A Multi-Study Investigation Into Dead Code

Simone Romano, *Member, IEEE,* Christopher Vendome, *Member, IEEE Computer Society,* Giuseppe Scanniello, *Member, IEEE,* and Denys Poshyvanyk, *Member, IEEE Computer Society*

Abstract—Dead code is a bad smell and it appears to be widespread in open-source and commercial software systems. Surprisingly, dead code has received very little empirical attention from the software engineering research community. In this paper, we present a multi-study investigation with an overarching goal to study, from the perspective of researchers and developers, $when$ and why developers introduce dead code, how they perceive and cope with it, and $whether$ dead code is harmful. To this end, we conducted semi-structured interviews with software professionals and four experiments at the University of Basilicata and the College of William & Mary. The results suggest that it is worth studying dead code not only in the maintenance and evolution phases, where our results suggest that dead code is harmful, but also in the design and implementation phases. Our results motivate future work to develop techniques for detecting and removing dead code and suggest that developers should avoid this smell.

Index Terms—Dead Code, Unreachable Code, Unused Code, Bad Smell, Empirical investigation, Multi-study.

1 Introduction

In software engineering, dead code is unnecessary source code, because it is unused and/or unreachable (*i.e.,* never executed) [1], [2]. The problem with dead code is that after a while it starts to "smell bad." The older it is, the stronger and more sour the odor becomes [3]. This is because keeping dead code around could be harmful [1], [4], [5]. For example, Mäntylä *et al.* [1] stated that dead code hinders the comprehension of source code and makes its structure less obvious, while Fard and Mesbah [4] asserted that dead code affects software maintainability, because it makes source code more difficult to understand. In addition, developers could waste time maintaining dead code [5].

Dead code seems to be quite common too [5], [6], [7], [8]. For example, Brown *et al.* [6] reported that, during the code examination of an industrial software system, they found a large amount of source code (between 30 and 50 percent of the total) that was not understood or documented by any developer currently working on it. Later, they learned that this was dead code. Boomsma *et al.* [7] reported that on a subsystem of an industrial web system written in PHP, the developers removed 2,740 dead files, namely about 30% of the subsystem's files. Eder *et al.* [5] studied an industrial software system written in .NET in order to investigate how much maintenance involved dead code. They found that 25% of all method genealogies[1] were dead. Romano *et al.* [8] focused on dead methods in desktop applications written in Java. They reported that the percentage of dead methods in these applications ranged between 5% and 10%.

Furthermore, Yamashita and Moonen [9] reported that dead code detection is one of the features software professionals would like to have in their supporting tools.

Although there is some consensus on the fact that dead code is a common phenomenon [5], [6], [7], [8], it could be harmful [1], [4], [5], and it seems to matter to software professionals [9]; surprisingly, dead code has received very little empirical attention from the software engineering research community.

In this paper, we present a multi-study investigation with multiple goals to understand $when$ and why developers introduce dead code, how they perceive and cope with it, and $whether$ dead code is harmful. To this end, we conducted semi-structured interviews with software professionals and four experiments with students (a few of them had professional experience) from the University of Basilicata (Italy) and the College of William & Mary (USA). Our results demonstrate that it is worth studying dead code not only in the maintenance and evolution phases, where our results indicate that dead code is harmful, but also in the design and implementation stages. Our empirical results motivate future work on this topic to develop techniques for detecting and removing dead code.

Paper structure. In Section 2, background information is provided and related work is discussed. In Section 3, we motivate our multi-study investigation into dead code, which is then described is Section 4. The semi-structured interviews are presented in Section 5, while the results from these interviews and the threats that could affect their validity are presented in Section 6. Similarly, we first introduce the experiments in Section 7, and then the obtained results and the threats to validity in Section 8. The overall discussion of the results is presented in Section 9. Final remarks conclude the paper.

- *S. Romano and G. Scanniello are with University of Basilicata, Potenza (PZ), Italy.*
  E-mail: simone.romano@unibas.it and giuseppe.scanniello@unibas.it
- *C. Vendome and D. Poshyvanyk are with The College of William and Mary, Williamsburg, VA, USA.*
  E-mail: cvendome@cs.wm.edu and denys@cs.wm.edu

1. A method genealogy is the list of methods that represent the evolution of a single method over different versions of a software system [5].

2 Background and Related Work

2.1 Background

Bad smells (shortly "smells") are symptoms of poor design and implementation choices [10]. Fowler [11] defined 22 smells together with instructions on how to remove them, *i.e.,* refactoring operations or simply refactorings. Fowler, in his book, did not mention dead code. On the contrary, Brown *et al.* [6] referred to dead code as lava flow, namely unused code frozen in an ever–changing design. Mäntylä *et al.* [1] defined dead code as source code used in the past, but currently never executed. Wake [2] referred to dead code as unused variables, parameters, fields, methods, or classes. Martin [3] defined dead code as never executed code (*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. In both the cases, dead code and dead functions are not contained in any program execution trace.[2] Although there are some slight differences among the definitions provided before, it seems that dead code is unnecessary because it is unused and/or unreachable.

The term "dead" is also used in the programming languages field, where dead code refers to computations whose results are never used (*e.g.,* a variable referenced in the code, but not actually used at run-time) [12]. A compiler removes dead code from a program in the optimization phase, namely it improves the intermediate representation of this program for optimization reasons (*e.g.,* faster code and/or shorter code). This implies that dead code is not removed from the source code, but from its intermediate representation; therefore, developers often are unaware of the presence and the removal of dead code.

In summary, dead code assumes a different meaning both within the same field (*i.e.,* software engineering) and between different fields (*i.e.,* software engineering and programming languages). In this paper, we refer to the software engineering definition of dead code and we use the term "dead" to also refer to lava flow, unreachable, and unused code. Whenever needed, we distinguish among the following kinds of dead code: classes, methods, variables (local and global), parameters, and statements.

2.2 Related Work

Chen *et al.* [13] proposed a data model for C++ software repositories that supported reachability analysis and dead code detection of both C++ and C entities (*e.g.,* variable, functions, and types). Dead entities were detected by means of the difference of two sets, S and $R(r)$, where S was the set of all the entities in a software system, while $R(r)$ was the set of entities reachable from the entity r (*i.e.,* the starting point of the software execution). Boomsma *et al.* [7] proposed a dynamic approach to identify and then eliminate dead files from web systems written in PHP. This approach consisted of monitoring the execution of web systems in a given time frame for the purpose of gathering data on the usage of PHP files. If a file was not used in the considered time frame, it was marked as potentially dead. The authors evaluated their approach on an industrial web system comprising six subsystems. They reported that, on Aurora (one of the subsystems), thanks to the authors' approach the developers removed 2,740 dead files. Fard and Mesbah [4] presented JSNOSE, a smell detection approach for web systems with client-side written in JavaScript. JSNOSE used static and dynamic analyses to detect 13 smells, including dead code, in client-side code. JSNOSE detected dead code (dead statements in particular) by either counting the execution of statements or reachability of statements. Romano *et al.* [8] proposed DUM, a static approach to detect dead code at method level in Java desktop application. The approach consisted of building a graph-based representation of methods and their relationships (*i.e.,* invocations). Then, this representation was traversed to detect dead methods. Nodes (*i.e.,* methods) reachable from a starting node, in the graph-based representation, were considered alive. All the other nodes were marked as dead. Romano *et al.* [14] implemented DUM in an Eclipse plug-in named as DUM-Tool. The main difference among our contribution and the papers discussed just before is that they proposed approaches and tools to detect dead code, while we investigated *when* and *why* developers introduce dead code, *how* they perceive and cope with it, and *whether* dead code is harmful.

Scanniello [15] defined an approach based on the Kaplan Meier estimator to analyze how dead code (dead methods specifically) affects five evolving software systems. The author observed that, on two software systems, the developers avoided introducing dead code as much as possible, thus suggesting that the removal of dead code was perceived for the developers as relevant. Later, Scanniello [16] conducted an investigation into 13 software metrics as predictors for dead code (dead methods in particular). Five out of 13 software metrics were identified as predictors for dead code. LOC (Lines of Code) seemed to be the best predictor, thus suggesting that the larger a class is, the higher the probability that its methods are dead code. Eder *et al.* [5] investigated on maintenance of dead code in an industrial web system written in .NET. The authors monitored the execution of methods in a given time frame. Methods not executed in this time frame were considered as dead. Then, they quantified the maintenance operations affecting dead methods. The authors observed that maintenance of dead code account for 7.6% of the total number of modifications. The differences with respect to these papers is that we conducted a multi-study investigation whose overarching goal is to study dead code from the developers' perspective. That is, we have conducted a user study in addition to studying the evolution of some software systems as our motivation.

Yamashita and Moonen [9] conducted a survey to investigate developers' knowledge about smells. The results from this survey suggested that 32% of the respondents did not know about smells. The most popular smell was duplicate code. Dead code was reported as the 10^{th} most popular smell among the 34 smells that the respondents mentioned. Dead code detection resulted as the 10^{th} most desired features (out of 29) for smell analysis tools. This is the sole study that reports results on developers' perceptions about dead code. In some sense, our multi-study investigation extends this survey and provides further insights into dead code (*e.g.,* its harmfulness).

3 Dead Code

In this section, we provide an example to show why dead code is believed to be harmful when comprehending and modifying source code. We also present a mining study to

2. Execution traces record information about the execution of a software system.

```java
public class Dot extends Figure {

  @Override
  public void rescaleX(double formerX, double newX,
        double percent, LaTeXDrawRectangle bound){
    if(percent==1.) return;
    LaTeXDrawPoint2D NW = bound.getTheNWPoint();
    SE = bound.getTheSEPoint(), farest;
    if(Math.abs(newX-SE.x)<Math.abs(newX-NW.x))
      farest = NW;
    else
      farest = SE;
    center.x = farest.x+(center.x-farest.x)*percent;
    updateDelimitor();
    updateShape();
    updateBorders();
    updateGravityCenter();
  }

  @Override
  public void rescaleY(double formerY, double newY,
        double percent, LaTeXDrawRectangle bound){
    if(percent==1.) return;
    LaTeXDrawPoint2D NW = bound.getTheNWPoint();
    SE = bound.getTheSEPoint(), farest;
    if(Math.abs(newY-SE.y)<Math.abs(newY-NW.y))
      farest = NW;
    else
      farest = SE;
    center.y = farest.y+(center.y-farest.y)*percent;
    updateDelimitor();
    updateShape();
    updateBorders();
    updateGravityCenter();
  }

  public synchronized void setWidth(float w){
    if(w<=0) throw new IllegalArgumentException();
    width = w;
    updateGap();
    updateDelimitor();
    updateBorders();
    updateShape();
  }

  ...
```

(a)

```java
public class Draw extends Figure {

  @Override
  public void rescaleX(double formerX, double newX, double
        percent, LaTeXDrawRectangle bound){
    if(percent==1.) return;
    if(bound==null) throw new IllegalArgumentException();
    int i, size = figures.size();
    for(i=0; i<size; i++)
      figures.elementAt(i).rescaleX(formerX, newX, percent,
            bound);
    updateBorders();
  }

  @Override
  public void rescaleY(double formerY, double newY, double
        percent, LaTeXDrawRectangle bound) {
    if(percent==1.) return;
    if(bound==null) throw new IllegalArgumentException();
    int i, size = figures.size();
    for(i=0; i<size; i++)
      figures.elementAt(i).rescaleY(formerY, newY, percent,
            bound);
    updateBorders();
  }

  ...
```

(b)

```java
public class ParametersDrawFrame extends
      AbstractParametersFrame {

  @Override
  public void saveParameters(){
    if(!(figure instanceof Draw))
      throw new ClassCastException();
    LaTeXDrawRectangle borders = figure.getBorders();
    LaTeXDrawPoint2D NW = borders.getTheNWPoint();
    LaTeXDrawPoint2D SE = borders.getTheSEPoint();
    float newVal = Float.valueOf(NWX.getValue().toString())
          .floatValue();
    figure.rescaleX(NW.x, newVal, Math.abs((newVal-SE.x)/(
          NW.x-SE.x)), borders);
    newVal = Float.valueOf(NWY.getValue().toString()).
          floatValue();
    figure.rescaleY(NW.y, newVal, Math.abs((newVal-SE.y)/(
          NW.y-SE.y)), borders);
    newVal = Float.valueOf(SEX.getValue().toString()).
          floatValue();
    figure.rescaleX(SE.x, newVal, Math.abs((newVal-NW.x)/(
          SE.x-NW.x)), borders);
    newVal = Float.valueOf(SEY.getValue().toString()).
          floatValue();
    figure.rescaleY(SE.y, newVal, Math.abs((newVal-NW.y)/(
          SE.y-NW.y)), borders);
    ((Draw)figure).updateBorders();
    ((Draw)figure).updateGravityCenter();
    super.saveParameters();
  }

  ...
```

(c)

Fig. 1: An excerpt of LaTeXDraw 2.0.8 concerning two dead methods.

understand both the prevalence of dead code in open-source software systems and to what extent the amount of such a smell changes during software evolution and maintenance. Finally, we summarize the motivations behind our multi-study investigation.

3.1 Dead Code vs. Comprehensibility and Modifiability

To demonstrate how the presence of dead code (in particular, dead methods) could affect source code comprehensibility and modifiability, we consider an excerpt of a real application written in Java (see Figure 1), namely LaTeXDraw 2.0.8 (*i.e.,* one of the experimental objects used in the experiments). LaTeXDraw is an open-source graphical drawing editor for LaTeX. Among the classes of LaTeXDraw, we report: `Dot` (see Figure 1.a), `Draw` (see Figure 1.b), and `ParametersDrawFrame` (see Figure 1.c). The `Dot` class represents a dot (*i.e.,* one of the drawable items of LaTeXDraw), whereas the class `Draw` is a container of drawable items (*e.g.,* it can contain rhombuses, squares, etc). The `ParametersDrawFrame` class is a dialog box that allows changing the parameters of a `Draw` object. Please note that we do not show the classes representing other drawable items of LaTeXDraw (*e.g.,* rhombus) because of their scarce relevance.

LaTeXDraw allows the user to scale both dots and containers of drawable items. When the user scales a given dot, `setWidth` of `Dot` is executed and consequently the dot is scaled. On the other hand, when the user scales a container of drawable items, `rescaleX` and `rescaleY` of `Draw` are executed; then, the `rescaleX` and `rescaleY` methods of any contained item are executed as well (see the calls in the lines 9 and 19 in Figure 1.b). LaTeXDraw does not allow creating `Draw` objects containing `Dot` objects. This implies that the calls to `rescaleX` and `rescaleY` (see the lines 9 and 19 in Figure 1.b, and the lines 11, 13, 15, and 17 in Figure 1.c) can never cause the execution of `rescaleX` and `rescaleY` of `Dot`. It is worth mentioning that besides the calls in Figures 1.b and 1.c, there is no caller in LaTeXDraw that can cause the execution of `rescaleX` and `rescaleY` of `Dot`. This is why these two methods are dead.

The presence of these two dead methods could lead to potential problems when comprehending and maintaining the source code of LaTeXDraw. We discuss these potential problems by considering the following three scenarios:

- **Newcomer developer modifies dead code.** Let us suppose that Alice, a newcomer developer, has to perform a change task on LaTeXDraw that consists of modifying how to scale a dot. Sillito *et al.* [17] reported that newcomer developers are often interested in finding a few focus points[3] at the beginning of a change task. Later, these focus points can be expanded. According to Sillito *et al.*'s results, we can postulate that Alice could start searching focus points such as "dot" and "scale". For example, she could use "dot" as a query of a text-based search to find that the `Dot` class represents a dot. Then, she could scroll the `Dot` class to search some methods named something like "scale" and she could wrongly consider `rescaleX` and `rescaleY` relevant to her change task. Alice could also try to expand her focus point to be more confident that `rescaleX` and `rescaleY` are responsible for scaling a dot. Thus, Alice could look for where these two methods are called and find out possible calls in `Draw` and `ParametersDrawFrame`. Without knowing that a `Draw` object cannot contain a `Dot` object, Alice could modify `rescaleX` and `rescaleY` of `Dot` and subsequently she could spend time understanding why the modifications are not affecting LaTeXDraw's behavior. In this context, newcomer developers would actually implement the change task only after understanding why the modifications do not affect LaTeXDraw's behavior.
- **Developer with some knowledge of the code base modifies dead code.** Let us consider again the change task that consists of modifying how to scale a dot, but unlike the scenario before, the developer (*i.e.,* Bob) has some knowledge of the LaTeXDraw code base. In particular, Bob has never dealt with dots and containers of drawable items, but he has already worked on the source code of another drawable object (*e.g.,* a rhombus) and he knows that the methods responsible for scaling this drawable object are `rescaleX` and `rescaleY`. He could think that this is also true for a `Dot` object. In this context, developers could modify `rescaleX` and `rescaleY` of `Dot` and then they would need to spend time understanding why their modifications do not affect LaTeXDraw's behavior before correctly implementing the change task.
- **(Newcomer or not) developer revives dead code.** When performing the change task mentioned in the previous scenarios, the developers could try to revive `rescaleX` and `rescaleY` of `Dot`. That is, Alice and Bob could write code that causes the execution of these two methods. Unfortunately for them, `rescaleX` and `rescaleY` of `Dot` are buggy because instead of scaling a dot, they translate it. Trying to reuse these two methods could be a waste of time since the developer (*i.e.,* a newcomer or not) should first understand why a given dot is not scaled and then fix `rescaleX` and `rescaleY` of `Dot`.

