Documenting Design-Pattern Instances: A Family of Experiments on Source-Code Comprehensibility

GIUSEPPE SCANNIELLO, University of Basilicata
CARMINE GRAVINO, MICHELE RISI, and GENOVEFFA TORTORA, University of Salerno
GABRIELLA DODERO, Free University of Bozen-Bolzano

Design patterns are recognized as a means to improve software maintenance by furnishing an explicit specification of class and object interactions and their underlying intent [Gamma et al. 1995]. Only a few empirical investigations have been conducted to assess whether the kind of documentation for design patterns implemented in source code affects its comprehensibility. To investigate this aspect, we conducted a family of four controlled experiments with 88 participants having different experience (i.e., professionals and Bachelor, Master, and PhD students). In each experiment, the participants were divided into three groups and asked to comprehend a nontrivial chunk of an open-source software system. Depending on the group, each participant was, or was not, provided with graphical or textual representations of the design patterns implemented within the source code. We graphically documented design-pattern instances with UML class diagrams. Textually documented instances are directly reported source code as comments. Our results indicate that documenting design-pattern instances yields an improvement in correctness of understanding source code for those participants with an adequate level of experience.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General; D.2.7 [**Software Engineering**]: Maintenance

General Terms: Design, Documentation, Human Factors

Additional Key Words and Phrases: Design patterns, controlled experiment, maintenance, replications, software models, source-code comprehension

ACM Reference Format:
Giuseppe Scanniello, Carmine Gravino, Michele Risi, Genoveffa Tortora, and Gabriella Dodero. 2015. Documenting design-pattern instances: A family of experiments on source-code comprehensibility. ACM Trans. Softw. Eng. Methodol. 24, 3, Article 14 (May 2015), 35 pages.
DOI: http://dx.doi.org/10.1145/2699696

1. INTRODUCTION

In the context of building and town design, Alexander et al. [1977] asserted that a "pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." This definition of pattern also holds for the design and development of object-oriented software [Gamma et al. 1995]. In such a context, a design pattern includes a name, an intent, a problem, its solution, some examples, and more. The core of both kinds

Authors' addresses: G. Scanniello (corresponding author), DiMIE, University of Basilicata, Viale Dell' Ateneo n. 10, Macchia Romana 85100, Potenza, Italy; email: giuseppe.scanniello@unibas.it; C. Gravino and M. Risi, Department of Management and Information Technology, University of Salerno, Via Giovanni Paolo II, nr. 132, 84084 Fisciano (SA), Italy; G. Tortora, Dipartimento di Matematica e Information, University of Salerno, Via Ponte Don Melillo, 84084 Fisciano (SA), Italy; G. Dodero, Computer Science Department, Room POS 218, Free University of Bozen-Bolzano, Dominikanerplatz 3-Piazza Domenicani, 3 39100 Bozen-Bolzano, Italy.

DOI: http://dx.doi.org/10.1145/2699696

of design pattern definitions concerns the solution to a problem in a given context. In software development, such a solution is named design-pattern instance or also design motif [Guéhéneuc and Antoniol 2008].

Gamma et al. [1995] assert that developers would benefit from the documentation of design patterns to comprehend source code, making its modification and evolution an easier task. Although there are a number of empirical investigations on design patterns (e.g., Prechelt et al. [2001], Bieman et al. [2003], Cepeda Porras and Guéhéneuc [2010], Jeanmart et al. [2009], Khomh and Guéhéneuc [2008], di Penta et al. [2008], Vokác [2004], Vokác et al. [2004], and Krein et al. [2011]), only a few evaluations have been conducted to study this conjecture, namely there is a lack of investigations to assess the benefit of documenting design-pattern instances for source-code comprehensibility in the context of software maintenance and evolution [Krein et al. 2011; Gravino et al. 2011, 2012; Prechelt et al. 2002].

To obtain an initial insight into the usefulness of the documentation of design-pattern instances (or simply pattern instances or instances, from here on), some of the authors (the first four) of this article carried out two experiments as a pilot study [Gravino et al. 2011]. These experiments aimed to assess the benefit of documenting instances with respect to not documenting them at all. The first experiment studied instances graphically documented by using the class diagrams of the *unified modeling language* (UML) [OMG 2005], while instances textually documented were considered in the second experiment. These results provided evidence that documenting pattern instances improves source-code comprehensibility with respect to not documenting them at all. This effect is even clearer in the case of graphically documented instances. To further investigate the effect of the kind of design-pattern instance documentation on source-code comprehensibility and to study any possible effect of the developers' experience [ISO 1991], we have performed a series of empirical investigations as a family[1] of controlled experiments. This family includes four experiments with 88 participants having different levels of experience (i.e., professionals and Bachelor, Master, and PhD students). The original experiment was carried out with professional developers and preliminarily findings are shown in Gravino et al. [2012]. The results of this experiment suggest that professionals provided with graphically and textually documented pattern instances achieved a significantly better comprehension of source code than those provided with source code alone. Therefore, we provide here the following new contributions: *(i)* three new experiments have been added; *(ii)* the analysis procedure of the already published experiment has been updated; *(iii)* the effect of the participants' experience has been analyzed; *(iv)* a more thorough discussion of the obtained results is reported; and *(v)* practical implications for the results are presented and discussed. Our family of experiments does not include the pilot study mentioned earlier [Gravino et al. 2011].

