F_1 and 24.065% greater with respect to Avg. Statistical power is high in both cases (0.925 and 0.835, respectively).

[4]It could be difficult to relate the Cohen's d and Cliff's d values to a practical meaning. Therefore, we also used percentage difference as a less robust though more intuitive and qualitative effect size indicator. In particular, given two values μ_{NoUM} (i.e., mean of values for a given variable for NoUM) and μ_{UM} (i.e., mean of values for UM), the percentage difference is computed as $\frac{\mu_{NoUM} - \mu_{UM}}{\mu_{NoUM}}\%$.

2) Hn1: Understanding Effort: The unpaired t-test was not able to reject this hypothesis (see Table II). We applied this test because the Shapiro test returned 0.0759 and 0.0678 as the p-values for NoUM and UM, respectively. The β-value is high (0.946).

3) Hn2: Correctness of Modification: As for this construct, we applied nonparametric analyses. The Mann-Whitney U test did not rejected the null hypothesis (see Table II). The β-value is high: 0.955.

4) Hn3: Modification Effort: This hypothesis was not rejected as shown in Table II. We used Mann-Whitney U test. The β-value is high (0.937).

C. Discussion

The results indicate that the presence of unreachable methods in unknown source code significantly affects source code comprehensibility. Independently from the metric used to assess comprehension, the participants showed the lowest level of correctness in comprehension of LaTazza when its source code contained unreachable methods. Thus, we can positively answer the **RQ1** stating that the presence of unreachable methods penalizes source code comprehensibility if software engineers are not familiar with source code.

We observed that the presence of unreachable methods in source code with respect to not having them at all did not significantly affect the time to accomplish the comprehension tasks. Although this outcome did not provide definitive results, we can speculate that the presence of unreachable methods in source code does not penalize the effort to accomplish comprehension tasks (i.e., **RQ2**).

As for **RQ3** and **RQ4**, we observed that when participants became familiar with LaTazza's source code, the presence of unreachable methods did not seem to affect either correctness of modifications or time to implement modifications. Although we did not reject both Hn2 and Hn3, the distributions of the values are very similar for the metrics used to assess the constructs: correctness of modification and modification effort (see Figures 2(d) and 2(e), respectively).

D. Implications

We delineate here the main practical implications for our experiment from both *practitioner* and *researcher* perspectives and possible future directions related to these implications:

- The results suggest that the execution of refactoring operations to remove unreachable methods would help to improve the correctness of understanding of source code. This implication is especially important from the practitioners' perspective. Researchers could assist developers in defining approaches to detect and remove unreachable methods. Researchers could also be interested in further assessing the effect of developers' familiarity with source code and its correctness of understanding. It could be possible that the greater the familiarity, the lesser correctness of understanding is affected. Additionally, researchers could be also interested in assessing if observed results scale to larger software systems. Our results pose the basis for future work.
- Refactoring has a cost due to the detection and the removal of unreachable methods as well as to execution of regression testing needed to verify that faults have not been introduced, which we also had the possibility to assess while preparing experimental material for our investigation. It would be crucial to know whether this cost is adequately paid back by the improved correctness of understanding. This aspect, not considered in this paper, is of particular interest for practitioners, and makes sense thanks to our study (improved comprehension of source code deprived of unreachable methods).
- Removing unreachable methods could reduce a number of issues related to well-known software engineering processes: maintenance, testing, quality assurance, reuse, and integration [48]. This implication is very relevant for practitioners.
- The presence of unreachable methods did not significantly impact the effort to perform comprehension tasks. This outcome is of particular interest for researchers, who could assess if this also holds for larger software.
- Familiarity with source code and modifiability seem to be related to each another. Researchers could be interested in assessing the extent to which developers' familiarity with source code produces a significant effect on correctness of modification and effort.
- It seems that unreachable methods are common (e.g., [1], [3], [6]) and that their presence affects source code comprehensibility as our results suggest. Therefore, it could be reasonable to perform investigations into "when" and "why" unreachable methods are introduced in software projects and how developers deal with such smells in their projects.

VI. Conclusion

In this paper, we present a controlled experiment with the goal of gaining understanding "if" the presence of unreachable methods has any effect on source code comprehensibility and modifiability. A total of 47 students participated in this experiment. The results indicate that the presence of unreachable methods significantly affects source code comprehensibility. That is, the presence of this kind of smell reduces the participants' comprehension on source code. However, modifiability is not affected if participants have some knowledge of the source code. As future work, we plan to replicate this work further to confirm or contradict obtained results and to assess if code modifiability is affected when participants are not familiar with source code.

References

[1] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[2] H. Boomsma, B. V. Hostnet, and H. Gross, "Dead code elimination for web systems written in PHP: lessons learned from an industry case," in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 2012, pp. 511–515.

[3] S. Romano, G. Scanniello, C. Sartiani, and M. Risi, "A graph-based approach to detect unreachable methods in java software," in *ACM Symposium on Applied Computing*, 2016.

[4] G. Scanniello, "Source code survival with the Kaplan Meier estimator," in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 2011, pp. 524–527.