3. A focus point is a software entity (*e.g.,* a method) that is relevant to a given task [17].

3.2 Dead Code Study

In this section, we present the design of a mining study that we conducted to further motivate our multi-study investigation. The obtained results and the threats that could affect their validity are presented as well.

3.2.1 Planning and Execution

Our motivational mining study aimed to understand the prevalence of dead code (*i.e.,* dead methods) in open-source software systems and to what extent the amount of dead code changes (increases or decreases) during the evolution and maintenance of these systems. To this end, we studied the following questions:

- *How prevalent is dead code in open-source Java applications?*
- *Do developers introduce more dead code or remove dead code during the maintenance and evolution of their software?*

The first question aimed to demonstrate the scope of this smell. By understanding the prevalence, we can determine whether this can be a common issue that impacts developers with respect to source code comprehensibility and modifiability. In the second question, we aimed to understand whether developers are introducing more dead code, removing dead code, or if it is staying consistent as the system evolves. By understanding these two phenomena, we can better understand the scope of dead methods and to what extent developers both introduce and remove them. The perspective is that of researchers interested in understanding the pervasiveness of this smell.

Procedure. The study focused on open-source Java GUI-based[4] applications hosted in GitHub.[5] As an initial filtering, we looked for Java projects that had at least a fork, a star, or a watcher (*i.e.,* the projects were not abandoned), and were not a fork (*i.e.,* the projects were not duplicated). We cloned these projects from GitHub and identified those projects that were GUI-based applications. To perform this

4. GUI-based applications mean applications based on GUI frameworks (*e.g.,* Swing).
5. github.com

TABLE 1: GUI frameworks considered in the mining study together with their import declarations.

Framework	Import declarations
AWT	java.awt
JavaFX	javafx.animation, javafx.application, javafx.beans, javafx.event, javafx.util, javafx.stage, javafx.scene, javafx.collections, javafx.concurrent, javafx.embed, javafx.fxml, javafx.geometry
JGoodies	com.jgoodies.forms
QT Jambi	com.trolltech.qt
Swing	javax.swing
SwingX	org.jdesktop.swingx
SWT	org.eclipse.swt

further filtering, we looked for projects that imported the GUI frameworks listed in Table 1. We identified 2,484 candidate GUI-based Java applications. Since our dead code detection tool required the class files for the projects, we considered projects using Ant[6] and Maven,[7] and were able to successfully build 35 projects in their current snapshot.

Additionally, to understand to what extent the amount of dead code changes during software maintenance and evolution, we considered projects that were able to build for multiple commits. As expected, many projects were not able to build at many commits; for example, these commits may have bugs preventing compilation or the build scripts may be outdated to reflect the source code changes [18]. We were able to identify 13 projects out of the 35 projects that build for at least two commits. While the analysis may miss intermediate dead code additions or removals, it demonstrates to what extent dead code present within the system between two buildable versions, which could be representative of more stable snapshots or releasable versions.

Analysis Procedure. With the projects locally cloned, we utilized DUM-Tool [14] to analyze the systems that were buildable to identify the dead methods. The tool provides an XML-formatted file detailing the identified dead methods.

To study how prevalent is dead code, we considered only the most recent buildable version of each project and provide the distribution of the number of dead methods in the Java applications.

To investigate to what extent developers both introduce and remove dead code, we traversed the Git[8] version history and ran DUM-Tool at each buildable version. Then, we analyzed how the number of dead methods changed between versions of the application to identify whether more dead code was introduce, removed, or remained consistent.

3.2.2 Results

How prevalent is dead code in open-source Java applications?

We observed a distribution of dead methods with a minimum of 0 methods, median of 7.5 methods, mean of 42.12 methods, and maximum of 405 methods. Dead methods were relatively common in our dataset with 32 of 35 applications containing them. Additionally, the applications typically had multiple dead methods (31 out of 35 applications) and certain applications had a high number of dead methods. Overall, we found that on average dead methods are 15.94% of methods of an application, while the median is 11.95%. It is important to note that these methods may be intended for future use. However, our study in the subsequent sections demonstrates that this can have a negative impact with respect to source code comprehensibility and modifiability.

Do developers introduce more dead code or remove dead code during the maintenance and evolution of their software?

The three projects that we previously observed without dead methods in their current version also did not contain any dead code through their revision history. Figure 2 shows the different behaviors that we observed in the remaining 10 Java applications that did contain dead code.

We observed that the number of dead methods remained constant for 5 applications during their commit history. Of these applications, JMario and javacus had the most dead methods at 38 methods and 20 methods, respectively. The ttyhlauncher application had eight dead methods, and was followed by SimuladorAutomatos with three dead methods. Finally, piggy only had two dead methods.

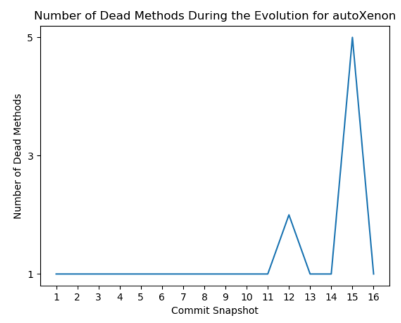
Interestingly, the other five projects showed fluctuations. In the case of autoXenon as in Figure 2.a, we observed two small peaks with the first representing two dead methods and the second five dead methods. In both cases, the number decreased back to 0 in the subsequent commit. Figure 2.e shows a similar behavior for InvApp, but it had a much larger increase. We observed the lack of dead methods through the revision history followed by a brief large spike to 60 dead methods later in the development that lasts only for that version.

In the case of BatesPapoServidor (see Figure 2.b), we observed the most fluctuations in terms of the amount of dead methods. Initially, we observed a large number of dead methods were removed. After two commits, we observed a slightly larger increase of dead methods, which were subsequently removed in the following commit. After the removal the another large increase in the number of dead methods occurred and remained relatively consistent after a minor decrease (there was an increase for a single commit, but decrease to the previous number of dead methods in the next commit).

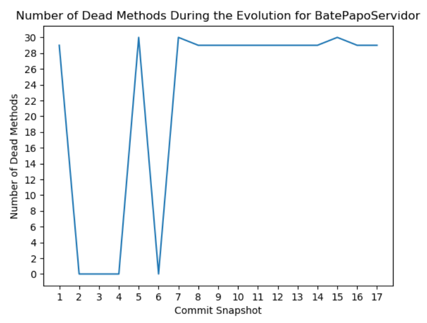
In Figure 2.c, we observed an initial period in the first nine commits where the number of dead methods were decreased to 0. Then, we observed this remains consistent for the next ten commits after which the number increases and remained consistent. Conversely, in Figure 2.d, we observed that FBOS initially had a small number of dead methods that remained relatively constant until the 8^{th} commit, where there was an increase to 40 dead methods. It remained constant for three commits and then took a sharp decrease in dead methods for a commit followed by a slight increase.

By considering the revision histories, there were two main observations about the number of dead methods. First, we observed a consistent number of dead methods in five of the applications. Second, we observed the other projects had periods where a large number of dead methods were introduced within the application. In both cases, it is important to understand the impact that these dead methods might have with respect to source code comprehensibility and modifiability, since it was relatively common for projects in our dataset to have dead methods throughout their

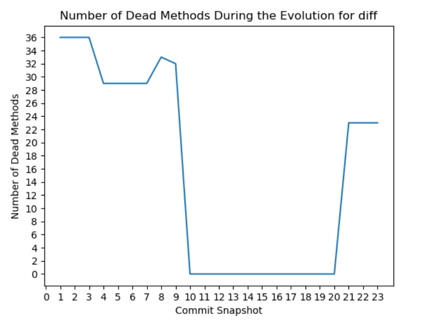
6. ant.apache.org
7. maven.apache.org
8. git-scm.com



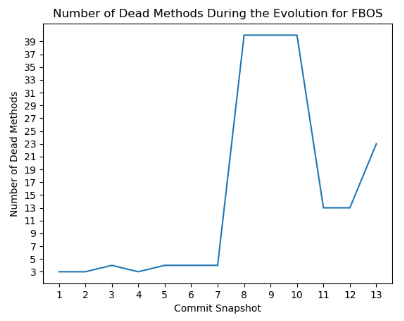
(a) autoXenon



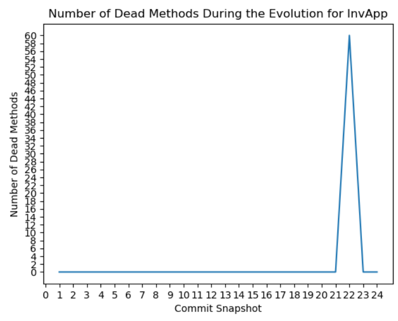
(b) BatePapoServidor



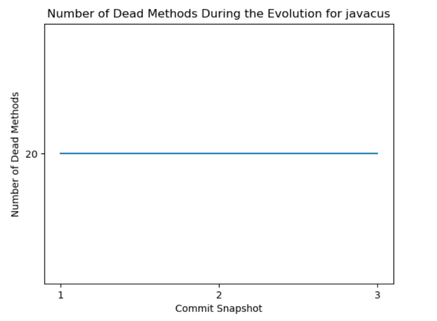
(c) diff



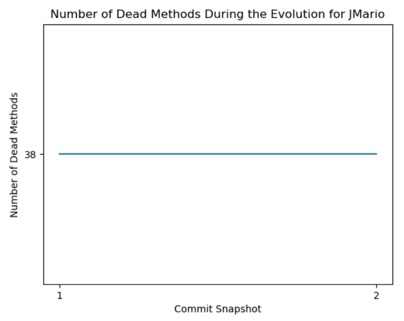
(d) FBOS



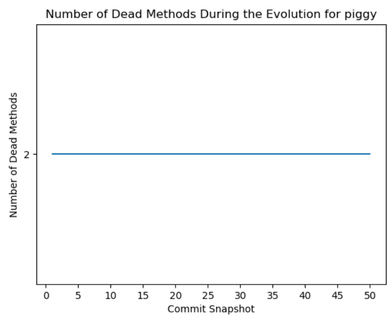
(e) InvApp



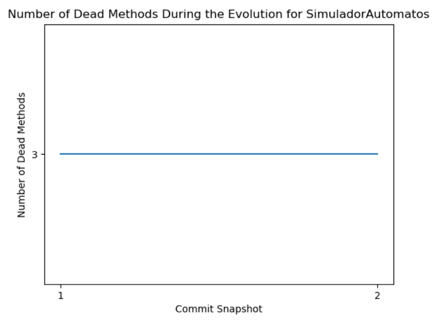
(f) javacus



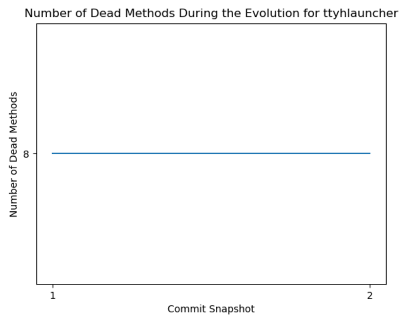
(g) JMario



(h) piggy



(i) SimuladorAutomatos



(j) ttyhlauncher

Fig. 2: Number of dead methods at each revision for each project.

development histories with instances where a large number of these dead methods may also be introduced within a single commit.

3.2.3 *Threats to Validity*

Threats to validity are reported and discussed by following the guidelines by Wohlin *et al.* [19].

Internal Validity. In terms of factors that could bias our results, we performed filtering to remove duplicate projects or abandoned projects. Then, we considered all GUI-based applications as other software systems such as libraries would yield misleading results as methods (*e.g.*, APIs) may inaccurately be identified as dead, which would inflate our results. Due to both the number of candidate applications

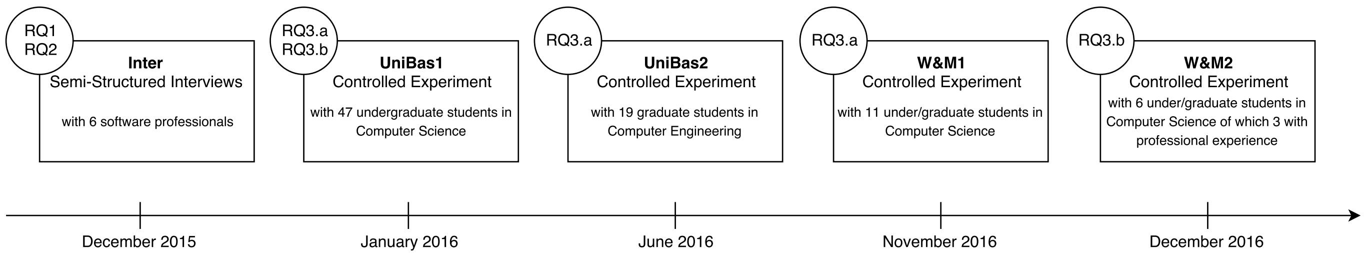

RQ1
RQ2
Inter
Semi-Structured Interviews
with 6 software professionals
RQ3.a
RQ3.b
UniBas1
Controlled Experiment
with 47 undergraduate students in Computer Science
RQ3.a
UniBas2
Controlled Experiment
with 19 graduate students in Computer Engineering
RQ3.a
W&M1
Controlled Experiment
with 11 under/graduate students in Computer Science
RQ3.b
W&M2
Controlled Experiment
with 6 under/graduate students in Computer Science of which 3 with professional experience
December 2015
January 2016
June 2016
November 2016
December 2016

Fig. 3: Summary of our multi-study investigation.

and the need to analyze at commit-level, it is not feasible to perform this manually. However, we relied on two popular build systems (*i.e.,* Ant and Maven) to automatically build the applications. The lack of the ability to build open-source software at commit-level is an inherent limitation to any similar study. We used the GitHub API to identify the project metadata that was used to select the candidate applications by extracting the language of the project, the stars, watchers, forks, and fork status (*i.e.,* if the project is a fork or not). We also utilized the Git command line utility to traverse the revision history so that we could analyze each commit. The use of the GitHub API and Git command line utility might affect the results.

External Validity. As we only consider Java GUI-based applications, we do not assert that our observations hold for all open-source systems or Java systems utilized through other mechanisms (*e.g.,* frameworks or libraries). Applications in other languages or hosted on other forges may exhibit different behaviors, but we focused on GitHub due to its overwhelming popularity and access to public repositories.

Conclusion Validity. To identify dead code, we utilized DUM-Tool [14], which was published in prior work and demonstrated to accurately identify dead methods in Java GUI-based applications.

Construct Validity. We used a single tool (*i.e.,* DUM-Tool) to detect dead code. In this context, there is a lack of tool support [8]. Additional threats could be related to the kind of dead code on which this motivational study is focused, *i.e.,* dead methods.

3.3 Wrap-up

The results from the mining study show that dead code is prevalent in the considered open-source Java applications. The presence of dead code could affect both comprehensibility and modifiability of source code (see the scenarios before presented). We also observed fluctuations in the amount of dead code through version history of some software projects. This might be related to the fact that developers perceive this smell as harmful and then remove it when possible. On the other hand, we observed that the amount of dead code remained constant through the version history of other projects, so letting us postulating that developers do not take care of this smell. These findings and postulations motivate the multi-study investigation into dead code presented in this paper.

4 Multi-Study Investigation

The overarching goal of our multi-study investigation is to answer the following Research Questions (RQs):

RQ1. *When and why do developers introduce dead code?*
RQ2. *How do developers perceive and cope with dead code?*
RQ3. *Is dead code harmful?*

RQ3.a. *Is dead code harmful to comprehend (unfamiliar) source code?*
RQ3.b. *Is dead code harmful to perform change tasks (familiar and unfamiliar) source code?*

In Figure 3, we graphically summarize our multi-study investigation.[9] In this figure, we depicted each individual study as a rectangle. Within every rectangle, we reported: the textual label (in bold) that we use to refer to the study in the rest of the paper (*e.g.,* Inter), the kind of study (*e.g.,* semi-structured interviews), and some information about the participants in the study (*e.g.,* six software professionals in Inter). We also provided a link between each individual study and the investigated RQ/s (see the circle at the top-left corner of any rectangle).