The experiments in our family were grounded on theoretical foundations to avoid casting serious doubts on their validity. To this end, we took advantage of the framework by Aranda et al. [2007], which has been properly adapted here to better handle the problem at hand. In addition, each experiment was carried out by following recommendations provided by Juristo and Moreno [2001] and by Wohlin et al. [2012].

The article is organized as follows. In Section 2, we discuss how we deal with comprehensibility in our family of experiments. In Section 3, we present the design of

[1]A family is composed of multiple similar experiments that pursue the same goal in order to build the knowledge needed to extract and to abstract significant conclusions [Basili et al. 1999]. Families of experiments allow researchers to answer questions that are beyond the scope of individual experiments and to generalize findings across studies. In addition, families of experiments can contribute to the conception of hypotheses that may not be suggested by individual experiments.

Table I. Moderator (Affecting) and Affected (Dependent) Variables

Kind of Variable	Name	Description
Moderator (Affecting)	Type of task	Tasks performed by people are facilitated or hindered at varying degrees. Comprehensibility requires the integration of information (e.g., source code and documentation) in the people mental model. This could lead to different results.
	Experience	Previous experience with the kind of pattern instance documentation and with programming and maintenance.
	Domain knowledge	Previous knowledge with the solution domain of the system under study and with its application domain.
	Problem size	Different notations might scale up with variable degrees of success.
Dependent (Affected)	Correctness of understanding	The degree to which a person can answer questions about a comprehension task on source code.
	Time	Time required to accomplish a comprehension task on source code.
	Confidence	Subjective confidence that people display regarding their own understanding.
	Perceived difficulty	Subjective judgment about the ease in obtaining information through the provided artifacts (e.g., analysis and design models).

the experiments according to the guidelines suggested by Jedlitschka et al. [2008]. In Section 4, we show the achieved results, while their discussion is presented in Section 5. Related work is shown in Section 6, while remarks and future work conclude the article in Section 7.

2. DEALING WITH SOURCE-CODE COMPREHENSIBILITY

Any study on comprehensibility should consider many variables [Aranda et al. 2007]. Some variables are affected by comprehensibility (*affected variables* or *dependent variables*), while others affect comprehensibility (*affecting variables* or *moderator variables*). In Table I, some dependent and moderator variables are highlighted with respect to source-code comprehensibility. In particular, in this table we summarize our modification to the framework by Aranda et al. [2007].

The larger the number of controlled dependent/moderator variables, the better. In controlled experiments some factors may be possible to control, while others difficult or impossible [Wohlin et al. 2012]. To increase confidence in the outcomes, we had to select and study a number of dependent and moderator variables. The following are the moderator variables considered in our family of experiments.

—*Type of task.* Tasks performed by people are facilitated or hindered at varying degrees. Comprehensibility of information search or maintenance tasks requires information integration, and this could lead to different results. We focus here on aspects related to the general comprehension of source code (e.g., question Q1 in Appendix A) and to the comprehension of source code with the goal of performing modification tasks (e.g., question Q12 in Appendix A).

—*Experience.* Finding competent people is a challenge in most software engineering studies [Bergersen et al. 2014]. In our family of experiments, we conducted four controlled experiments. The participants in each experiment had a similar level of experience with design-pattern development. We classified participants on the basis of their knowledge of high- and low-level design of object-oriented software systems and software development and maintenance.

The dependent variables are as follows.

—*Correctness of understanding.* The degree to which a person correctly answers questions about source code is a variable that estimates the comprehensibility achieved by people. To assess correctness of understanding, we used the comprehension questionnaire in Appendix A.
—*Time.* The time required to answer questions about source code is a variable that represents an approximation for effort. This is almost customary in literature (e.g., Ricca et al. [2014]) and compliant with the ISO/IEC 9126 standard [ISO 1991], where effort is the productive time associated with a specific project task.
—*Confidence.* The subjective confidence that a person shows about his/her own understanding of the furnished source code is expressed for each question in the comprehension questionnaire.
—*Perceived difficulty.* This concerns subjective judgment about the ease in obtaining information through the documentation of design-pattern instances (if present) and/or source code.

The theoretical framework by Aranda et al. [2007] was originally conceived to empirically evaluate model comprehensibility. It consists of a sequence of guidelines organized in the following order.