[5] ——, "An investigation of object-oriented and code-size metrics as dead code predictors," in *Proc. of EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, 2014, pp. 392–397.

[6] M. Mäntylä, "Bad smells in software—a taxonomy and an empirical study," Ph.D. dissertation, Helsinki University of Technology, 2003.

[7] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, Jul. 2008.

[8] B. F. Webster, *Pitfalls of object-oriented development.* M & T, 1995.

[9] A. J. Riel, *Object-Oriented Design Heuristics*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[10] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: Wiley, 1998.

[12] W. C. Wake, *Refactoring Workbook*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[13] S. K. Debray, W. S. Evans, R. Muth, and B. D. Sutter, "Compiler techniques for code compaction." *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, 2000.

[14] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proc. of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2000, pp. 281–293.

[15] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 390–400.

[16] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," *Innov. Syst. Softw. Eng.*, vol. 10, no. 1, pp. 3–18, Mar. 2014.

[17] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 403–414.

[18] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness." *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[19] A. F. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey." in *Proceedings of Working Conference on Reverse Engineeringg*. IEEE Computer Society, 2013, pp. 242–251.

[20] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," in *Proceedings of International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2014, pp. 101–110.

[21] I. S. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. J. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability." *Journal of Systems and Software*, vol. 72, no. 2, pp. 129–143, 2004.

[22] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does God Class Decomposition Affect Comprehensibility?" in *IASTED International Conference on Software Engineering*, 2006.

[23] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension." in *Proceedings of Conference on Software Engineering and Maintenance*, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE Computer Society, 2011, pp. 181–190.

[24] N. Juristo and A. Moreno, *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, 2001.

[25] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Trans. on Soft. Eng.*, vol. 28, no. 8, pp. 721–734, 2002.

[26] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[27] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to Advanced Empirical Software Engineering*. Springer London, 2008, pp. 201–228.

[28] V. Basili, G. Caldiera, and D. H. Rombach, *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[29] F. Ricca, M. Torchiano, M. Di Penta, M. Ceccato, P. Tonella, and C. A. Visaggio, "Are fit tables really talking? a series of experiments to understand whether fit tables are useful during evolution tasks," in *Procedeeings of International Conference on Software Engineering*. IEEE Computer Society, 2008, pp. 361–370.

[30] A. Marchetto and F. Ricca, "From objects to services: toward a stepwise migration approach for java applications," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 6, pp. 427–440, 2009.

[31] F. Ricca, G. Scanniello, M. Torchiano, G. Reggio, and E. Astesiano, "Assessing the effect of screen mockups on the comprehension of functional requirements," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 1:1–1:38, 2014.

[32] S. Romano and G. Scanniello, "DUM-Tool," in *Proc. of International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2015, pp. 339–341.

[33] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York: McGraw Hill, 1983.

[34] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, "How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments," *IEEE Trans. on Softw. Eng.*, vol. 36, no. 1, pp. 96–118, 2010.

[35] International Organization for Standardization (ISO), *ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE)*, Std., 2005.

[36] S. Shapiro and M. Wilk, "An analysis of variance test for normality," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 1965.

[37] W. J. Conover, *Practical Nonparametric Statistics*. Wiley, 3rd Edition, 1998.

[38] J. Cohen, *Statistical power analysis for the behavioral sciences (2nd ed.)*. Hillsdale, NJ: Lawrence Earlbaum Associates, 1988.

[39] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information & Software Technology*, vol. 49, no. 11-12, pp. 1073–1086, 2007.

[40] T. Dybå, V. B. Kampenes, and D. I. K. Sjøberg, "A systematic review of statistical power in software engineering experiments." *Information & Software Technology*, vol. 48, no. 8, pp. 745–755, 2006.

[41] P. Ellis, *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*. Cambridge University Press, 2010.

[42] M. Ciolkowski, D. Muthig, and J. Rech, "Using academic courses for empirical validation of software development processes," *EUROMICRO Conference*, pp. 354–361, 2004.

[43] J. Hannay and M. Jørgensen, "The role of deliberate artificial design elements in software engineering experiments," *IEEE Trans. on Soft. Eng.*, vol. 34, pp. 242–259, March 2008.

[44] J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "Issues in using students in empirical studies in software engineering education," in *Proceedings of International Symposium on Software Metrics*. IEEE CS Press, 2003, pp. 239–.

[45] J. Verelst, "The influence of the level of abstraction on the evolvability of conceptual models of information systems," in *Procedeedings of the International Symposium on Empirical Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 17–26.

[46] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *IEEE Trans. on Soft. Eng.*, vol. 31, no. 9, pp. 733–753, 2005.

[47] L. Huang and M. Holcombe, "Empirical investigation towards the effectiveness of test first programming," *Information & Software Technology*, vol. 51, no. 1, pp. 182–194, 2009.

[48] G. Canfora and M. Di Penta, "New frontiers of reverse engineering," in *Workshop on the Future of Software Engineering*, 2007, pp. 326–341.