Our multi-study investigation took place between December 2015 and December 2016. We started this investigation with semi-structured interviews with six software professionals to study *when* and *why* dead code is introduced (*i.e.,* RQ1) and *how* developers perceive and cope with this smell (*i.e.,* RQ2). From January to December 2016, we conducted four controlled experiments with students to study *whether* dead code is harmful (*i.e.,* RQ3.a and RQ3.b). Indeed, in UniBas1—the baseline experiment—we investigated both RQ3.a and RQ3.b [20]. UniBas1 involved 47 undergraduate students at the University of Basilicata. The participants were asked to perform a comprehension task on a code base and then implement five change requests on that code base. In such a way, students continually increased their familiarity with the given code base, while performing the comprehension task. As a result, when the participants in UniBas1 had to implement the change requests (*i.e.,* the modification task), they had already investigated the greater part of the code base and largely increased their familiarity with this code. Therefore, we consider a code base *familiar* to a participant if she had already performed a comprehension task on that code base, while a code base is *unfamiliar* otherwise.

The results in UniBas1 suggested that comprehensibility of unfamiliar source code is significantly better when a code base does not contain dead code. On the other hand, we did not observe a statistically significant difference with respect to the modifiability (effort and effectiveness in modifying source code) of familiar source code.

9. The interested reader can find both the experimental materials and raw data at: www2.unibas.it/gscanniello/DeadCode/package.zip

In the replications, we focused on either RQ3.a (*i.e.,* UniBas2 and W&M1) or RQ3.b (W&M2). The participants in UniBas2 were graduate students at the University of Basilicata, while those in W&M1 and W&M2 were 11 and 6 undergraduate/graduate students at the College of William & Mary, respectively. The total number of students involved in these four experiments was 83.

UniBas2, W&M1, and W&M2 can be classified as operational replications because we varied some dimensions of experimental configuration [21]. For example, we varied the experimenters (*e.g.,* W&M1), the kind of participants (*e.g.,* UniBas2), and the experimental protocol (*e.g.,* experimental objects in W&M2). We introduced variations between the baseline experiment and the replications to increase our confidence in the results [22].

In the next sections, we present in detail the individual studies comprising our multi-study investigation.

5 Interviews: Planning and Execution

In this section, we present the design of the semi-structured interviews comprising our multi-study investigation.

5.1 Goal

Based on RQ1 and RQ2, we defined the main goal of the semi-structured interviews, by applying the Goal Question Metrics (GQM) template [23], as follows:

analyze	dead code
for the purpose of	understanding when and why it is introduced and how developers perceive and cope with it
from the point of view of	researchers and practitioners
in the context of	software professionals.

Although the use of GQM is uncommon in qualitative studies, we decided to exploit this template to ensure that important aspects were defined before the semi-structured interviews took place.

5.2 Participants

The participants (or also interviewees) in our semi-structured interviews were software professionals. We looked for potential participants within the contact network of the software engineering research group at the University of Basilicata. Six out of nine software professionals accepted our invitation. In Table 2, we show some information about the background of the participants who took part in our interviews. The interviewees had from four to ten years of work experience. Two out of six interviewees were project managers, whereas the others were software developers. All the interviewees had a degree in Computer Science: three had a doctoral degree, one a Master's degree, and two a Bachelor's degree. The companies in which the interviewees worked developed software in the following sectors: health, research, and Information and Communications Technology (ICT). Five out of six interviewees worked for independent private companies, while one interviewee was employed in a subsidiary public company. The workforce of these companies ranged from about 10 to 40,000 employees. The interviewees were employed in either multinational (four out of six interviewees) or Italian companies. The interviewees usually dealt with software systems whose estimated size, in terms of LOC (Lines of Code), ranged from medium to very large. Such a system-size estimate was based on the following classification: small if LOC $<$ 10,000, medium if 10,000 $\leq$ LOC $<$ 50,000, semi-large if 50,000 $\leq$ LOC $<$ 100,000, large if 100,000 $\leq$ LOC $<$ 500,000, and very large if LOC $\geq$ 500,000.

In Table 3, we report how each interviewee evaluated her experience in design, development, testing, maintenance, and management of software systems. Most of the interviewees had extensive experience in design, development, and maintenance of software systems. With respect to testing and management of software systems, two interviewees had extensive experience, while the others had some experience.

5.3 Procedure

The used procedure is inspired by the one Murphy *et al.* [24] and Francese *et al.* [25] used in their studies. The first author (from here onward referred to as interviewer) interviewed professionals in person, if they worked in the Potenza area, or via Skype otherwise. Each interview was audio-recorded and involved the interviewer and only one interviewee at a time. It was conducted in Italian, since it was the native language for both interviewer and interviewees (the use of English could introduce unnecessary complexity and threaten the results due to misunderstandings). Each interview lasted at most one hour and it was divided into four parts. In the first part, the interviewer gathered demographic information related to interviewee's work experience and summarized in Table 2 and Table 3. The second part consisted of an open-ended question, where the interviewee could freely talk about dead code according to her experience. The interviewer did not influence the answer of each interviewee. In the third and fourth part, the interviewer provided the interviewee with a list of software engineering topics. In the third part, the interviewee chose two of these topics and then he/she discussed dead code in relation to the chosen topics. In the fourth part, two topics, among those not yet discussed, were chosen by the interviewer; then, the interviewee discussed these topics in relation to dead code similarly to the third part of the interview. While choosing the topics from the list, the interviewer tried to ensure coverage of all the topics at least once. The interviewer chose topics with which the interviewee had more familiarity. This was done by using both demographic information and personal knowledge of the interviewee (if any). The interviewees could state that they were not familiar with a given topic. In such a case, the interviewer would choose another topic. Topics were discussed one at a time in both the third and fourth parts. To ensure that all the software engineering topics related to dead code were covered, we selected the following topics from the Software Engineering Body of Knowledge (SWEBOK) [26]: *(1) Software Design*, *(2) Software Construction*, *(3) Software Testing*, *(4) Software Maintenance*, *(5) Software Configuration Management*, *(6) Software Engineering Management*, *(7) Software Engineering Process*, and *(8) Software Engineering Tools and Methods*.

TABLE 2: Information on the interviewees and on the companies where they worked.

ID	Degree	Years of Experience	Position	Company				
				Sector	Ownership	Workforce	Location	Typical System-Size
I1	PhD	7	Developer	ICT	Independent, private	~300	Many, in Europe	Semi-large/large
I2	Master	10	Project Manager	ICT	Independent, private	~40,000	Many, in any continent	Large/very large
I3	Bachelor	4	Developer	ICT	Independent, private	~9,000	Many, in Europe, South and North America	Large/very large
I4	PhD	8	Developer	Health	Subsidiary, public	~1,900	Unique, in Italy	Medium/semi-large
I5	PhD	10	Developer	Research	Independent, private	~10	Unique, in Italy	Medium/semi-large
I6	Bachelor	6	Project Manager	ICT	Independent, private	~40,000	Many, in any continent	Large/very large

TABLE 3: Experience of the interviewees. Each of them could evaluate her experience as: none, some, or extensive.

ID	Design	Development	Testing	Maintenance	Management
I1	Some	Extensive	Some	Extensive	Some
I2	Extensive	Extensive	Some	Extensive	Extensive
I3	Some	Extensive	Some	Extensive	Some
I4	Extensive	Extensive	Extensive	Some	Some
I5	Extensive	Extensive	Extensive	Extensive	Some
I6	Extensive	Extensive	Some	Extensive	Extensive

5.4 Analysis Procedure

We first transcribed the audio-recordings of the interviews and then we (the first and the third authors of this paper) analyzed the transcriptions of these recordings by means of a Thematic Analysis Template (TAT) [27]. Thematic analysis is a qualitative method for identifying, analyzing, and reporting patterns (*i.e.,* themes) within data (*i.e.,* transcriptions in our case) [28]. A template is a hierarchical structure of themes. Researchers first build an initial template to analyze data; then, this template can be modified during the course of the ongoing analysis. As King [27] suggests, the best starting point to build an initial template is represented by the interview topics. Therefore, the themes of our initial templates were the selected SWEBOK topics. We exploited TAT because it is both fast to build and is also flexible [27].

To carry out our analysis, we exploited ATLAS.ti,[10] a program to support qualitative analysis. Thus, it is also suitable for supporting thematic analysis.

6 Interviews: Results

In this section, we present the results from the study of RQ1 and RQ2 and then the threats that could affect their validity.

6.1 Studying RQ1 and RQ2

In Figure 4, we show the graph-based visualization of the final template resulting from the thematic analysis. Themes and sub-themes, constituting the final template, are represented as rectangles while relations among them are depicted through arrow-shaped lines. When there is a relation between a theme and one or more sub-themes, they are drawn with the same background color. The visualization shown in Figure 4 was built in ATLAS.ti by means of the network view feature.

The final template is also shown in Table 4, where we assign a number/letter to each theme/sub-theme and we report how each identified sub-theme contributes to answer the research questions. In the following, we present the results with respect to the themes in our final template. We also provide some excerpts from the interviews.

10. atlasti.com

Theme 1. Software Design

a) The interviewees stated that dead code could result from a design choice to support anticipated future features. For example, I1 said:

   You could design a software component that is dead, because you believe to use it someday.

Theme 2. Software Construction

a) To support anticipated future features, the interviewees stated that developers write dead code that they think to revive someday. This is when code is created dead. For example, I1 said:

   Just yesterday, I modified a program to allow it to work on TVs manufactured by A[11] *in 2014 and 2015. When I was making my modifications, I thought that next year I'd modify my program again to allow it to work on A's 2016 TVs. So I wrote a piece of code that should allow the program to work on A's 2016 TVs. I know that, at the moment, this piece of code is dead, but it'll be used in 2016.*

b) The interviewees said that getter/setter methods are written without worrying about if these methods will be used or not. This is another case in which code was created as dead, but interviewees seem to tolerate its presence. On this point, I5 said:

   I usually keep dead getters/setters because they may come in handy in the future.

c) When the interviewees are doubtful about whether a code fragment is dead or not (*i.e.,* alive), they use the debugger to find if there is an execution that involves this fragment. However, the interviewees were conscious that the use of a debugger can only guarantee that code is alive, but not the contrary. With respect to this theme, I3 said:

   When debugging the program I can figure out if there is unused code.

Theme 3. Software Testing

a) To be sure that a piece of code is alive, test case execution traces are exploited. The interviewees recognized that these traces can involve code that is alive when testing the subject program, but dead when running

11. A refers to a TV manufacturer that was anonymized.

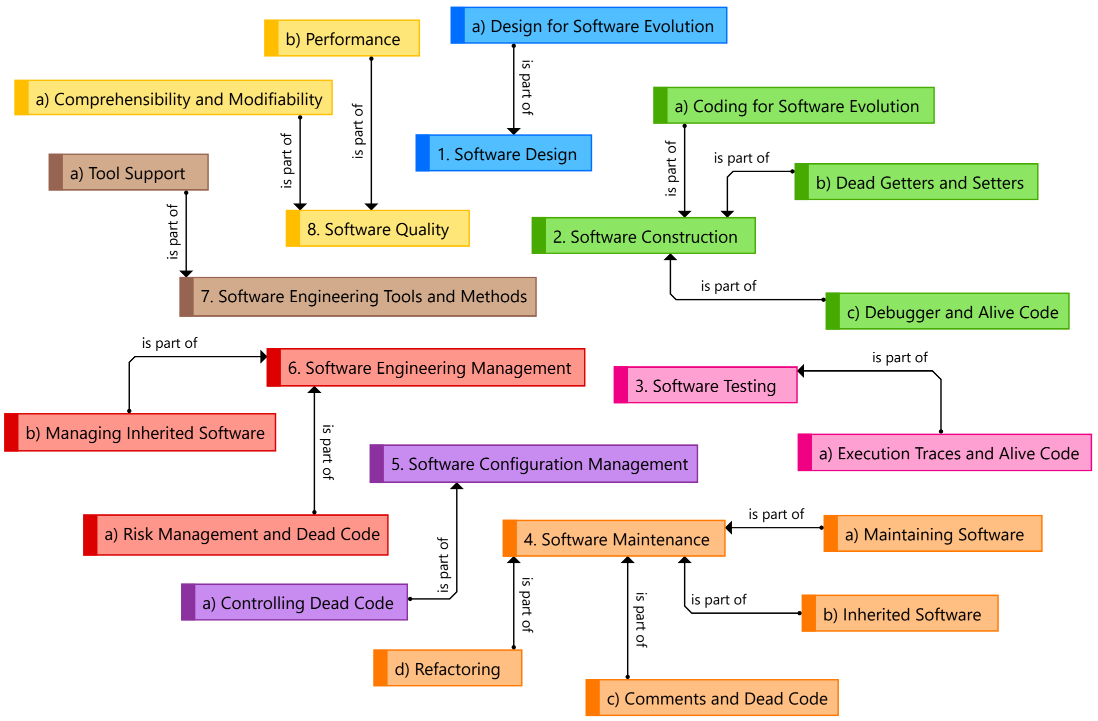

a) Design for Software Evolution
b) Performance
a) Comprehensibility and Modifiability
1. Software Design
a) Coding for Software Evolution
a) Tool Support
b) Dead Getters and Setters
8. Software Quality
2. Software Construction
7. Software Engineering Tools and Methods
c) Debugger and Alive Code
6. Software Engineering Management
3. Software Testing
b) Managing Inherited Software
5. Software Configuration Management
a) Execution Traces and Alive Code
a) Risk Management and Dead Code
4. Software Maintenance
a) Maintaining Software
a) Controlling Dead Code
b) Inherited Software
d) Refactoring
c) Comments and Dead Code
is part of

Fig. 4: Graph-based visualization of the final template built in ATLAS.ti.

TABLE 4: Final Template and links to the investigated RQs.

	Theme		Sub-theme	RQs
1.	Software Design	a)	Design for Software Evolution	RQ1, RQ2
2.	Software Construction	a)	Coding for Software Evolution	RQ1, RQ2
		b)	Dead Getters and Setters	RQ1, RQ2
		c)	Debugger and Alive Code	RQ2
3.	Software Testing	a)	Execution Traces and Alive Code	RQ2
4.	Software Maintenance	a)	Maintaining Software	RQ1, RQ2
		b)	Inherited Software	RQ1
		c)	Comments and Dead Code	RQ2
		d)	Refactoring	RQ2
5.	Software Configuration Management	a)	Controlling Dead Code	RQ2
6.	Software Engineering Management	a)	Risk Management and Dead Code	RQ2
		b)	Managing Inherited Software	RQ2
7.	Software Engineering Tools and Methods	a)	Tool Support	RQ2
8.	Software Quality	a)	Comprehensibility and Modifiability	RQ2
		b)	Performance	RQ2

the program within its target environment. The interviewees tolerate the presence of this kind of dead code, because it is exercised by at least one test case and it would not be a major issue if this code was inadvertently (or not) revived. On this point, I4 said:

The analysis of test case execution traces told you if a given method is executed. Thus, you can assume that it isn't dead. However, it can happen that a test case exercises a method that is currently not used, that is dead.

Theme 4. Software Maintenance

a) The interviewees asserted that dead code can be introduced during the execution of any maintenance operation (*i.e.*, adaptive, corrective, and perfective). Dead code is introduced because: *(i)* developers are unaware that a modification to a code fragment makes another code fragment dead (*e.g.*, if there is only a method m that invokes a method n and m is modified so that it does not invoke n anymore, then n becomes dead); and *(ii)* developers believe that a code fragment that after some changes is no longer used, could be reused someday (*i.e.*, they are aware that this code is dead, but it could be useful later). With respect to this sub-theme, I5 said:

When I develop a new version of a piece of source code, I often keep the old one for a certain time period although it's dead.

b) The interviewees stated that dead code could be inherited together with software changes. This happens when a software company acquires an existing software system from another company, and becomes responsible for its maintenance and evolution. For example, I2 said:

Dead code is often inherited. That is, we get it from another software company.

c) When dead code is identified, the interviewees do not always perform refactorings or do not adequately perform refactoring operations to remove it. Two strategies, both based on the use of source code comments, are applied: *(i)* adding a comment before dead code to take note that it is or could be dead and *(ii)* commenting out dead code. The interviewees (and developers, in general) go towards option *(i)* when they are aware that a code fragment is dead, but they do not have adequate time to remove it. This option is also used when the developer is unsure whether the removal of a code fragment (supposed to be dead) will affect the external behavior of a subject software system. Option *(ii)* is applied when interviewees are sure that a code fragment is dead, and they believe that they will utilize that code in the future (without possibly taking advantage from a version control system, see Theme 5). This is the case when dead code is not adequately refactored. Commenting out dead code causes the introduction of new smells (*i.e.,* commented out code [3]) in the source code. With respect to this sub-theme, I6:

To satisfy delivery times, we just comment code that's supposed to be dead. That is, we add a sort of label to identify this kind of code. Similarly, we just comment code that's supposed to be dead if we deal with inherited code because modifying it is risky.

d) The interviewees deal with refactoring as follows: *(i)* removing dead code when they are sure that it is actually dead and its removal is both low-risk and low-cost; *(ii)* ignoring chances for the removal of dead code because of high-risk; and *(iii)* postponing the removal of dead code because of high cost. In the cases *(ii)* and *(iii)*, comments can be used to inform developers about the presence of dead code (see Theme 4.c). On this sub-theme, I1 said:

Recently, we've passed from a version to another one of a program, its GUI's been changed a lot, the application logic as well, and much code's been became dead. This dead code's been there for a while until someone's found the time to remove it.