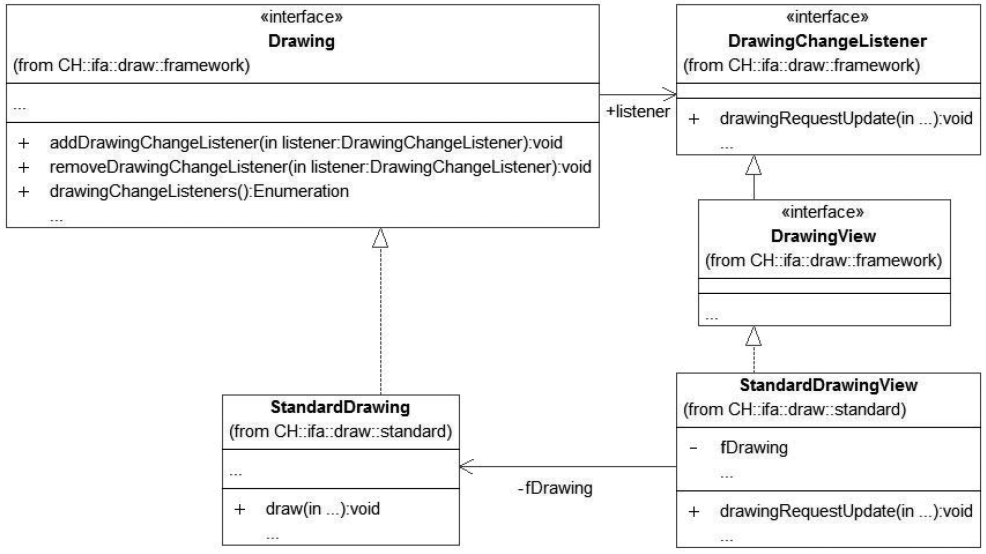
(1) *Selecting the notation*. There are several decisions to be taken to reduce as much as possible biases in the experimentation.
(2) *Articulating the underlying theory*. The assumptions under which a notation is useful should be made clear.
(3) *Formulating the claims of the notation*. The underlying theory is expressed as a set of claims regarding comprehensibility.
(4) *Choosing a control*. A baseline for the comparison has to be chosen.
(5) *Turning the claims into hypotheses*. Claims are turned into testable hypotheses that cover most of the dependent variables.
(6) *Informing the hypotheses*. The defined hypotheses are informed using frameworks from cognitive science theory.
(7) *Designing and executing the study*. Whatever the empirical method used for comprehensibility evaluation, some design choices should be taken into account (e.g., comprehension questionnaire).

In the following sections, we show how we have instantiated the guidelines in between selecting the notation and informing the hypotheses, while designing and executing the study is described in Section 3 (i.e., the family of experiments).

2.1. Selecting the Notation

Some notations have been proposed for documenting design patterns and their instances in source code. Many of these notations are based either on UML or on textual descriptions [Gamma et al. 1995; Heer and Agrawala 2006]. Those based on UML are widely used in software-engineering academic and industrial courses to teach design pattern-based development [Gamma et al. 1995; Bruegge and Dutoit 2003]. In our family of experiments, we investigated source-code comprehensibility when pattern instances implemented in it are documented graphically, as proposed by Gamma et al. [1995] using UML class diagrams (see, for example, Figure 1(a)), or textually, as done in JHotDraw[2] (see, for example, Figure 1(b)). In this latter case, textual descriptions

[2]It is a two-dimensional graphics framework for structured drawing editors. It can be downloaded at www.jhotdraw.org.



(a)

```
 1: /*
 2:  * @(#)DrawingChangeListener.java 5.1
 3:  *
     * Observer Pattern
     * Subject: Drawing
     * ConcreteSubject: StandardDrawing
     * Observer: DrawingChangeListener
     * ConcreteObserver: StandardDrawingView
     *
 4:  */
 5:
 6: package CH.ifa.draw.framework;
 7: ...
14: public interface DrawingChangeListener
                        extends EventListener {
16: ...
21:     /**
22:      ...
23:      */
24:     public void drawingRequestUpdate(...);
25: }
26:
```

(b)

Fig. 1. A sample of an instance for the observer design pattern: graphically (a) and textually documented (b).

are directly reported in source code as comments (e.g., Guéhéneuc and Antoniol [2008]), when design patterns are intentionally used. In many cases, identifier names (e.g., class names) suggest the pattern instances implemented in source code. We selected these two notations because both are widely used and known in academic and industrial contexts, so improving external validity. The kind of documentation for design-pattern instances represents the manipulated factor (or independent variable or main factor) in our family of experiments.

2.2. Articulating the Underlying Theory

The underlying theory is ideally obtained from the literature by identifying target users, domains, problem sizes, and required domain experience of people who will use a notation. The effect of design-pattern instance documentation on source-code comprehensibility has been scarcely investigated, therefore we could not rely on the literature. Based on our experience, knowledge, and results from our pilot study [Gravino et al. 2011], we postulated the following claims.

(1) Source-code comprehensibility increases when documented pattern instances are used as complementary information.
(2) Graphically documented pattern instances can be exploited by any user with experience in UML.
(3) Textually documented instances require the target user to have some experience in dealing with source code written by other developers.

Both kinds of documentation are mostly used as a communication mechanism among developers to make a software implementation (and, implicitly, the rationale behind it) clearer and more understandable.