Theme 5. Software Configuration Management

a) To record the effect of the removal of dead code, the interviewees (and developers, in general) use version control systems. These tools "remember" dead code removal operations and, therefore, the developer can successively use the removed source code. For example, I6 said:

If someday I need removed dead code, then I ask the version control system to look for it.

Theme 6. Software Engineering Management

a) The interviewees reported that both no work activity is planned and no ad hoc tool is used for the purpose of both identifying and removing dead code. However, code inspection and analysis activities are planned and executed to identify and then remove bad smells (including dead code). For example, I2 stated:

We use a code analyzer. It measures cyclomatic complexity, identifies large classes, etc. However, it isn't specific for analyzing dead code.

b) When dealing with inherited software (see Theme 4.b), interviewees stated that project managers suggest that developers avoid removing dead code even if it is recognized in the code. This choice is inherently related to the risks of removing dead code. On this sub-theme, I2 said:

Generally speaking, dead code should be removed. However, suppose you have a piece of code that smells bad but works. Making modifications on this piece of code means taking a risk. If you introduce a bug, the customer can ask your explanations on why you've modified working code.

Theme 7. Software Engineering Tools and Methods

a) To support the identification and removal of dead code, the interviewees asserted that, besides tools to get execution traces, they only exploit canonical features provided by the used Integrated Development Environment (IDE) such as searching, debugging, etc. For example, I4 said:

I check if a method is dead by right-clicking on the method, and then selecting Find Usages.[12]

Theme 8. Software Quality

a) The interviewees agreed on the fact that the presence of dead code worsens both source code comprehensibility and modifiability. This is why such a kind of smell should be removed from the source code, but this is not always possible (as mentioned in Theme 4). For example, I3 said:

Dead code should be removed. It could get you confused. It makes source code bulky and little readable.

b) The interviewees also stated that dead code may negatively impact software performance (*e.g.,* execution time), but they remove it for the purpose of achieving better performance in a few cases. On this sub-theme, I5 said:

As for software performance, I remove dead code only if it strongly worsens execution time.

6.2 Threats to Validity

Despite our efforts to mitigate as many threats to validity as possible, some limitations have to be considered when

12. It is a feature of NetBeans IDE.

interpreting the results. For example, we applied triangulation[13] strategies to increase the credibility and validity of the results [29], [30].

Internal Validity. It is impossible to know whether interviewees answered truthfully. For example, scarce motivation could negatively affect observed insights. To mitigate this threat, we chose topics that were of interest to the interviewees, while they directly chose other topics for discussion (see Section 5.3). We sampled participants only from people in our contact network. It may be that different interviewees would differ in some way from those in our sample. We mitigated this point by applying data source triangulation.

External Validity. Outcomes might have limited generalizability, because we report the results of the interviews with six software professionals. Although this could represent a strong limitation for the results, Guest *et al.* [31] noted that data saturation[14] may be attained by as little as six interviews.

Construct Validity. This kind of threat to validity concerns the behavior of the interviewees and the interviewer. When people take part in a study, they might try to figure out what is the purpose and intended result of the study. In our case, interviewees are likely to base their behavior on their guesses about the study goal, either positively or negatively. As for the interviewer, he can unconsciously bias the results on the basis of what he expects from the study. We mitigate this bias, because more than one experimenter participated in the thematic analysis (investigation triangulation).

7 Experiments: Planning and Execution

To carry out the controlled experiments, we followed the recommendations by Juristo and Moreno [32], Kitchenham *et al.* [33], and Wohlin *et al.* [19].

7.1 Goal

On the basis of RQ3.a and RQ3.b, we defined the main goal of UniBas1, by applying the GQM template, as follows:

analyze	the presence of dead code
for the purpose of	evaluating its effect
with respect to	the comprehensibility of unfamiliar source code
and with respect to	the modifiability of familiar source code
from the point of view of	researchers and practitioners
in the context of	undergraduate students and object-oriented software implemented in Java.

In UniBas2 and W&M1, we asked participants to work on a Java application larger than that used in UniBas1. This allowed us to reduce possible external validity threats. Keeping this in mind, we had to decide to investigate on either source code comprehensibility or modifiability, because the investigation of both could introduce threats to internal validity (*e.g.,* fatigue). Therefore, we studied comprehensibility (RQ3.a) in UniBas2 and W&M1, while modifiability (RQ3.b) in W&M2. On the basis of the prior considerations, we defined the main goal of UniBas2 and W&M1 as follows:

analyze	the presence of dead code
for the purpose of	evaluating its effect
with respect to	the comprehensibility of unfamiliar source code
from the point of view of	researchers and practitioners
in the context of	undergraduate and graduate students, and object-oriented software implemented in Java.

The main goal of W&M2 was defined as follows:

analyze	the presence of dead code
for the purpose of	evaluating its effect
with respect to	the modifiability of unfamiliar source code
from the point of view of	researchers and practitioners
in the context of	undergraduate and graduate students, professionals, and object-oriented software implemented in Java.

As for comprehensibility, we focused on the following three constructs: *(i)* comprehension effort (time to complete a comprehension task); *(ii)* comprehension effectiveness (ability to perform a comprehension task); and *(iii)* comprehension efficiency (ability to effectively perform a comprehension task without wasting time). As for the modifiability, we focused on the following three constructs: *(i)* modification effort (time to complete a modification task); *(ii)* modification effectiveness (ability to perform a modification task); and *(iii)* modification efficiency (ability to effectively perform a modification task without wasting time).

7.2 Participants

The participants in each experiment had the following characteristics:

- **UniBas1.** The participants were 3rd-year undergraduate students in Computer Science at the University of Basilicata. The experiment was carried out as an optional laboratory exercise of a course on the Design and Implementation of Information Systems (DIIS). The participants had passed all the exams related to the following courses: Procedural Programming, Object-Oriented Programming I, and Databases. Thanks to these courses the participants had gained programming experience in Java and had sufficient level of technical maturity and knowledge of software design, development, and refactoring. The participants had also experience in performing maintenance operations on source code written by others. In particular, before the experiment took place, we asked the participants to perform homework on legacy code written in Java. They had to add some functionality using the Test-Driven-Development approach to real software they did not know.

13. Possible triangulation types are: data source triangulation; observer triangulation; and methodological triangulation (*e.g.,* [29]). In data source triangulation, data sources can vary based on the times the data were collected, the place, or setting and from whom the data were obtained. Investigator triangulation involves using more than one observer, interviewer, coder, or data analyst in the study, while methodological triangulation concerns the combination of quantitative and qualitative approaches.

14. When no new information or themes are observed in the data when adding new interviews.

TABLE 5: Information on the participants in W&M2.

ID	Participant Kind	Work Experience (Months)
P1	Graduate Student	-
P2	Graduate Student	-
P3	Graduate Student	36
P4	Undergraduate Student	-
P5	Undergraduate Student	3
P6	Graduate Student	24

- **UniBas2.** The participants were 2nd-year graduate students in Computer Engineering. The experiment was carried out as an optional laboratory exercise of the Software Engineering course. The participants had passed all the exams related to the following courses: Procedural Programming, DIIS, Object-Oriented Programming I and II, Web Programming, and Databases. In these courses, the participants had gained programming experience in Java. The participants had sufficient level of technical maturity and knowledge of software design, development, and refactoring.
- **W&M1.** The participants were both undergraduate and graduate students in Computer Science at the College of William & Mary. The solicitation was independent of a class but targeted Computer Science majors and graduated students, and it required that the participants had either prior Software Development (CS301) or Software Engineering (CS435) coursework. Thus, the students would have been exposed to refactoring and maintaining source code written by others and they would have received sufficient experience in both Java and Object-Oriented Programming Principles. The participants had sufficient level of technical maturity and knowledge of software design, development, testing, and refactoring.
- **W&M2.** The participants were both undergraduate and graduate students in Computer Science at the College of William & Mary. Three out of the six participants had prior industrial experience ranging from 3 to 36 months (see Table 5). Similar to W&M1, the participants were required to have either prior Software Development or Software Engineering coursework or prior industrial experience (when applicable). The completed coursework would ensure that the participants were familiar with Java as well as refactoring and maintaining source code written by others. The participants in W&M1 did not take part in W&M2.

The participants in all the experiments were informed that their grade on the course, in which the experiment was conducted, would not be affected by their performance in that experiment. To encourage participation in UniBas1 and UniBas2, we rewarded the participants with a bonus in their final mark, while in W&M1 and W&M2, we reimbursed the students for their participation ($30). It is worth mentioning that students in UniBas1 and UniBas2 could not be payed for their participation, because this is forbidden in Italy (while rewarding them with a bonus in their final mark is allowed). Participation was strictly voluntary and the students were not coerced to participate in any way. All these choices were made to have motivated participants. We also informed the participants that the collected data would have been used only for research purposes, treated confidentially, and shared anonymously.

7.3 Procedure

Before each experiment took place, all the participants had to fill out a pre-questionnaire. The gathered information on the participants allowed us to better characterize the context of each experiment (see Section 7.2).

The participants in UniBas1, UniBas2, and W&M1 attended an introduction lesson, just before the experiments took place, where we provided detailed instructions on the experimental procedure. The experimental tasks (see Section 7.5) of UniBas1, UniBas2, and W&M1 were carried out under controlled conditions in research laboratories at either the University of Basilicata or the College of William & Mary. We monitored the participants to avoid possible interactions. On the contrary, the participants in W&M2 were provided with detailed instructions on the experimental procedure via e-mail. Then, they had to carry out the experimental tasks in the environment they preferred and asked them to work individually. We did not suggest any strategy to perform the assigned modification task (*e.g.*, a participant could take a break whatever she wanted).

Independently from the experiment, the participants had to use the Eclipse[15] IDE. We opted for Eclipse, because the participants were familiar with this IDE. The participants could freely exploit any functionality (*e.g.*, debugger and text-based search) already present in the Eclipse version for Java developer, that is, they could not install any additional plug-in. This is to have the same development environment for all the participants. It is worth mentioning that no plug-in to detect dead methods or classes was present in the used Eclipse version and we did not inform the participants that they worked on source code deprived or not of this smell.

During the experiments, we allowed the participants to use the Internet, because actual developers usually exploit this medium as support for their daily work activities. In addition, the participants needed the Internet to fill out questionnaires/forms, since we used Google Forms to create them. We instructed the participants not to use the Internet to communicate to each another.

As for the comprehension task (in UniBas1, UniBas2, and W&M1), we defined a comprehension questionnaire comprised of five open-ended questions. Then, we asked the participants to follow these steps: *(i)* indicating their name and start-time (shown on the used PC); *(ii)* answering the questions in the comprehension questionnaire by exploiting any functionality of Eclipse already installed; and *(iii)* indicating the end-time (shown on the used PC).

As for the modification task in UniBas1, we defined a task comprised of five change requests and asked the participants to implement them. The steps to follow were the same as the comprehension task except for the step *(ii)*. In particular, in the step *(ii)*, the participants had to implement the five change requests we provided them by exploiting any functionality already present in Eclipse.

When performing the modification task in W&M2, each participant had to create a Git repository just before executing the modification task that, in this case, comprised

15. www.eclipse.org

four change requests. Once a change request was implemented, the participant had to commit the changes made and indicate in the commit message the change request the changes refer to. Similar to UniBas1, the participants had to use Eclipse.

We did not impose any time limits for the tasks in our experiments. In other words, the participants could take as long as needed to complete a given task.

7.4 Experimental Objects

The applications (*i.e.,* LaTazza, aTunes, and LaTeXDraw) used in the experiments are summarized in Table 6. In this table, we report the name and a brief description of each application, and the experiment where this application was used. For each application, we also report some descriptive statistics of both its original version and its version deprived of dead code: LOC (comments were not taken into account), the number of types (*e.g.,* classes), and the number of methods. Descriptive statistics were gathered by means of the CodePro AnalytiX tool.[16]

The used applications were implemented in Java. LaTazza is a small desktop application that supports the sale and the supply of small bags of beverages (*e.g.,* coffee). Its domain can be considered a good compromise between generality and industrial application. In addition, LaTazza has been already employed in a number of empirical studies (*e.g.,* [34], [35], [36]). As for aTunes and LaTeXDraw, the former is an open-source cross-platform media-player, while the latter is a graphical drawing editor. We opted for aTunes and LaTeXDraw, because they are both larger than LaTazza (*e.g.,* aTunes and LaTazza had 42,357 and 1,291 LOC, respectively) and unlike LaTazza they were evolving software systems. The choice of LaTazza, aTunes, and LaTeXDraw as experimental objects allowed us to reduce possible construct validity threats (*i.e.,* experimenter's expectancies/biases [19]), since none of the authors was involved in the implementation and maintenance of these applications.

To obtain a version deprived of dead code for each application, the first author of this paper identified dead code and performed refactorings to remove it from the original version of the considered application. He focused on dead methods and classes. In particular, he followed the steps below:

1) Inspired by Boomsma *et al.* [7] and Eder *et al.* [5], the application was instrumented. Namely, an instruction was added to every method in order to record which methods were executed.
2) He lunched the instrumented application, thus the executed methods were gathered. To cover the highest number of usage scenarios, and consequently as many methods as possible, he consulted the user manual of the application. He lunched the instrumented application on Windows, Linux, and Macintosh operating systems. This is because the execution of some methods could be trigged only in Macintosh operating systems rather than in Windows ones, for example. We assumed that executed methods were alive.
3) Each non-executed method was further analyzed to determine whether it was actually dead or not. To this end, he manually inspected each method with the support of some Eclipse functionality (*e.g.,* call hierarchy) and ad-hoc tools for dead code detection (*e.g.,* DUM-Tool [14]).
4) If the analysis revealed that a method was dead, he refactored the application (*e.g.,* dead method removal). Then, he tested the application to verify that the applied refactoring did not introduce bugs.
5) If the analysis revealed that a class was dead (*e.g.,* all its methods are dead), he behaved as in the previous step.

The execution of these steps produced versions of LaTazza, aTunes, and LaTeXDraw deprived of dead code (see some descriptive statistics in Table 6).

7.5 Tasks

Independently from the experiment, the comprehension and modification tasks were related to dead code that was present in the original version of the experimental objects. The participants, based on the experiment, had to perform different experimental tasks. In UniBas1 the tasks were:

1) **Comprehension task.** The participants were provided with the code base of LaTazza. Then, they had to answer a comprehension questionnaire, comprised of five open-ended questions, on the provided code base. The questions of the comprehension questionnaire were the same for any participant. We chose the questions on the basis of the recommendations by Sillito *et al.* [17]. Some questions required the identification and expansion of focus points. That is, participants had to find software entities (*e.g.,* classes and methods) related to a given question and then explore relationships (*e.g.,* references and calls) among software entities. Other questions were related to understanding concepts in the source code that involved multiple relationships and software entities. All the questions were formulated using a similar form/schema. A sample question (here translated into English from Italian), taken from the comprehension questionnaire of LaTazza, is: *"What are the kinds of users that can buy small bags of beverages?"*. The expected answers for this sample question were: *"Visitors"* and *"Employees."* When answering the questions, we did not impose any approach on the participants. For example, a participant could skip a question and then get it back.
2) **Post-comprehension task.** We asked the participants to fill out a post-questionnaire (see Table 7) consisting of multiple-choice questions. These questions were reported as statements that admitted answers according to a 5-point rating scale. An example is *"I had enough time to complete the task"*, which admitted as answers: (1) *"I totally agree"*; (2) *"I agree"*; (3) *"I neither agree nor disagree"*; (4) *"I disagree"*; and (5) *"I totally disagree"*. The goal of this post-questionnaire was to get feedback about participants' perceptions of the comprehension task execution.
3) **Modification task.** We provided the participants with five change requests to the code base of LaTazza. The change requests were the same for any participant. We defined the change requests to mimic an actual modification task on LaTazza. A sample change request we used for LaTazza is: *"Change the cost of the boxes of*

16. marketplace.eclipse.org/content/codepro-analytix

TABLE 6: Information on the applications used in the off-line experiments.

Experiment	Application	Application Description	Original Version			Version Deprived of Dead Code		
			LOC	# Types	# Methods	LOC	# Types	# Methods
UniBas1	LaTazza	It is a desktop application that helps a secretary to manage the sale and the supply of small bags of beverages (coffee, tea, lemon-tea, etc.) for the coffeemaker.	1,291	18	116	1,078	18	72
UniBas2 W&M1	aTunes (ver. 1.10.1)	It is an open-source cross-platform media-player. aTunes allows playing and managing audio files. It also provides support for on-line radios, podcasts, and CD ripping. See www.atunes.org.	42,357	778	4,067	40,483	768	3,780
W&M2	LaTeXDraw (ver. 2.0.8)	It is an open-source graphical drawing editor that allows translating draws into PSTricks code or PS/PDF pictures. See latexdraw.sourceforge.net.	65,320	252	3,130	63,263	252	2,875

TABLE 7: Questions used in the post-comprehension/post-modification tasks. The first question was not used in W&M2.

	Question
1.	I had enough time to complete the [comprehension/ modification] task.
2.	The goals of the [comprehension/modification] task were clear.
3.	The [questions/change requests] were clear.
4.	[Answering the questions/Implementing the change requests] was easy.
5.	I have found the [comprehension/modification] task useful.

small bags of beverages. The new cost must be 28 €." We did not impose any approach restriction to accomplish the modification task.

4) **Post-modification task.** We asked the participants to fill out a post-questionnaire (see Table 7). The questions in this questionnaire were similar to those used for the post-comprehension task and aimed to get feedback about participants' perceptions of the modification task execution.

As for UniBas2 and W&M1, the tasks were:

1) **Comprehension task.** The participants had to answer a comprehension questionnaire similar to that used in UniBas1, but the experimental object was aTunes. This comprehension questionnaire consisted of five open-ended questions.
2) **Post-comprehension task.** The participants had to fill out the same post-questionnaire as UniBas1.

Finally, the tasks in W&M2 were:

1) **Modification task.** The participants were asked to implement four change requests to the code base of LaTeXDraw. Similar to the modification task of UniBas1, the change requests were conceived to mimic an actual modification task, but its execution was different (*e.g.,* the use of Git as described in Section 7.3). This task on an actual open-source application like LaTeXDraw together with the use of Git should allow making it even more similar to the modification task that actual developers usually perform.
2) **Post-modification task.** The participants had to fill out a post-questionnaire similar to that used in the post-comprehension task of UniBas1 (see Table 7).

7.6 Independent and Dependent Variables

Independent variables. Participants provided with the original code base of LaTazza, aTunes, or LaTeXDraw comprised the *treatment group*, while those provided with the code base deprived of dead code comprised the *control group*. Therefore, **Method** is the main independent variable, also named main factor or main manipulated factor or main explanatory variable. This variable is nominal and assumes values: NoDC (*i.e.,* code base deprived of dead code) and DC (*i.e.,* original code base — with dead code).

When studying RQ3.a (see Section 7.9), we considered **Exp** as independent variable and analyzed the interaction variable **Method:Exp**. These variable are both nominal. The former represents the experiment and assumes the following values: UniBas1, UniBas2, and W&M1. The latter corresponds to the interaction between Method and Experiment (*e.g.,* UniBas1:NoDC is the value that represents the interaction between UniBas1 and NoDC).

Dependent variables. To measure comprehension effort, we used the following dependent variable:

- **CTime.** It is the time (in minutes) a participant spent to complete the comprehension task. The higher the value for CTime, the greater the comprehension effort is. Using time as effort approximation is customary in literature and it is compliant with the ISO/IEC 25000 standard [37] definition, where effort is the productive time associated with a specific project task.

To quantitatively assess the answers to the comprehension questionnaire and thus quantify the comprehension effectiveness construct, we used two dependent variables:

- **CF.** Let A_i^p be the set of answers that a participant p has provided to answer the question i. Let A_i^* the set of expected answers (*i.e.,* the oracle) for the question i. We quantified comprehension effectiveness by using an information retrieval-based approach [38]. In particular, for each question i we computed the precision and recall measures as follows:

$$CPrecision_i^p = \frac{|A_i^p \cap A_i^*|}{|A_i^p|} \tag{1}$$

$$CRecall_i^p = \frac{|A_i^p \cap A_i^*|}{|A_i^*|} \tag{2}$$

From a practical point of view, $CPrecision_i^p$ and $CRecall_i^p$ estimate, respectively, correctness and completenesses of the set of answers provided by the participant p to answer the question i. We computed the balanced F-measure between $CPrecision_i^p$ and $CRecall_i^p$. This allowed us to get a tradeoff between the precision and recall measures:

$$CF_i^p = \frac{2 \cdot CPrecision_i^p \cdot CRecall_i^p}{CPrecision_i^p + CRecall_i^p} \tag{3}$$

For each participant, we estimated comprehension effectiveness as the average of the F-measure values of all the questions in the comprehension questionnaire:

$$CF = \frac{\sum_{i=1}^{n} CF_i^p}{n} \tag{4}$$

where n is the number of questions (*i.e.,* 5 in any case). This (ratio) metric takes values in between 0 and 1. A value for CF close to 1 means that the comprehension effectiveness achieved by the participant is high, because she answered rather well to all the questions in the comprehension questionnaire. Conversely, a value for CF close to 0 means that the source code comprehension was bad. F-measure has been largely adopted in software engineering (*e.g.,* [39], [40]).

- **CAvg.** Let Q_i^p be a boolean variable defined for the participant p and the question i:

$$Q_i^p = \begin{cases} 1 & if\ A_i^p = A_i^* \\ 0 & otherwise \end{cases} \tag{5}$$

Q_i^p assumes 1 as the value if and only if the set of answers the participant p has provided to the question i (*i.e.,* A_i^p) is equal to the set of expected answers to i (*i.e.,* A_i^*). For each participant, the number of correctly and completely answers can be computed as follows:

$$CCnt = \sum_{i=1}^{n} Q_i^p \tag{6}$$

To measure comprehension effectiveness, we divided $CCnt$ by the number of answers in the comprehension questionnaire:

$$CAvg = \frac{CCnt}{n} \tag{7}$$

This metric assumes values in between 0 and 1. Higher the CAvg value, the better the comprehension effectiveness is. Unlike CF, CAvg does not take into account partial answers.

We used two dependent variables to estimate the comprehension efficiency construct:

- **CηF.** It is computed as follows:

$$C\eta F = \frac{CF}{CTime} \tag{8}$$

The larger the CηF value, the better the comprehension efficiency is.

- **CηCnt.** Differently from CηF, we estimated the comprehension effectiveness by means of CCnt:

$$C\eta Cnt = \frac{CCnt}{CTime} \tag{9}$$

The larger the CηCnt value, the better the comprehension efficiency is. This metric can be also seen as the number of both correct and complete answers per minute.

We used two measures for the effectiveness and efficiency constructs to reduce as much as possible construct validity threats.

To estimate the modification effort, we used:

- **MTime.** It is the time (expressed in minutes) a participant spent to execute a modification task. The higher the MTime value, the greater the effort to complete the modification task.

We quantified modification effectiveness by means of the following dependent variable:

- **MAvg.** Let R_i^p a boolean variable defined for the participant p and the change request i:

$$R_i^p = \begin{cases} 1 & if\ i\ is\ correctly\ implemented \\ 0 & otherwise \end{cases} \tag{10}$$

We defined an acceptance test suite for each change request. Therefore, the change request i (implemented by the participant p) is considered correctly implemented if and only if all the test cases of the corresponding acceptance test suite passed. It is worth mentioning that we did not provide the participants with any test suite. The number of change request correctly implemented by a participant can be computed as follows:

$$MCnt = \sum_{i=1}^{m} R_i^p \tag{11}$$

where m is the number of change requests in the modification task (*i.e.,* 5 for LaTazza in UniBas1 and 4 for LaTeXDraw in W&M2). To measure modification effectiveness, we used:

$$MAvg = \frac{\sum_{i=1}^{m} R_i^p}{m} \tag{12}$$

This metric assumes values in between 0 and 1. The higher the MAvg value, the better modification effectiveness is.

We estimated modification efficiency by means of the following dependent variable:

- **MηCnt.** It is computed as follows:

$$M\eta Cnt = \frac{MCnt}{MTime} \tag{13}$$

The larger the MηCnt value, the better modification efficiency is. This metric can be also interpreted as the number of change requests correctly implemented per minute.

7.7 Hypotheses Formulation

We formulated and tested eight null hypotheses. We described them by means of two parameterized null hypotheses with respect to RQ3.a and RQ3.b. As for RQ3.a, we formulated the following parameterized null hypothesis:

$\boldsymbol{HN_{CX}}$ **-** The presence of dead code in a code base does not significantly affect CX (*i.e.,* CTime, CF, CAvg, CηF, CηCnt).

To answer RQ3.b, we formulated the following parameterized null hypothesis:

$\boldsymbol{HN_{MX}}$ **-** The presence of dead code in a code base does not significantly affect MX (*i.e.,* MTime, MAvg, and MηCnt).

Alternative hypotheses for HN_{CX} and HN_{MX} ($\neg HN_{CX}$ and $\neg HN_{MX}$, respectively) admit a significant effect of Method on a given dependent variable. For instance, if the null hypothesis HN_{CTime} is rejected, the alternative one ($\neg HN_{CTime}$) is accepted, that is "the presence of dead code in a code base significantly affects CTime."

7.8 Design of the Experiments

In each experiment, we used one factor with two treatments design [19]. The design setup in each experiment uses the same experimental object (*e.g.,* LaTazza) for both of the treatments (*i.e.,* NoDC and DC). That is, each participant in a given experiment was asked to perform a task on only one experimental object and we administered only one treatment to each participant. The participants were randomly assigned to the treatments in each experiment.

In Table 8, we summarize the conducted experiments (*e.g.,* the number of participants in NoDC and DC). This table also makes clearer the differences among the experiments. It is worth mentioning that the participants from P1 to P3 in W&M2 (see Table 5) were in the NoDC group, while the others in the DC one.

7.9 Analysis Procedure

To perform the data analysis, we used the *R* environment.[17] The analysis procedure that we followed to investigate RQ3.a and RQ3.b follows.

RQ3.a. We computed descriptive statistics for each dependent variable concerning the investigation of RQ3.a (*i.e.,* CTime, CF, CAvg, CηF, and CηCnt). Then, we exploited graphical representations (*i.e.,* boxplots and bar charts) to summarize the values of these dependent variables and answers to the questionnaires of the post-comprehension tasks. To test the hypotheses concerning the effect of dead code on source code comprehensibility (*i.e.,* HN_{CTime}, HN_{CF}, HN_{CAvg}, $HN_{C\eta F}$, and $HN_{C\eta Cnt}$), we used multivariate linear mixed model analysis methods. This kind of model is an extension of the general linear model and it is a better method for analyzing models with random coefficients (as is the case of participants in software engineering experiments) and data dependency (as it is the case of our experiments) [41]. Thanks to the use of multivariate linear mixed model analysis methods, we verified the effect of Method, Exp, and their interaction.[18] To apply multivariate linear mixed model analysis methods, residuals have to follow a normal distribution with a mean equals to 0. If the residuals do not meet the condition of normality, transforming the response variable data is an option (*e.g.,* logarithmic transformations).

RQ3.b. Given the different objectives between UniBas1 and W&M2 (the former investigated the effect of dead code on the modifiability of familiar source code, while the latter focused on unfamiliar source code), we analyzed the data from these experiments separately. In UniBas1, we computed descriptive statistics for MTime, MAvg, and MηCnt. Then, we graphically summarized the values of these dependent variables by means of boxplots. We used bar charts to represent the answers to the questionnaire of post-comprehension task. To test HN_{MTime}, HN_{MAvg}, and $HN_{M\eta Cnt}$, we used an unpaired two-sided t-test, if the data were normally distributed. Otherwise, we used a two-sided Wilcoxon rank-sum test [42] (also known as Mann-Whitney U test). This test represents a non-parametric alternative to the unpaired two-sided t-test. As far as W&M2 is concerned, we present the values of the dependent variables (*i.e.,* MTime, MAvg, and MηCnt) for each participant. We also showed the participants' answers to the questionnaire of post-modification task. We did not run any statistical test, because we have few participants.

To verify the normality assumption of multivariate linear mixed model analysis methods and unpaired t-test, we used the Shapiro-Wilk W test [43] (Shapiro test, from here onwards). For each statistical test, we decided to accept (as customary) a probability of 5% of committing Type-I-error (*i.e.,* $\alpha = 0.05$).

8 Experiments: Results

In this section, we first present the results from the investigation of RQ3.a, then those concerning RQ3.b with respect to familiar and unfamiliar code. We conclude discussing threats that could affect the validity of these results.

8.1 Studying RQ3.a

Descriptive Statistics and Boxplots. In Table 9, we report the values of mean and standard deviation for the dependent variables CTime, CF, CAvg, CηF, and CηCnt, grouped by experiment and method. For these dependent variables, we also graphically summarized the distributions of the values by means of the boxplots reported in Figure 5.

The descriptive statistics and boxplots for CTime (*i.e.,* Figure 5.a) suggest that there is not a big difference between NoDC and DC in all the experiments together and considering these experiments individually. Indeed, participants in NoDC spent slightly more time to complete the comprehension tasks.

As for the variables used to quantify comprehension effectiveness (*i.e.,* CF and CAvg), there is a difference between NoDC and DC in favor of NoDC (see Table 9, and Figure 5.b

17. www.r-project.org/

18. For example, we built for $CTime$ the following linear mixed model with fixed effects equal to $CTime \sim method * exp$ and random effects equal to $\sim 1|ID$, where ID is the participant's identifier.

TABLE 8: Summary of the experiments.

Characteristic	UniBas1	UniBas2	W&M1	W&M2
Participants	47 Undergraduate Students in Computer Science	19 Graduate students in Computer Engineering	11 Under/graduate students in Computer Science	6 Under/graduate students in Computer Science (3 had work experience)
NoDC/DC Group Size	20/27	9/10	5/6	3/3
Experimental Object	LaTazza	aTunes	aTunes	LaTeXDraw
Investigated RQs	RQ3.a, RQ3.b	RQ3.a	RQ3.a	RQ3.b
Constructs (Dependent Variable/s)	Comprehension Effort (CTime) Comprehension Effectiveness (CF, CAvg) Comprehension Efficiency (CηF, CηCnt) Modification Effort (MTime) Modification Effectiveness ($MAvg$) (MAvg) Modification Efficiency (MηCnt)	Comprehension Effort (CTime) Comprehension Effectiveness (CF, CAvg) Comprehension Efficiency (CηF, CηCnt)	Comprehension Effort (CTime) Comprehension Effectiveness (CF, CAvg) Comprehension Efficiency (CηF, CηCnt)	Modification Effort (MTime) Modification Effectiveness (MAvg) Modification Efficiency (MηCnt)
Experimental Tasks	1) Comprehension 2) Post-comprehension 3) Modification 4) Post-modification	1) Comprehension 2) Post-comprehension	1) Comprehension 2) Post-comprehension	1) Modification 2) Post-modification
Experimenters	Researchers from the University of Basilicata	Researchers from the University of Basilicata	Researchers from the College of William and Mary	Researchers from the College of William and Mary

TABLE 9: Some descriptive statistics, grouped by experiment and method, for CTime, CF, CAvg, CηF, and CηCnt.

		Compr. Effort		Compr. Effectiveness				Compr. Efficiency			
Experiment	Method	CTime		CF		CAvg		CηF		CηCnt	
		Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
UniBas1	NoDC	35.05	6.9772	0.8467	0.0848	0.68	0.1196	0.0251	0.0055	0.1019	0.0307
	DC	33.7037	7.9895	0.7363	0.1265	0.5481	0.1718	0.0238	0.0104	0.0905	0.0529
UniBas2	NoDC	86.5556	15.1831	0.6593	0.1392	0.5111	0.1453	0.0078	0.0021	0.03	0.0087
	DC	82.7	13.6955	0.4073	0.1856	0.2	0.1333	0.0051	0.0027	0.0125	0.0089
W&M1	NoDC	56.4	23.5436	0.56	0.2966	0.48	0.2683	0.0108	0.0063	0.0478	0.0287
	DC	47.6667	13.3965	0.2944	0.2225	0.1333	0.2066	0.0073	0.0059	0.0161	0.025
ALL	NoDC	51.8235	25.5692	0.7549	0.1813	0.6059	0.174	0.0184	0.0095	0.0749	0.042
	DC	47.0465	22.7648	0.5981	0.2394	0.4093	0.2467	0.0171	0.0122	0.062	0.0569

TABLE 10: Results from the multivariate linear mixed model method for: HN_{CTime}, HN_{CF}, HN_{CAvg}, $HN_{C\eta F}$, and $HN_{C\eta Cnt}$.

Dependent variable	Method	Exp	Method:Exp
CTime	0.2519	**<0.0001**	0.7572
CF	**<0.0001**	**<0.0001**	0.1177
CAvg	**<0.0001**	**<0.0001**	**0.0034**
CηF	**0.0282**	**<0.0001**	0.9784
CηCnt	**0.0013**	**<0.0001**	0.5194

and Figure 5.c). This pattern holds when considering both the experiments individually and together.

For CηF and CAvg, the descriptive statistics and boxplots (*i.e.,* Figure 5.d and Figure 5.e) suggest that the NoDC values are slightly better than the DC ones in all the experiments.