2.3. Formulating the Claims

Since source-code comprehension increases when documentation is given (e.g., Arisholm et al. [2006], Budgen et al. [2011], Prechelt et al. [2002], and Scanniello et al. [2013]), we claimed that correctness in understanding unknown source code increases when pattern instances are documented and given to developers dealing with comprehension tasks. The use of this documentation should reduce possible misunderstandings among developers even when they have different levels of experience.

2.4. Choosing a Control

Our natural choice for the control/baseline was source code without documented instances, and without any reference to pattern instances from source-code identifiers. We opted for this strategy because it is rather common that professional developers and students unintentionally (or accidentally) use design patterns when writing source code (e.g., Guéhéneuc and Antoniol [2008]). Therefore, when a different developer has to comprehend this source code, S/he to deal with pattern instances that are not documented at all.

2.5. Turning Claims into Hypotheses

A graphical representation of a pattern instance might show a superset of the information provided by the corresponding textual representation. For example, in both representations, the roles each class plays within the pattern instance might be indicated, while in a textually documented instance the relationships among abstract and concrete classes and interfaces might not be reported. Figure 1(a) shows an example of a graphically documented instance of the observer design pattern [Gamma et al. 1995] within the source code of JHotDraw v5.1. Figure 1(b) shows how the same instance is explicitly reported within the source code as comment.

The graphical instance shows more information and thus should improve source-code comprehensibility. For example, from the class diagram in Figure 1(a), we understand that, when a drawing is changed, all its views are updated. More experienced developers might find the additional information provided by graphically documented instances to be of little use, because they can directly deduce such information from the name of the design pattern and from the role of each class and interface. On the other hand, developers with different levels of experience (e.g., less experienced) would receive

some help in using graphically documented instances even if the use of this kind of documentation might require more cognitive resources to switch the perception mode from text to graphic and back.

2.6. Informing the Hypotheses

We consider external cognition to have particular relevance in our study. This branch of cognitive science treats humans and the artifacts they use to solve a given task as a single cognitive entity. The perspective is that artifacts are part of problem solving resources and thus improve human reasoning. Representations might improve reasoning in several ways [Bauer and Johnson-Laird 1993]. According to Scaife and Rogers [1996], documented instances can reduce cognitive effort by putting knowledge in the representation, rather than in the head of the developer (i.e., offloading). That is, the less information to be kept in mind and the fewer rules needed to process it, better. This is especially true for graphically documented instances, because they show relationships among classes and the roles that each class plays [Guéhéneuc and Antoniol 2008]. When using textually documented instances, more experienced developers might not be hindered in inferring this information. Indeed, these developers might have concerns going from source code to documentation and vice versa when using graphically documented instances. Approaching source-code comprehension tasks in such a way might affect comprehensibility when a developer is not accustomed to using documentation that is kept externally from source code.

3. THE FAMILY OF EXPERIMENTS

The software engineering community has been embracing replications more readily (e.g., da Silva et al. [2014]). Even with increased interest in replications, there is no agreement yet on terminology, typology, purposes, operation, and other replication issues [Kitchenham 2008; Shull et al. 2008]. The lack of shared terminology allows authors to use different definitions for the same kind of replication and to use the same definition to refer to different kinds of replication [Baldassarre et al. 2014]. In general, we can define a replication as the repetition of an experiment, either as closely following the original experiment as possible or with a deliberate change to one or several of the original experiment parameters [Gómez et al. 2010]. In particular, there are two important factors that underlie the definition of replications: the *procedure* (i.e., the steps followed) and the *researchers* (i.e., the experimenters conducting the replication). As for the procedure, the kinds of replications range in-between *close* and *conceptual*. A close replication is a replication whose procedure is as close to the original procedures as possible [Shull et al. 2008], while a replication is conceptual if the research question is the same, but the experimental procedure is completely different from that of the original experiment. As for the researchers, we can distinguish between *internal* and *external* replications. Internal replications are conducted by the same group of researchers as in the original experiment [Mendonça et al. 2008], while external replications are executed by different experimenters also using the same procedure as the original experiment. In case of external replications, the results might be slightly affected by experimenters' biases [Juristo and Moreno 2001; Shull et al. 2008].

Experiments need to be replicated in different contexts, at different times, and under different conditions before they can produce generalizable knowledge [Shull et al. 2008]. From the scientific and industrial perspectives, we can identify two primary motivations for conducting replications: (i) they are necessary to solve problems and collect evidence because they bring credibility to a given research and (ii) they are valuable because they provide evidence of the benefits of a software engineering practice, thus allowing industrial stakeholders to use this information to support adoption decisions [Baldassarre et al. 2014; Colosimo et al. 2009; Pfleeger and Menezes 2000].

Table II. Participants in the Family of Experiments Grouped by Experiment

Experiment	Context	Description	Participants	Type	Year
E1-Prof	Practitioners	Original experiment	25	-	2011
E2-UniSA	University of Salerno	Internal replication	25	2nd year Master students	2012
E3-UniBAS	University of Basilicata	Internal replication	23	3rd year Bachelor students	2012
E4-UniBZ	Free University of Bozen-Bolzano	External replication	15	2nd/3rd year PhD students	2013

Differently from a single replication, a family of controlled experiments can contribute to the conception of important and relevant hypotheses that may not be suggested by an individual experiment/replication (e.g., any possible effect of the developers' experience) [Basili et al. 1999]. This is why we carried out a family of controlled experiments composed of the original experiment, two internal replications, and one external replication.