Hypotheses Testing. In Table 10, we summarize the results of the multivariate linear mixed model method for: HN_{CTime}, HN_{CF}, HN_{CAvg}, $HN_{C\eta F}$, and $HN_{C\eta Cnt}$. For each built model, we report the p-values (in bold if smaller than α) for Method, Exp, and their interaction (*i.e.,* Method:Exp). In Table 11, we reported the Cliff's δ effect size[19] values and related Confidence Intervals (CI) if the effect of method is statistically significant. To give an indication about the magnitude of such an effect, we also provide a textual label that assumes the following values: negligible if $|\delta| < 0.147$, small if $0.147 \leq |\delta| < 0.33$, medium if $0.33 \leq |\delta| < 0.474$, or large otherwise [45].

One of the assumptions, while applying the multivariate linear mixed model method, was not satisfied for CTime. In particular, the residuals did not satisfy the normality assumption (the Shapiro test returned a p-value less than α). Therefore, we had to apply a square root transformation to meet the method assumptions. As shown in the second column of Table 10, we could not reject HN_{CTime} (*i.e.,* the p-value of the model was larger than α for CTime). The built model indicated a significant effect of Exp. This is because the participants in UniBas1 spent less time to perform the comprehension task (see Figure 5.a) than the participants in UniBas2 and W&M1. We did not observe any significant interaction between Method and Exp (*i.e.,* Method:Exp).

The results of the multivariate linear mixed model method indicated that there was a statistically significant

19. While a statistical test allows for checking the presence of significant differences, such a kind of test does not provide any information about the magnitude of these differences. There are a number of effect size measures to estimate these differences. The Cliff's δ effect size [44] is used in case data are not normally distributed or the normality assumption is discarded.

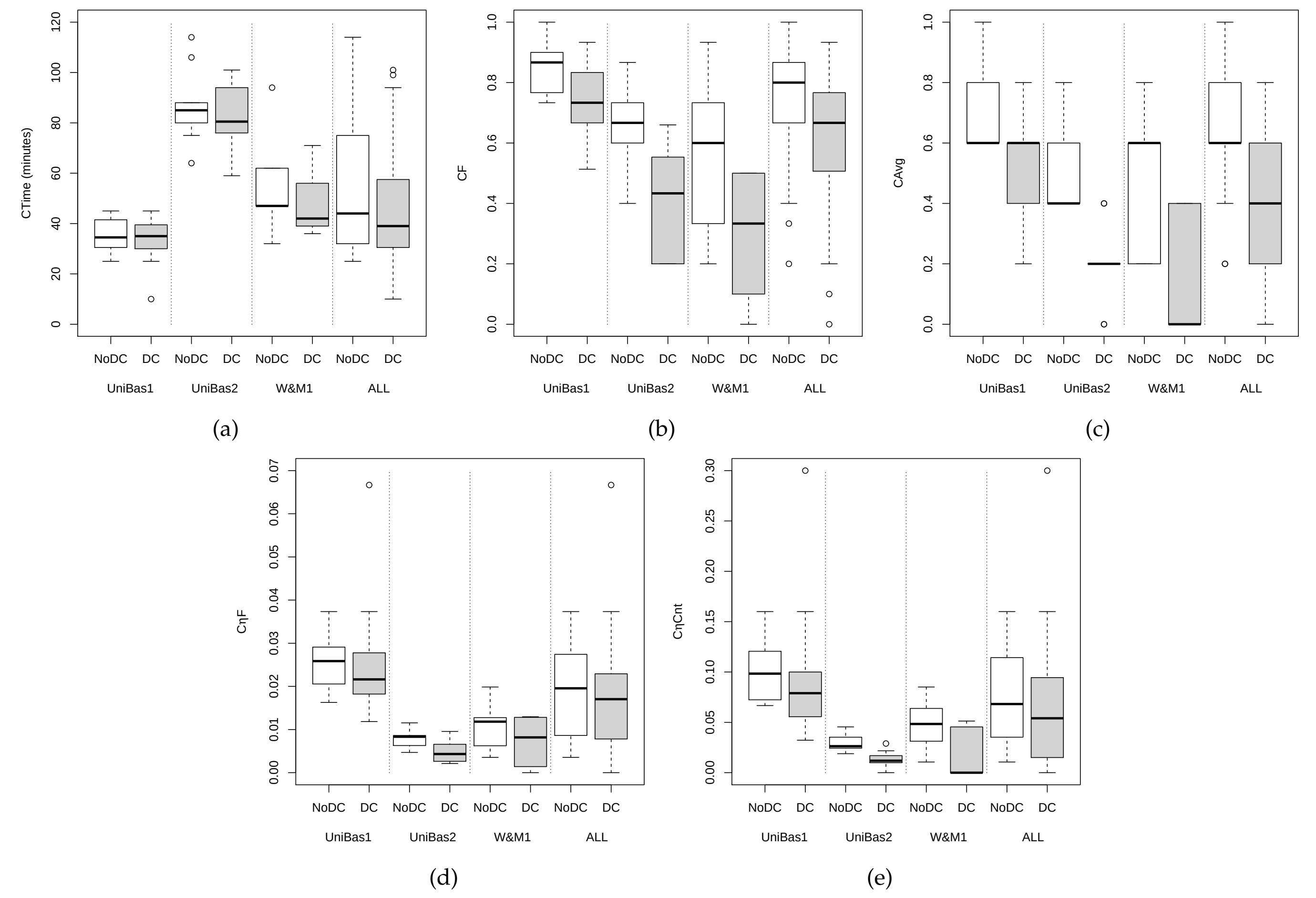


Fig. 5: Boxplots for: CTime (a), CF (b), CAvg (c), CηF (d), and CηCnt (e).

TABLE 11: Cliff's δ effect size values and the related confidence intervals (CI) for CTime, CF, CAvg, CηF, and CηCnt grouped by experiment.

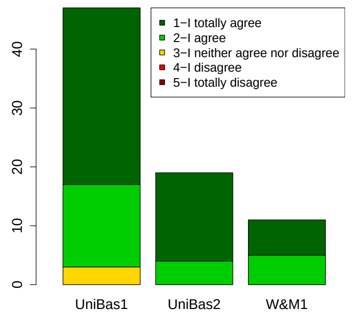
Experiment	Compr. Effort		Compr. Effectiveness				Compr. Efficiency			
	CTime		CF		CAvg		CηF		CηCnt	
	δ	CI	δ	CI	δ	CI	δ	CI	δ	CI
UniBas1	-	-	–0.5167 (large)	[-0.7357, -0.1998]	-0.4278 (medium)	[-0.6556, -0.1287]	-0.2481 (small)	[-0.5372, 0.093]	-0.2722 (small)	[-0.5537, 0.0651]
UniBas2	-	-	-0.7667 (large)	[-0.9673, 0.0234]	-0.8889 (large)	[-0.9728, -0.5992]	-0.4889 (large)	[-0.8137, 0.0685]	-0.8667 (large)	[-0.9744, -0.4352]
W&M1	-	-	-0.5667 (large)	[-0.9041, 0.2063]	-0.7333 (large)	[-0.971, 0.2341]	-0.2667 (small)	[-0.7442, 0.3913]	-0.6667 (large)	[-0.9428, 0.1518]

difference between DC and NoDC (the p-value for method was less than 0.0001) on CF. Therefore, we could reject HN_{CF} and accepted the alternative hypothesis. That is, the effect of dead code significantly penalizes comprehension effectiveness and the effect size is large in all the experiments (see Table 11). The built model also showed a significant effect of Exp. This is due to the better CF values the participants obtained in UniBas1 (see Figure 5.b). We did not observe any significant effect of Method:Exp.

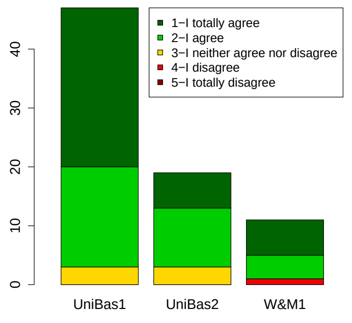
As for CAvg, we had to apply a logarithmic transformation because the normality assumption of the residuals was not satisfied (the p-value the Shapiro test returned was less than α). This model allowed us to reject HN_{CAvg} (the p-value for method was less than 0.0001). Thus, the effect of dead code significantly penalizes comprehension effectiveness (assessed by CAvg) and the effect size is either medium (in UniBas1) or large (in UniBas2 and W&M). The model also suggested significant effects of Exp and Method:Exp.

The residuals of the multivariate linear mixed model, which we built for the variable CηF, did not follow a normal distribution (the Shapiro test returned a p-value less than α). Then, we had to apply a rank transformation to satisfy the assumptions. The build model included two significant variables: Method and Exp. Therefore, we rejected $HN_{C\eta F}$ in favor of its alternative hypothesis. That is, the presence of dead code significantly penalizes comprehension efficiency (assessed by CηF) and the effect size is either small (in UniBas1 and W&M1) or large (in UniBas2).

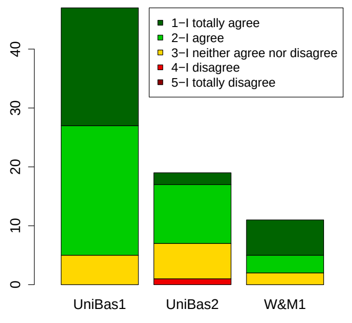
As for CηCnt, we had to apply a rank transformation because the residuals of the multivariate linear mixed model were not normally distributed (the Shapiro test returned a p-value less than α). The built model indicated a significant



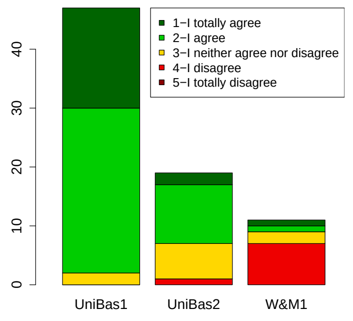
(a) Q1
I had enough time to complete the comprehension task.



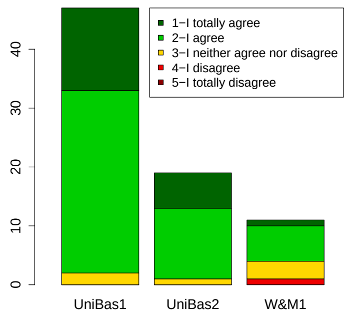
(b) Q2
The goals of the comprehension task were clear.



(c) Q3
The questions were clear.



(d) Q4
Answering the questions was easy.



(e) Q5
I have found the comprehension task useful.

Fig. 6: Stacked bar charts depicting the answers to the questionnaire of the post-comprehension task.

effect of Method and allowed us to reject $HN_{C\eta Cnt}$. That is, the participants in DC achieved CηCnt values significantly worse than the participants in NoDC. The value of the effect size was either small (in UniBas1) or large (in UniBas2 and W&M1). The variable Exp was significant as well.

Post-Comprehension Tasks. In Figure 6, we graphically summarize, by means of stacked bar charts, the answers the participants gave to the questionnaire of the post-comprehension task. In particular, each stack summarizes the answers to a question in a given experiment. The participants considered the given time to complete the comprehension task appropriate (see Figure 6.a). In all the experiments, most of the answers were *I totally agree* or *I agree*. Both the goals (see Figure 6.b) and questions (see Figure 6.c) of the comprehension task were considered clear. The greater part of the participants answered *I totally agree* or *I agree*. This pattern holds in all the experiments. As for Q4 (see Figure 6.d), the participants in UniBas1 and UniBas2 found the task easy (most of them answered *I totally agree* or *I agree*), while in W&M1 the greater part of the participants found it to be not easy (they answered *I disagree*). The participants in all the experiments found the task useful from a practical point of view (see Figure 6.e). Most of them answered *I totally agree* or *I agree*.

Further Analysis. We performed a further analysis to understand whether dead methods and dead classes are equally harmful. To this end, we analyzed the answers to each question individually. We computed the values of the dependent variables CF_i (see Equation 3) and Q_i (see Equation 5) for each question i. We took into account only the dependent variables CF_i and Q_i because we did not know the time each participant spent to answer a given question (*i.e.,* CTime), and thus we could not compute the values for CηF, and CηCnt per question. We tested the following null hypotheses: *(i)* the presence of **dead classes** in a code base does not significantly affect CF_i (or Q_i); and *(ii)* the presence of **dead methods** in a code base does not significantly affect CF_i (or Q_i). Comparing the results from the testing of the hypotheses *(i)* with the results from the testing of the hypotheses *(ii)* would allow understanding whether dead methods and dead classes are equally harmful. That is, if we would find that there is an effect of dead methods on CF_i and Q_i and, on the contrary, there is no effect of dead classes, we could say that dead classes are less harmful than dead methods. To test theses null hypotheses, we exploited permutation tests. This kind of tests can be applied to continuous, ordered, and categorical (*e.g.,* Q_i) data, and to normal and non-normal distributions [46]. Permutation tests can be also used for multivariate analyses [47]. The results from the hypotheses testing are summarized in Table 12.

In this further analysis, we did not take into account the data from UniBas1 because this experiment focused on dead methods only. The questionnaire for UniBas2 and W&M2 included one question (*i.e.,* the question number 3 in Table 12) concerning a dead class, while the other questions concerned dead methods. These results indicated that there is a statistically significant effect of dead methods on both CF_i and Q_i for three out of four questions. Indeed, the p-values for method are less than $\alpha = 0.05$ for the first, fourth,

TABLE 12: Results from the permutation tests concerning the dependent variables CF_i and Q_i.

Question	Dependent Variable	Method	Exp	Method:Exp
1	CF_i	**<0.0001**	0.9286	0.7677
	Q_i	**<0.0001**	0.9796	0.2994
2	CF_i	0.7547	0.6546	0.881
	Q_i	0.1175	0.4773	0.5915
3	CF_i	0.8401	0.6947	0.5694
	Q_i	0.7214	0.5412	0.4423
4	CF_i	**0.0362**	0.1371	0.5463
	Q_i	**0.0104**	0.2663	0.3198
5	CF_i	**0.0192**	0.3027	0.8547
	Q_i	**0.0255**	0.6249	0.5975

TABLE 13: Some descriptive statistics for MTime, MAvg, and MηCnt grouped by method.

Method	Mod. Effort MTime		Mod. Effectiveness MAvg		Mod. Efficiency MηCnt	
	Mean	SD	Mean	SD	Mean	SD
NoDC	52.95	8.153	0.54	0.2162	0.0541	0.0284
DC	51.6667	12.7189	0.5333	0.2353	0.0543	0.0306

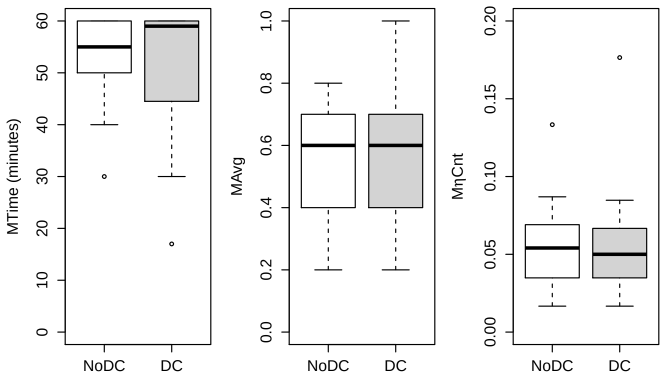


Fig. 7: Boxplots for UniBas1 with respect to MTime, MAvg, and MηCnt.

and fifth questions. As for the question number 3, the p-values for CF_i and Q_i did not allow us to reject the null hypotheses *(i)*. Although caution is needed, it seems that dead methods are more harmful than dead classes.

8.2 Studying RQ3.b - Familiar Source Code

Descriptive Statistics and Boxplots. In Table 13, we show the values of mean and standard deviation (grouped by method) obtained in UniBas1 (the only experiment where we investigated the effect of dead code on modifiability of familiar source code) for MTime, MAvg, and MηCnt. The boxplots for these dependent variables are shown in Figure 7. The descriptive statistics and boxplots for MTime indicate that there is not a big difference between NoDC and DC as well as for MAvg and MηCnt.

Hypotheses Testing. To test HN_{MTime}, we applied the Mann-Whitney U test because the MTime values (in UniBas1) did not follow a normal distribution (the Shapiro test returned a p-value less than α for both DC and NoDC). The Mann-Whitney U test was not able to reject HN_{MTime} (p-value was 0.5752).

Since the normality condition was not satisfied for MAvg (the Shapiro test returned a p-values less than α for both DC and NoDC), we applied the Mann-Whitney U test. According to the p-value returned by this test (0.8933), we could not reject the null hypothesis HN_{MAvg}.

As for the dependent variable MηCnt, the Shapiro test returned a p-value less than α for DC. Therefore, we applied the Mann-Whitney U test, which was not able to reject $HN_{M\eta Cnt}$ (p-value was 0.7298).