In Table II, we summarize information on the four experiments we conducted in our family. The original experiment, named E1-Prof, was carried out with professional developers in 2011. Some results from this experiment were preliminarily presented in Gravino et al. [2012]. All the participants in E1-Prof had at least a Bachelor degree in computer science. Among the participants in E1-Prof, there were neither PhD students nor PhD graduates. By varying participants' experience, E1-Prof was replicated in 2012 at the University of Salerno with 25 second-year students from the Master program in computer science. This experiment is named E2-UniSA. In the same year, we conducted the second replication, named E3-UniBAS. The participants in this replication were 23 third-year students from the Bachelor program in computer science at the University of Basilicata. The third replication was E4-UniBZ. It was performed at the Free University of Bozen-Bolzano with 15 PhD students in computer science. We considered this experiment an external replication because an investigator (the fifth author) performed this experiment on the basis of an experimental package provided by the original investigators (the first four authors). The new investigator participated neither in the design of the experiments nor in the preparation of the experimental material. The experiments E2-UniSA, E3-UniBAS, and E4-UniBZ are presented for the first time in this article.

Features that made the experiments in our family distinct from the pilot study [Gravino et al. 2011] were: (i) the experiment design and (ii) the kind of involved participants[3]. We also renewed and improved the experimental material.

For replication purposes, we made available on the Web an experiment package, the raw data, and a technical report [Scanniello et al. 2014b].

3.1. Goal

According to the *goal question metrics* (GQM) template by Basili and Rombach [1988], the goal of our family of experiments can be formalized as follows.

Analyze source-code comprehensibility *for the purpose of* evaluating graphically and textually documented design-pattern instances *with respect to* correctness of understanding, time, confidence, and perceived difficulty *from the point of view of* the

[3]The participants in the first experiment of the pilot study were Master students in computer science from the University of Basilicata, while those in E3-UniBAS were students from a Bachelor program in computer science from the same university. In the second experiment of the pilot study, the students were enrolled in the first year of a Master program in computer science at the University of Salerno, while in our family of experiments the Master students came from the same university, but were enrolled in the second year of the same program.

researcher and *from the point of view of* the practitioner *in the context of* object-oriented software.

3.2. Context Selection

For each experiment, its participants had the following characteristics.

—*E1-Prof*. The participants were Italian software professionals. They have been working for software companies in the contact network of the research groups of the original investigators. These companies were partners in research projects, or hosted students from the Universities of Basilicata and Salerno for internships, or employed alumni from either universities.
—*E2-UniSA*. The participants were students from the Software Engineering II course of the Master program at the University of Salerno. They all passed the following courses: Software Engineering I, Object-Oriented Programming I and II, and Databases. In particular, they knew the basics of high- and low-level design of object-oriented software systems (based on UML), software development, and software maintenance. The majority of the participants took an internship in a company as the final part of their Bachelor degree. The experience level of these students is considered inferior to that of the participants in E1-Prof.
—*E3-UniBAS*. The participants were students from the Software Engineering I course of a Bachelor program at the University of Basilicata. They all passed the following courses: Procedural Programming, Object-Oriented Programming I, and Databases. In the Software Engineering I course, they learned object-oriented modeling and UML. They had limited experience in developing and maintaining nontrivial software systems. These students can be considered the participants in our family of experiments with the lowest level of experience.
—*E4-UniBZ*. The participants were students from an international PhD course at the Free University of Bozen-Bolzano. They took their Master degree in computer science from universities worldwide. In particular, they came from: Italy (6), Lithuania (2), Russia (2), Romania (1), Morocco (1), Serbia (1), Thailand (1), and Brazil (1). The participants in E4-UniBZ can be considered those with the highest level of experience in our family of experiments, or at least their experience is not inferior to that of the participants in E1-Prof.

The prior knowledge and experience of the participants in each experiment can be considered rather homogeneous (especially in those experiments conducted with students [Verelst 2004]). The classification of participants in levels of experience follows an approach similar to that proposed by Ricca et al. [2010] and Abrahão et al. [2013].

Participation in the experiments was on a voluntary basis. Each participant took part in only one experiment. Results did not influence the grades in the software engineering courses where the replications E2-UniSA and E3-UniBAS took place. Professionals participated in the experiment as part of their work hours. This choice was made to encourage professionals to participate. No participants were obliged to participate in the experiments.