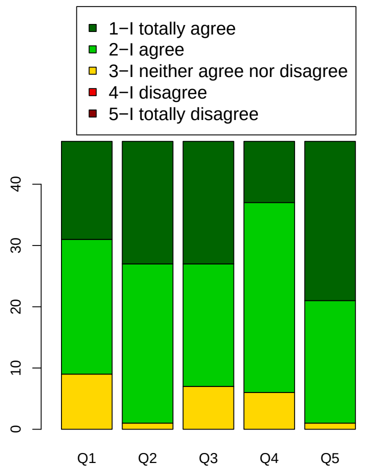


Fig. 8: Bar charts depicting the answers to the questionnaire of the post-modification task in UniBas1.

Post-Modification Task. In Figure 8, we report the staked bar charts that graphically summarize the participants' answers to the questionnaire of the post-modification task. In these charts, each stack represents the answers to a given question. The time needed to complete the modification task was considered appropriate (see Figure 8). Most of the participants answered *I totally agree* or *I agree* to Q1 (*I had enough time to complete the modification task*). The goals of the task were considered clear as well as the change requests. The greater parts of the answers to Q2 (*The goals of the modification task were clear*) and Q3 (*The change requests were clear*) were *I totally agree* or *I agree*. As for the difficulty and usefulness (from a practical point of view) of the modification task, the participants stated that the task was easy and useful. Most of the participants answered *I totally agree* or *I agree* to Q4 (*Implementing the change requests was easy*) and Q5 (*I have found the modification task useful*).

Further Analysis. We performed a qualitative analyses inspired by work of Robillard *et al.* [48]. That is, we evaluated the source code that a participant wrote to implement a change request by means of the following classification:

- **Success.** The participant provided a correct solution that respects the original design of LaTazza.
- **Inelegant.** The participant provided a correct solution that did not respect the design of LaTazza (*e.g.,* by hard-coding a value that should have been obtained from a property object);
- **Buggy.** The participant provided a generally workable solution that contains one or more bugs;
- **Unworkable.** The participant provided a solution that does not work in most cases;
- **Not attempted.** The participant did not provide any solution.

A change request can be also classified as **attempted** when a

TABLE 14: Count of the participants' solutions (in UniBas1) we classified as: success, inelegant, buggy, unworkable, not attempted, and attempted.

Classification	CR1		CR2		CR3		CR4		CR5		ALL	
	NoDC	DC	NoDC	DC	NoDC	DC	NoDC	DC	NoDC	DC	NoDC	DC
Success	13	18	16	25	4	5	5	4	15	17	53	69
Inelegant	0	0	0	0	0	0	0	0	1	3	1	3
Buggy	0	0	0	0	15	18	0	0	1	0	16	18
Unworkable	7	9	3	2	1	4	14	21	2	6	27	42
Not attempted	0	0	1	0	0	0	1	2	1	1	3	3
Attempted	20	27	19	27	20	27	19	25	19	26	97	132
Modification to dead code	-	0	-	0	-	11	-	0	-	3	-	14

TABLE 15: Percentage (approximated) values of the participants' solutions (classified as success, inelegant, buggy, and unworkable) with respect to the attempted solutions in UniBas1.

Classification	CR1		CR2		CR3		CR4		CR5		ALL	
	NoDC	DC	NoDC	DC	NoDC	DC	NoDC	DC	NoDC	DC	NoDC	DC
Success	65%	67%	84%	93%	20%	19%	26%	16%	79%	65%	55%	52%
Inelegant	0%	0%	0%	0%	0%	0%	0%	0%	5%	12%	1%	2%
Buggy	0%	0%	0%	0%	75%	67%	0%	0%	5%	0%	16%	14%
Unworkable	35%	33%	16%	7%	5%	15%	74%	84%	11%	23%	28%	32%

participant provided a solution for such a change request. That is, a change request is attempted if it is: success, inelegant, buggy, or unworkable.

We also inspected the source code to verify if the participants in the DC group modified existing dead code. This is to get indications on the possible negative effect of dead code while performing a modification task.

In Table 14, we report the count of the participants' solutions we classified as: success, inelegant, buggy, unworkable, not attempted, and attempted. In the last row of this table, we also show how many times the participants modified existing dead code when dealing with a given change request. To better compare the solutions of the participants in the NoDC group with those of the participants in the DC group, we report the participants' solutions in percentage values (with respect to the attempted solutions) in Table 15. Independently from the group, the participants' solutions to CR1 were mostly classified as success: 65% for the NoDC group and 67% for the DC one. The other solutions were unworkable. We did not observe a substantial difference between the solutions of the participants in the NoDC and DC groups. As for CR2, most of the participants' solutions were correct and respected the original design of LaTazza (*i.e.,* classified as success). We observed a slight difference between the solutions of the participants in the NoDC and DC groups (*e.g.,* the percentage values of the solutions classified as success were equal to 84% and 93% for the NoDC and DC groups, respectively). No participant modified existing dead code when dealing with CR1 or CR2. Most of the solutions to CR3 were correct or partially correct (*i.e.,* classified as success or buggy, respectively). In particular, we observed that for participants who worked on a code base deprived of dead code performed slightly better than those who dealt with dead code (75% vs. 67% of buggy solutions). We also observed 11 modifications to existing dead code when accomplishing CR3, of which three concerned unworkable solutions, five buggy solutions, and three success solutions. As for CR4, few solutions were correct and respected the original design of LaTazza. The results also indicate that the participants who tackled CR4 in the presence of dead code produced slightly worse solutions than the others (the percentage of solutions classified as success were equal to 26% for the NoDC group, while equal to 16% for the DC group). No participant modified existing dead code. The greater part of the solutions to CR5 were correct (*i.e.,* classified as either success or inelegant) or partially correct. We observed that the participants in the DC group performed slightly worse than those in the NoDC group (*e.g.,* the unworkable solutions were 23% for DC and 11% for NoDC). In addition, three participants in the DC group modified dead code when dealing with the change request. In two cases, the modifications concerned unworkable solution (the other case concerned a success solution). Overall, we observed that those participants who performed the modification task without dead code produced slightly better solutions than who carried out the same task exposed to dead code. Moreover, in 10 cases out of 14, where the participants modified existing dead code, we observed that they were not able to produce correct solutions to the change requests.

TABLE 16: Dependent variable values (*i.e.,* MTime, MAvg, and MηCnt), grouped by method, for each participant in W&M2.

Method	ID	(Mod. Effort) MTime	(Mod. Effectiveness) MAvg	(Mod. Efficiency) MηCnt
NoDC	P1	87	0.75	0.0345
	P2	113	0.25	0.0088
	P3	259	0.25	0.0039
DC	P4	231	0.25	0.0043
	P5	115	0.5	0.0174
	P6	199	0.25	0.005

8.3 Studying RQ3.b - Unfamiliar Source Code

Dependent Variable Values. In Table 16, we report the values of MTime, MAvg, and MηCnt for each participant in W&M2. The values for MTime suggest that participants working on source code deprived of dead code spent less time than those working on source code with dead code. The participants in NoDC spent 153 minutes on average to complete the modification task — P3 obtained the worst time (259 minutes), while P1 the best time (87 minutes). The participants in the DC group spent 182 minutes on average — P4 obtained the worst time (231 minutes), while P5 the best time (115 minutes).

As for MAvg, the results in Table 16 show that the MAvg values in the NoDC group were slightly better than those of the DC group (the mean values for the NoDC and DC groups are 0.4166 and 0.3333, respectively). The best value for $MAvg$ in the NoDC group is 0.75, while it is 0.5 in the DC one.

The MηCnt values suggest that the participants in the NoDC group were more efficient in performing the modification task with respects to those in the DC one. The mean values of MηCnt for the NoDC and DC groups were 0.0157 and 0.0089, respectively. The best value of MAvg in the NoDC group was 0.0345, while it was 0.0174 in the DC one. In each group, the worst values for this variable were very close, 0.0039 (NoDC) vs 0.0043 (DC).

Post-Modification Task. In Table 17, we report the answers each participant gave to the questionnaire of the post-modification task. The answers to Q1 (*The goals of the modification task were clear*) indicate that the participants, with the only exception of P6 (DC group), considered the goals of the modification task clear independently from his/her treatment (DC vs. NoDC). As for Q2 (*The change requests were clear*), all the participants in NoDC found the change requests clear, while two out of three participants in DC stated that the change requests were not clear. The participants in NoDC considered the change requests easy to implement, as suggested by their answers to Q3 (*Implementing the change requests was easy*). On the contrary, there was not unanimity in the answers that the participants in the DC group gave. Finally, the answers to Q4 (*I have found the task useful*) indicate that most of the participants, independently from the treatment, found the task useful. Summarizing, it seems that the overall perception on the modification tasks is more negative for the participants in the DC group.

Further Analysis. We performed the same further analysis as UniBas1 (see Section 8.2). The results from this further analysis are shown in Table 18. We also highlighted in bold if a participant (in the DC group) modified existing dead code when dealing with a change request.

As shown in Table 18, no participant was able to provide a correct solution for all the change requests. As for the NoDC group, the best participant was P1 (this is also the best participant in the study). He provided the correct solution that respected the original design of LaTeXDraw for all the change requests (*i.e.,* the implementations were classified as success) with the only exception of CR2 (no solution was provided, *i.e.,* his solution was considered as not attempted). The P2 and P3 participants performed similarly to one another. The solution for CR1 was workable, but it contained one or more bugs (*i.e.,* the solution was classified as buggy). As for CR4, P2 and P3 provided correct solutions that respected the original design of the experimental object (*i.e.,* the solutions were classified as success). They were unable to correctly implement the change requests CR2 and CR3, namely the solutions to these change requests were classified as either not attempted or unworkable (the solution did not work in most cases). As for the DC group, the best participant was P5, whose solution for CR1 was classified as success. The solution for CR2 was classified as inelegant (*i.e.,* this solution was correct, but it did not respect the design of LaTeXDraw). To be classified as success, this solution needed the initialization of a field of class; conversely, P5 implemented several changes in different parts of the source code of LaTeXDraw. As for CR3 and CR4, P5 provided source code classified as unworkable. It is worth noting that when dealing with CR3, the participant wasted his time modifying the body on an existing dead method. This was the only modification performed to solve this change request. The solution for CR1 was buggy for P4, while that for CR4 was classified as success. The other solutions of P4 were either not attempted or unworkable. As for P6, the source code he wrote was unworkable except for CR4 (its solution was classified as success). When dealing with CR1, this participant modified both alive and dead methods. In addition, P6 wasted his time modifying a dead method, while dealing with the solution of CR3. In this case, P6 modified only the existing dead code. This was why the solution provide for CR3 was classified as unworkable.

8.4 Threats to Validity

We report and discuss threats to validity by following the guidelines by Wohlin *et al.* [19].

Internal Validity. Fatigue effect might negatively affect the modification task in UniBas1 (*i.e., maturation*). To mitigate this effect, we used a small experimental object (*i.e.,* LaTazza) and we gave the participants a break before starting the modification task. As for the replications, the used experimental object (*i.e.,* aTunes and LaTeXDraw) were medium/semi-large. Therefore, we decided to investigate the effect of dead code on either source code comprehensibility or modifiability. This should allow mitigating fatigue effects. In W&M2, fatigue effects should be further mitigated since participants could execute the tasks when and whatever they want, so they could take a break in case they felt tired.

The effect of letting volunteers take part in UniBas1 and UniBas2 might influence the results, because volunteers are generally more motivated (*selection*) . In W&M1 and W&M1, the participants were remunerated. This might influence the results as well.

Instrumentation is the effect caused by the artifacts used for the experiment execution. For example, the version deprived of dead code of LaTazza, aTunes, and LaTeXDraw might affect the results as well as the experimental tasks.

Diffusion or imitation of treatments concerns the information exchanged among the participants. In W&M2, we did not have any means to prevent the participants from communicating to one another. However, we analyzed the source code the participants gave back to look for similarities among the solutions they wrote to implement the change requests.

External Validity. *Interaction of selection and treatment* concerns the involvement of students as participants since this might affect the generalizability of the results with respect to practitioners [49], [50]. However, the participants had knowledge of Java programming and software maintenance as it is the case of majority of young software professionals working in small-medium companies. With regard to the

TABLE 17: Answers to the questionnaire of the post-modification task (in W&M2).

Method	ID	Q1. The goals of the modification task were clear.	Q2. The change requests were clear.	Q3. Implementing the change requests was easy.	Q4. I have found the task useful.
NoDC	P1	I agree	I agree	I agree	I agree
	P2	I agree	I agree	I agree	I agree
	P3	I totally agree	I agree	I agree	I neither agree nor disagree
DC	P4	I agree	I totally disagree	I neither agree nor disagree	I agree
	P5	I totally agree	I agree	I agree	I agree
	P6	I disagree	I totally disagree	I totally disagree	I totally disagree

TABLE 18: Participants' solutions (in W&M2) classified as: success, inelegant, buggy, unworkable, and not attempted. In bold, the solutions modifying existing dead code.

Method	ID	Classification			
		CR1	CR2	CR3	CR4
NoDC	P1	Success	Not attempted	Success	Success
	P2	Buggy	Not attempted	Unworkable	Success
	P3	Buggy	Unworkable	Unworkable	Success
DC	P4	Buggy	Not attempted	Unworkable	Success
	P5	Success	Inelegant	**Unworkable**	Unworkable
	P6	**Unworkable**	Unworkable	**Unworkable**	Success

participants in UniBas2, they will soon be integrated into the software industry market, so they can be considered as representatives of young professionals [33]. Finally, three participants in W&M2 had some industrial experience.

Interaction of setting and treatment concerns the use of non-representative experimental setting or material. The chosen experimental objects might affect the validity of results. To mitigate this threat, we varied the experimental objects in the replications. Also the versions deprived of dead code might affect the validity of results.

Construct Validity. To mitigate threats to *confounding constructs and levels of constructs*, we randomly assigned participants to treatments.

Both the number of questions (*i.e.,* five) and of the change requests (*i.e.,* either five or four) was chosen to mitigate threats to *mono-operation bias*.

We mitigated threats to *mono-method bias* by considering different measures for comprehension effectiveness and efficiency. We used a single measure for the other constructs. This might introduce a measurement bias.

Knowing the experimental hypotheses might affect participants' behavior (*hypothesis guessing*). We did not inform the participants on the experimental hypotheses. In addition, each of them was assigned to only one treatment. Consequently, it should be difficult to guess the experimental hypotheses for the participants.

We mitigated threats to *evaluation apprehension* in two ways. We informed the participants that they would not have been evaluated/remunerated on the basis of the results they had achieved. They also knew that the achieved results would be shared anonymously.

Experimenters can bias results both consciously and unintentionally based on their expectancies. We mitigated this kind of threat (*experimenters' expectations*) in several ways. The application used in our experiments were downloaded from the web. The researchers who analyzed the artifacts produced by the participants were different from those who performed the data analysis.

Conclusion Validity. *Reliability of measures* concerns how dependent variables were measured. The methods used to quantify comprehension effectiveness and efficiency, and modification effectiveness and efficiency might affect the results. Regarding the comprehension and modification efforts, we asked the participants to write down their start- and end-times. This might affect the results.

There could be a risk that the experimental objects used in the treatment and control groups are different from one another (*reliability of treatment implementation*). Therefore, the observed differences in the results could be due not only because of the treatment but also because of the experimental objects. For example, the size of the applications used in the control groups was larger than that of the applications used in the treatment groups, because they were deprived of dead code.

There could be heterogeneity among participants within each experiment (*random heterogeneity of participants*). If participants are heterogeneous, results could be affected due to their individual differences. To deal with this threat, we drew fair samples and conducted our experiments with participants belonging to these samples.

9 Overall Discussion

In the rest of this section, we first discuss our findings by linking them to the investigated research questions, which are summarized in Table 19. Then, we delineate the practical implications of our findings together with future directions.

9.1 Linking Results and Research Questions

RQ1. *When and why do developers introduce dead code?*

We found that source code fragments can be either:

1) created dead;
2) made dead;
3) or inherited dead.

The case 1) can occur during software development: programmers are aware that they have just written dead code, but they plan to use it someday. The case 2) can happen during any software maintenance activity (*e.g.,* adaptive maintenance): developers can be either aware or not that their changes have made some code fragments dead. The case 3) occurs when developers inherit software systems with dead code from another company.

RQ2. *How do developers perceive and cope with dead code?*

Developers perceive dead code as harmful when comprehending and modifying source code. They also believe that dead code can negatively impact software performances. Nevertheless, it seems that developers consider dead code as means of anticipating changes and reusing code.

In the management of software projects, no specific work activity is planned to cope with dead code. There

TABLE 19: Summary of main findings grouped by research questions.