We selected a chunk (i.e., vertical slice) of JHotDraw v5.1 that included: (i) a nontrivial number of instances and (ii) well-known and widely adopted design patterns. In the source-code selection process, we have taken into account a trade-off between complexity of the implemented functionality in the chunk and the effort to comprehend source code. One of the authors (i.e., the third) detected and documented the pattern instances present in the source code of the experimental object. To this end, both the JHotDraw documentation and the PMARt dataset [Guéhéneuc 2007] were used. This allowed to document both intentional and unintentional instances (see Appendix B, where it is also possible to understand which classes participated in more than one instance). We

also took care not to exclude those classes in the selected source code that were related to other classes which played a role in a pattern instance. The selection process was iterative and allowed to select a total number of 10 instances. The following instances were present in the selected chunk of JHotDraw: State, Adapter, Strategy, Decorator, Composite, Observer, Command, Template Method, and Prototype. For the State design pattern, there were two instances, while only one instance was present for all the remaining design patterns. The chunk consisted of 1326 lines of code, 26 classes, and 823 lines of source-code comment.

The use of incomplete documentation and of a subset of the entire software on which a maintenance operation impacts is quite common in the software industry [Scanniello et al. 2014a]. For example, this happens when only a part of the documentation actually exists (e.g., in lean development processes) or is updated [Bruegge and Dutoit 2003; McDermid 1991], or only a subset of the entire documentation is useful to perform a given maintenance operation [Asuncion et al. 2007; Lindvall and Sandahl 1996].

As for E1-Prof, E2-UniSA, and E3-UniBAS, we translated the experimental material including the comments in the original source code of JHotDraw from English into Italian. This allowed to reduce biases related to different levels of participants' familiarity with English. For E4-UniBZ, the comments were not translated because the official language of the PhD program at the Free University of Bozen-Bolzano is English. As for those participants provided with source code alone and graphically documented instances, we removed any possible information on pattern instances present in comments and identifiers (e.g., CompositeFigure was renamed ArrayFigure). We took this decision to analyze only the effect of pattern instance documentation.

3.3. Selection of Variables

To determine time, we used the overall time (expressed in minutes) to answer the comprehension questionnaire. The higher the value of time, the greater the effort to accomplish the comprehension tasks.

To quantify correctness of understanding, we used an approach based on information retrieval theory [Salton and McGill 1983]. We defined: $A_{s,i}$ as the set of string items provided as answer to the Question i in the comprehension questionnaire by the participant s; and C_i as the correct set of items expected for the question i (i.e., the oracle). Then, we computed precision and recall for each answer to a given question in the questionnaire as follows:

$$precision_{s,i} = \frac{|A_{s,i} \cap C_i|}{|A_{s,i}|}, \tag{1}$$

$$recall_{s,i} = \frac{|A_{s,i} \cap C_i|}{|C_i|}. \tag{2}$$

Precision (i.e., the fraction of items in the answer that are correct) and recall (i.e., the fraction of correct items in the answer) measure accuracy and completeness of the answer to a given question, respectively. To get a trade-off between accuracy and completeness, we used the balanced F-measure of precision and recall:

$$F{-}Measure_{s,i} = \frac{2 \cdot precision_{s,i} \cdot recall_{s,i}}{precision_{s,i} + recall_{s,i}}. \tag{3}$$

For each participant, correctness of understanding is computed as the average of the F-measure values of all the questions in the comprehension questionnaire. This variable takes values in-between 0 and 1. A value close to 1 means that the participant achieved good source-code comprehension because she/he answered rather well to all questions

Q2. Indicate the class/es and the method/s in charge of creating, drawing, and updating the instances of the class Figure.				
How much do you trust your answer^{+}?				
☐ Unsure	☐ Not Sure Enough	☐ Sure Enough	☐ Sure	☐ Very Sure
How do you assess the question^{+}?				
☐ Very Difficult	☐ Difficult	☐ On Average	☐ Simple	☐ Very Simple
What is the source of information used to answer the question^{+}?				
☐ Previous Knowledge (PK)		☐ Internet (I)	☐ Source Code (SoC)	
$^{+}$ Mark only one answer				

Fig. 2. A sample question from comprehension questionnaire.

Table III. Experiment Design

Experiment	GD	TD	SC	Total
E1-Prof	8	9	8	25
E2-UniSA	8	9	8	25
E3-UniBAS	8	8	7	23
E4-UniBZ	5	5	5	15

in the comprehension questionnaire. Conversely, a value close to 0 means that source-code comprehension was not good. We chose the aforesaid measures because they are widely used in the empirical software engineering field (e.g., Ricca et al. [2010], and Abrahão et al. [2013]).

Figure 2 shows a sample question. It expected as the correct answer the following set of items: `CreationTool`, `ArrayFigure`, `StandardDrawingView`, `createFigure()`, `draw()`, and `drawingRequestUpdate()`. These items can be derived by the instance of the observer pattern shown in Figure 1 and by the instances of the prototype and composite patterns. The Observer instance is in charge of painting and/or repainting objects that are instances of `Figure`. The Prototype instance helps in understanding the object that would manage the creation of a template for `Figure`, while the Composite instance draws each base element of an object `Figure`. Details on the latter two instances can be found in the experimental material we made available on the Web [Scanniello et al. 2014b]. If, for example, a participant provided `CreationTool`, `ArrayFigure`, and `drawingChangeListeners()` as his/her answer, the corresponding value for correctness of understanding would be 0.44. This results from the precision and recall values, respectively being 0.66 and 0.33. In fact, the number of correct items provided is 2 (`CreationTool` and `ArrayFigure`), the total number of items provided is 3, and the expected number of correct items is 6.

As mentioned in Section 2.1, *method* is the main factor (or manipulated factor) in our family of experiments. It is a nominal variable and assumes values in {SC, TD, GD}, where SC indicates that the participant was provided with source code without documented design-pattern instances. On the other hand, TD and GD indicate that participants were provided with source code added with textually or graphically documented instances, respectively. The moderator variable experience assumes values in {E1-Prof, E2-UniSA, E3-UniBAS, E4-UniBZ}.

3.4. Experiment Design

We used the one-factor-with-three treatments design [Wohlin et al. 2012] shown in Table III. The use of a different experimental design (such as a within-participant counterbalanced design) with nontrivial experimental objects (as in our family of experiments) may bias results, introducing a possible carry-over effect: if a participant is tested first under the experimental condition *A* and then under the experimental condition *B*, she/he could potentially exhibit better or worse performance under the condition *B*.

We used the participants' working experience[4] as the blocking factor for E1-Prof, while the participants' ability[5] was the blocking factor for both E2-UniSA and E3-UniBAS. As for E4-UniBZ, we used the final grade of the Master degree[6]. Due to the used experiment design, we use hereafter "group" and "treatment" interchangeably.

3.5. Hypotheses

We have defined and investigated two null hypotheses. Each is described as a parameterized hypothesis as follows.

—*Hn1_X*. With respect to the dependent variable X (i.e., correctness of understanding or time), there is no difference among source code with undocumented design-pattern instances, source code with textually documented design-pattern instances, and source code with graphically documented design-pattern instances.
—*Hn2_X*. With respect to the dependent variable X and the participants' experience, there is no difference among source code with undocumented design-pattern instances, source code with textually documented design-pattern instances, and source code with graphically documented design-pattern instances.

When null hypotheses are rejected, it is possible to accept the alternative ones that can be easily derived (e.g., Ha1_X: with respect to the dependent variable X, there is difference among source code with undocumented design-pattern instances, source code with textually documented design-pattern instances, and source code with graphically documented design-pattern instances).

3.6. Experiment Tasks

We asked participants to perform the following tasks.

—*Comprehension task*. Independently from the treatment, the comprehension questionnaire was composed of 14 open questions (see Appendix A). These questions were divided into three groups to let participants take a break, if needed, when passing from one group of questions to the next. This choice aimed to reduce fatigue-effect biases. The questions were arranged into these groups on the basis of the recommendations by Sillito et al. [2008]. In particular, the questions in the first group focused on expanding those points in the source code believed to be relevant and related to software comprehension, often by exploring relationships among entities (e.g., classes and method). The questions in the second group were concerned with understanding concepts in the source code that involved multiple relationships and software entities. Answering these questions required understanding the overall structure of a subgraph[7]. To answer the questions in the third group, the participant

[4]A professional was considered as junior developer when his/her working experience was less than or equal to 3 years, senior otherwise. There were 10 junior developers and 15 senior developers.

[5]In Italy, the exam pass grades are expressed as integers and take values in-between 18 and 30. The lowest grade is 18, while the highest 30. Students with a *grade point average* (GPA) less than or equal to 24 were considered low-ability participants; otherwise high [Abrahão et al. 2013].

[6]A PhD student was considered to be low when his/her final grade was less than or equal to 8, high otherwise. All the final grades were normalized in-between 1 and 10 because of differences in the organization of the universities where the PhD students took their Master grade. For example, final Master grades in Italy are expressed as integers and take values in-between 66 and 110, so a student with a final grade less than or equal to 88 was considered a low-ability participant.

[7]Sillito et al. [2008] see a code base as a graph of entities (classes, methods, and fields, for example) and the relationships between these entities (references and calls, for example). To answer any specific question related to the comprehension of a software, the authors believe that it requires considering some subgraph of the code base.

Table IV. Post-Experiment Survey Questionnaire

Id	Question	Answers
S1	I had enough time to perform the task	(1-5)
S2	The tasks I performed were perfectly clear to me	(1-5)
S3	The task objectives were perfectly clear to me	(1-5)
S4	Comments included in the source code were clear	(1-5)
S5	I found the arrangement of the comprehension questions in three parts to be useful	(1-5)
S6	The design-pattern instances were useful to answer the questions	(1-5)
S7	The design-pattern instances were well documented	(1-5)
S8	How much time (in percentage) you believe to have spent in reading source code?	(A-E)
S9	How much time (in percentage) you believe to have spent in analyzing design-pattern instances?	(A-E)
1 = Strongly agree, 2 = Agree, 3 = Neutral, 4 = Disagree, 5 = Strongly disagree		
A. < 20%; B. ≥ 20% and < 40%; C. ≥ 40% and < 60%; D. ≥ 60% and < 80%; E. ≥ 80%		

had to deal with overrelated groups of subgraphs. That is, these questions involved understanding relationships between multiple subgraphs or the understanding of the interaction between a subgraph and the rest of the system. All the questions in the comprehension questionnaire were formulated using a similar form/schema. For each question, the participants in the groups GD and TD had to specify whether the answer was deduced from: (DPI) *design-pattern instances*, (PK) *previous knowledge*, (I) *internet*, or (SoC) *source code*. If participants specified DPI, they were also asked to indicate the used instance/s. Participants in the group SC could choose among PK, I, or SoC (see Figure 2). This was the only difference introduced in the comprehension questionnaires used for SC, GD, and TD. As for confidence level, participants had to chose among the following possibilities: Unsure, Not sure enough, Sure enough, Sure, and Very sure. Furthermore, perceived difficulty-level admitted values were {Very Difficult, Difficult, On Average, Simple, and Very Simple}.