RQ	Finding		Software Engineering Topics
RQ1	F1	Source code was born dead	Software design and construction
	F2	Source code is made dead	Software maintenance
	F3	Source code is inherited dead	Software maintenance
RQ2	F4	Dead code is perceived to worsen comprehension and maintenance	Software quality
	F5	Dead code is believed to worsen software performances	Software quality
	F6	Dead code is considered as a means to anticipate changes and reuse code	Software design and construction
	F7	No specific activity is planned to cope with dead code	Software engineering management
	F8	There is a lack of tool support to cope with dead code	Software engineering management
	F9	Canonical IDE features and execution trace are exploited to identify dead code	Software testing and engineering management
	F10	Dead code removal is performed if low-risk and low-cost	Software maintenance and engineering management
	F11	Dead code removal is ignored if high-risk	Software maintenance and engineering management
	F12	Dead code removal is postponed if high-cost	Software maintenance and engineering management
	F13	Version control systems are used to keep track of dead code removal	Software maintenance and engineering management
	F14	Dead code is commented out to take note of its removal	Software maintenance and engineering management
	F15	Source code comments are used to warn developers about dead code presence	Software maintenance and engineering management
RQ3.a	F16	Dead code does not significantly affect effort to comprehend unfamiliar code	Software quality and maintenance
	F17	Dead code significantly worsens comprehension of unfamiliar code	Software quality and maintenance
	F18	Dead methods are more harmful than dead classes	Software maintenance
RQ3.b	F19	Dead code does not significantly affect modifiability of familiar code	Software quality and maintenance
	F20	Dead code seems to worsen modifiability of unfamiliar code	Software quality and maintenance
	F21	Developers unconsciously modify dead code	Software maintenance

is also a lack of tool support to identify and remove this kind of smell. Developers exploit only canonical features of IDEs (*e.g.,* debugging and call hierarchy) and analyze execution traces.

When a developer identifies dead code, its removal is:

1) performed if this operation is low-risk and low-cost;
2) ignored if it is high-risk;
3) postponed if it is high-cost.

When dead code is removed, developers exploit either version control systems to keep track of the performed changes or they comment out dead code. On the other hand, when the removal of dead code is ignored or postponed, developers use comments to warn other developers about the presence of dead code.

RQ3.a *Is dead code harmful to comprehend source code?*
Since we did not reject HN_{CTime}, we can postulate that the presence of dead code did not impair comprehension effort of unfamiliar source code. On the other hand, we could reject the null hypotheses on both comprehension effectiveness (*i.e.,* HN_{CF} and HN_{CAvg}) and efficiency (*i.e.,*, $HN_{C\eta F}$ and $HN_{C\eta Cnt}$): dead code has a significant negative effect on effectiveness and efficiency when comprehending unfamiliar source code. That is, dead code significantly worsens comprehension of unfamiliar code. The results of a further analysis suggest that dead methods are more harmful than dead classes. Based on the obtained results, we can positively answer RQ3.a although caution is needed and future work is advisable.

RQ3.b *Is dead code harmful to perform change tasks?*
We were not able to reject the defined null hypotheses in UniBas1 on modification effort (*i.e.,* HN_{MTime}), effectiveness (*i.e.,* HN_{MAvg}), and efficiency (*i.e.,* $HN_{M\eta Cnt}$). The results from W&M2 suggest that the participants dealing with dead code achieved on average worse values for modification effort, effectiveness, and efficiency than the other participants. Summarizing, the results seem to indicate that dead code is not harmful in cases where participants accomplished comprehension tasks and then modification tasks on the same code base, while dead code appears to be harmful in cases where the participants dealt with modification tasks without having accomplished a comprehension task on that code base before. We can speculate that this difference is due to the fact that in UniBas1 the participants were forced to inspect more parts of the source code in a more systematic and methodical fashion,[20] while in W&M2 the participants had possibly used a more opportunistic approach to implement the modifications given also the size of the software system used in the experiment [48]. We also found that some participants wasted time modifying existing dead code while implementing a change request and it seems that they were not aware of changing dead code. On the basis of our outcomes, we cannot provide a definitive answer to RQ3.b. However, our results justify future research (including replication studies) on this subject.

9.2 Implications

We highlight findings (using frames) focusing on possible implications from the *practitioner* and *researcher* perspectives.

Removing dead code improves (unfamiliar) source code comprehensibility and seems to better support (unfamiliar) source code modifiability (F8, F9, F17, F20, and F21).

Practitioner: Providing developers with source code deprived of dead code could reduce a number of issues related to well-known software engineering processes: maintenance, testing, quality assurance, reuse, and integration [51]. An effective tool support to detect and remove dead code is then advisable.

Researcher: She could be interested in defining approaches/tools for detecting and removing dead code since currently developers analyze execution traces and/or exploit canonical features of IDEs (*e.g.,* debugger).

20. The term *methodical* in its general sense indicates a behavior characterized by method and order. This term can be contrasted with *systematic* that refers to a line-by-line investigation of the entire source code. The term *opportunistic* is an antonym of methodical [48].

Every participant, regardless of their experience and background, achieved a better source code comprehension when dealing with source code without dead code (F17).

Practitioner: Developers with different profiles and skills can benefit from the removal of dead code.

Dead code is harmful, it is perceived as harmful, and it is consciously introduced to anticipate future changes or consciously left (F1, F2, F4, F6, F17, and F20).

Practitioner: She needs to be aware of the risks due to introducing or leaving dead code in the source code.
Researcher: She could be interested in defining approaches and tools to make dead code less harmful. For example, it would be advisable to define tools to hide dead code and to unhide it when need.

The removal of dead code is performed when it is both low-risk and low-cost (F10, F11, and F12).

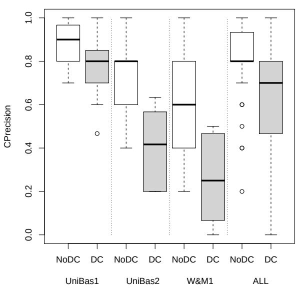
Researcher: She could be interested in defining approaches/tools to let the professionals know when the removal of dead code is low-risk and low-cost and whether the cost for this operation is adequately paid back by an improved source code comprehensibility and modifiability.

The familiarity of the participants with the codebase could affect source code modifiability (F19 and F20).

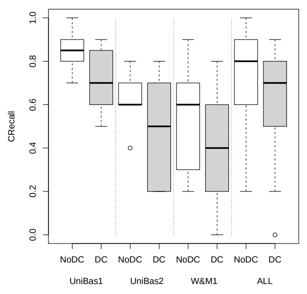
Researcher: A first improvement over the current design is to have treatment groups at different levels of familiarities with source code.

10 Conclusion

We first conducted a preliminary mining study to motivate the multi-study investigation presented in this paper. This multi-study investigation has been designed to answer the following questions: $when$ and why developers introduce dead code, how they perceive and cope with it, and $whether$ or not dead code is harmful. We first conducted semi-structured interviews with professional developers and then three off-line experiments and one on-line experiment with graduate and undergraduate students. The most important results are that *(i)* source code can be either created dead (*e.g.,* developers plan on using it someday), made dead during software evolution (*e.g.,* developers unintentionally make source code dead), or inherited dead (*e.g.,* a company performs maintenance on source code developed by others); *(ii)* developers perceive dead code as harmful when comprehending and modifying source code and this harmfulness is well-founded since the presence of dead code significantly penalizes the comprehension of unfamiliar source code, and it also negatively affects modification of unfamiliar source code; and *(iii)* when dead code is identified, refactorings are usually not performed to remove it (*e.g.,* developers add a comment before dead code to take a note that it is dead or could be potentially dead).



(a)



(b)

Fig. 9: Boxplots for CPrecision (a) and CRecall (b).

TABLE 20: Results from the multivariate linear mixed model methods with respect to CPrecision and CRecall.

Dependent Variable	Method	Exp	Method:Exp
CPrecision	**<0.0001**	**<0.0001**	**0.0053**
CRecall	**0.0006**	**<0.0001**	0.9629

Acknowledgments

The authors would like to thank all the software professionals and students from both continents who participated in our multi-study investigation.

Appendix A

In this appendix, we delineate whether the significant effect of dead code on CF (see Section 8.1) was due to little correctness or little completeness (or both) of the answers provided by the participants who comprehended source code in the presence of dead code. To this end, we estimated correctness and completeness of participants' answers through the dependent variables **CPrecision** and **CRecall**, respectively. We computed these variable as follows:

$$CPrecision = \frac{\sum_{i=1}^{n} CPrecision_i^p}{n} \tag{14}$$

$$CRecall = \frac{\sum_{i=1}^{n} CRecall_i^p}{n} \tag{15}$$

where n is the number of questions in the comprehension questionnaire, while $CPrecision_i^p$ and $CRecall_i^p$ are the precision and recall measures defined for the participant p and the question i (see Equation 1 and Equation 2).

In Figure 9, we graphically summarize, by means of boxplots, the distributions for CPrecision and CRecall. The boxplots in Figure 9.a suggest that the CPrecision values achieved by the participants in the NoDC group were better than those achieved by the participants in the DC group. We observed a similar trend, but less pronounced, for the dependent variable CRecall (see Figure 9.b).

To understand whether the differences observed in the boxplots were statistically significant, we exploited multivariate linear mixed model methods. We built a multivariate linear mixed model for CPrecision by using the following independent variables: Method, Exp, and Method:Exp. Similarly, we built a model for the CRecall. The results from the multivariate linear mixed model methods are presented in Table 20 (p-values less than $\alpha = 0.05$ are reported in bold). As for CPrecision, the built model indicated a statistically significant effect of Method as well as of Exp and Method:Exp. With respect to CRecall, the built model included two significant variables: Method and Exp. Based on these results, we can conclude that the presence of dead code significantly affects both correctness and completeness of the answers provided by the participants who comprehended source code in the presence of dead code.

References

[1] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proc. of International Conference on Software Maintenance*. IEEE CS Press, 2003, pp. 381–384.

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

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

[4] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *Proc. of International Working Conference on Source Code Analysis and Manipulation*. IEEE CS Press, 2013, pp. 116–125.

[5] S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K. H. Prommer, "How much does unused code matter for maintenance?" in *Proc. of International Conference on Software Engineering*. IEEE CS Press, 2012, pp. 1102–1111.

[6] 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.

[7] 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.

[8] S. Romano, G. Scanniello, C. Sartiani, and M. Risi, "A graph-based approach to detect unreachable methods in java software," in *Proceedings of the Symposium on Applied Computing*. New York, NY, USA: ACM, 2016, pp. 1538–1541.

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

[10] 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 *Proc. of the International Conference on Software Engineering*. IEEE CS Press, 2015, pp. 403–414.

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

[12] 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.

[13] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proc. of 8th International Workshop on Program Comprehension*, 2000, pp. 241–247.

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

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

[16] ——, "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 CS Press, 2014, pp. 392–397.

[17] 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.

[18] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, pp. e1838–n/a, 2017, e1838 smr.1838. [Online]. Available: http://dx.doi.org/10.1002/smr.1838

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

[20] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, "Are unreachable methods harmful? results from a controlled experiment," in *Proc. of International Conference on Program Comprehension*. IEEE CS Press, 2016, pp. 1–10.

[21] O. S. Gómez, N. J. Juzgado, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information & Software Technology*, vol. 56, no. 8, pp. 1033–1048, 2014.

[22] J. C. Carver, N. Juristo, M. T. Baldassarre, and S. Vegas, "Replications of software engineering experiments," *Empirical Software Engineering*, vol. 19, no. 2, pp. 267–276, Apr 2014.

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

[24] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in *Proc. of the International Conference on Software Engineering*. ACM, 2014.

[25] R. Francese, C. Gravino, M. Risi, G. Tortora, and G. Scanniello, "Mobile app development and management: Results from a qualitative investigation," in *Proc. of the International Conference on Mobile Software Engineering and Systems*, 2017.

[26] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, Eds., *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2001.

[27] N. King, "Using templates in the thematic analysis of text," in *Essential Guide to Qualitative Methods in Organizational Research*, C. Cassell and G. Symon, Eds. Sage, 2004, pp. 256–270.

[28] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[29] R. Stake, *The Art of Case Study Research*. SAGE Publications, 1995.

[30] V. A. Thurmond, "The point of triangulation," *Journal of Nursing Scholarship*, vol. 33, no. 3, pp. 253–258, 2001.

[31] G. Guest, A. Bunce, and L. Johnson, "How many interviews are enough?: An experiment with data saturation and variability," *Field Methods*, vol. 18, no. 1, p. 59, 2006.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

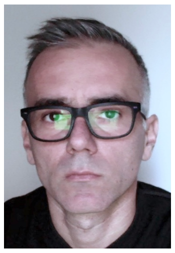
[39] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, "How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments," *IEEE Trans. on Softw. Eng.*, vol. 36, no. 1, pp. 96–118, 2010.

[40] G. Scanniello, C. Gravino, M. Genero, J. Cruz-Lemus, and G. Tortora, "On the impact of uml analysis models on source-code comprehensibility and modifiability," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 2, 2014.

[41] S. Vegas, C. Apa, and N. Juristo, "Crossover designs in software engineering experiments: Benefits and perils," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 120–135, Feb 2016.
[42] W. J. Conover, *Practical Nonparametric Statistics*. Wiley, 3rd Edition, 1998.
[43] S. Shapiro and M. Wilk, "An analysis of variance test for normality," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 1965.
[44] N. Cliff, *Ordinal methods for behavioral data analysis*. New-York, USA: Psychology Press, Sep. 1996.
[45] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?" in *annual meeting of the Florida Association of Institutional Research, February*, 2006, pp. 1–3.
[46] P. Good, *Permutation Tests: A Practical Guide to Resampling Methods for Testing Hypotheses*. Springer, 2000.
[47] M. J. Anderson, "A new method for non-parametric multivariate analysis of variance." *Austral Ecology*, vol. 26, pp. 32–46, 2001.
[48] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Trans. on Soft. Eng.*, vol. 30, no. 12, pp. 889–903, 2004.
[49] M. Ciolkowski, D. Muthig, and J. Rech, "Using academic courses for empirical validation of software development processes," *EUROMICRO Conference*, pp. 354–361, 2004.
[50] 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.
[51] G. Canfora and M. Di Penta, "New frontiers of reverse engineering," in *Workshop on the Future of Software Engineering*, 2007, pp. 326–341.

Simone Romano received his masters degree in Computer Engineering from the University of Basilicata, Italy, in October 2014. Currently, he is a PhD student in Computer Science at the University of Basilicata (in collaboration with the University of Salento) under the supervision of Prof. Giuseppe Scanniello. He is a member of the BASELab research group at the University of Basilicata. His research interests include empirical software engineering, reverse engineering, software maintenance and testing, peopleware, and agile software development methodologies. He is a member of IEEE. More information at: http://www2.unibas.it/sromano/

Christopher Vendome is a Ph.D. student at The College of William & Mary. He is a member of the SEMERU Research Group and is advised by Dr. Denys Poshyvanyk. He received a B.S. in Computer Science from Emory University in 2012 and he received his M.S. in Computer Science from The College of William & Mary in 2014. His main research areas are software maintenance and evolution, mining software repositories, program comprehension, software provenance, and software licensing. He is a member of ACM, IEEE, and IEEE Computer Society. More information at: http://www.cs.wm.edu/~cvendome/.



Giuseppe Scanniello received his Laurea and Ph.D. degrees, both in Computer Science, from the University of Salerno, Italy, in 2001 and 2003, respectively. In 2006, he joined, as an Assistant Professor, the Department of Mathematics and Computer Science at the University of Basilicata, Potenza, Italy. In 2015, he became an Associate Professor at the same university. His research interests include requirements engineering, empirical software engineering, reverse engineering, reengineering, software visualization, workflow automation, migration, wrapping, integration, testing, green software engineering, global software engineering, cooperative supports for software engineering, visual languages and e-learning. He has published more than 160 referred papers in journals, books, and conference proceedings. He serves on the organizing of major international conferences (as general chair, program co-chair, proceedings chair, and member of the program committee) and workshops in the field of software engineering (e.g., ICSE, ASE, ICSME, ICPC, SANER, and many others). Giuseppe Scanniello leads both the group and the laboratory of software engineering at the University of Basilicata (BASELab). He recently obtained the Italian National Scientific Qualification as Full Professor in Computer Science. He is a member of IEEE and IEEE Computer Society. More on http://www2.unibas.it/gscanniello/.

Denys Poshyvanyk is the Class of 1953 Term Distinguished Associate Professor of Computer Science at the College of William and Mary in Virginia. He received the MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine, and Wayne State University in 2003 and 2006, respectively. He received the PhD degree in Computer Science from Wayne State University in 2008. He served as a program co-chair for ICSME'16, ICPC'13, WCRE'12 and WCRE'11. He currently serves on the editorial board of IEEE Transactions on Software Engineering (TSE), Empirical Software Engineering Journal (EMSE, Springer) and Journal of Software: Evolution and Process (JSEP, Wiley). His research interests include software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. His research papers received several Best Paper Awards at ICPC'06, ICPC'07, ICSM'10, SCAM'10, ICSM'13 and ACM SIGSOFT Distinguished Paper Awards at ASE'13, ICSE'15, ESEC/FSE'15, ICPC'16 and ASE'17. He also received the Most Influential Paper Awards at ICSME'16 and ICPC'17. He is a recipient of the NSF CAREER award (2013). He is a member of the IEEE and ACM. More information available at: http://www.cs.wm.edu/ denys/.