—*Post-experiment task*. We asked participants to fill in the post-experiment survey questionnaire shown in Table IV. The goal of this questionnaire was to obtain feedback about participants' perceptions of the experiment execution. Answers to questions S1–S7 were based on a five-point Likert scale [Oppenheim 1992]: from strongly agree (1) to strongly disagree (5). Questions S8 and S9 demanded answers according to a different five-point Likert scale based on the percentage of time used to deal with source code and documented pattern instances (see last row of Table IV). The participants in the GD and TD groups had to answer all the questions in Table IV, while other participants answered just questions in-between S1 and S5.

3.7. Experiment Procedure

The experiment procedure adopted for the replications was different from that in the original experiment, where at most two professionals were involved in each experimental session. In fact, it is practically impossible to conduct a single experimental session gathering together professionals from different companies. Before the controlled experiment, all the professionals had to fill in a pre-questionnaire. This questionnaire was sent and returned by email. The gathered information was used to classify

participants as junior or senior developers. Just before an experiment session, experiment supervisors (two of the authors, namely the first and the third) highlighted the study goal without providing details on its hypotheses. Experiment sessions were carried out under controlled conditions to avoid biasing results. Experiment supervisors were the same at each session. This allowed to reduce possible observation biases. No training session on tasks similar to the ones in the experiment was carried out. The main reasons were: (i) they should have adequate experience in performing maintenance operations on source code written by others and (ii) time and logistic constraints made the execution of a training session impossible (the use of professionals might cause this kind of concern).

As for the replications, we conducted them in research laboratories under controlled conditions. A few days before each experiment, participants attended a training session where supervisors presented detailed instructions on the experiment and recalled basic notions on design-pattern-based development. All the participants in a single experiment attended this session together.

We monitored participants to prevent their communication with each other. This was also done in E1-Prof, when more professionals participated in one session. At the end of each experiment, we asked participants to return material. No time limit to perform the experiment was imposed. This implies that quality and time are considered separately when analyzing results [Bergersen et al. 2011, 2014]. As suggested by Bergersen et al. [2011], this is acceptable when the goal of a study is not to characterize individual differences among participants [Bergersen et al. 2011].

The participants used a general-purpose and well-known text editor, namely UltraEdit (www.ultraedit.com). We opted for this editor because it was used in basic programming language courses at both the University of Salerno and University of Basilicata. Therefore, the prior knowledge of the participants in E2-UniSA and E3-UniBAS should be rather homogeneous on this tool. Finally, we did not allow the participants to choose their preferred IDE (*integrated development environment*) because familiarity with the tool might skew results in an undesirable way.

We allowed all the participants to use the Internet to accomplish the experiment session since developers usually exploit this medium as support for their daily work activities. We tried to reproduce actual working conditions.

We asked all the participants to use the following experimental procedure: (i) writing down their name and start-time; (ii) answering the questions of the comprehension questionnaire without executing source code; and (iii) writing down the end-time. The experiment procedure was the same for each group of questions within the comprehension questionnaire. We did not suggest any approach to deal with source code. To avoid wasting time, we only discouraged reading all the source code.

We provided all the participants with a printed copy of the following experiment material: (i) the comprehension questionnaire and (ii) a post-experiment survey questionnaire. The participants in group GD were provided with the source code (without any reference to pattern instances) and a printed copy of the document where each pattern instance was graphically reported (see Figure 1(a)). The participants in the group TD were provided with source code that included references to the instances in the comments (see Figure 1(b)). As for the group SC, the participants were provided with source code without any kind of documentation.

3.8. Analysis Procedure

To perform data analysis, we used the R environment (www.r-project.org) for statistical computing and carried out the following steps.

(1) We undertook the descriptive statistics of the dependent variables.
(2) To test null hypotheses, we adopted nonparametric tests due to the sample size and mostly the non-normality of the data. As for Hn1_X, we used the Kruskal-Wallis test. This statistical test is a nonparametric alternative to ANOVA in case of one factor with more than two treatments [Wohlin et al. 2012]. We used the Kruskal-Wallis test because of the design of the experiments (only unpaired analyses were possible) in the family and for its robustness [Wohlin et al. 2012]. As for Hn2_X, we opted for a two-way permutation test [Baker 1995], a nonparametric alternative to the two-way ANOVA test. In all the performed statistical tests, we decided (as customary) to accept a probability of 5% of committing Type-I error [Wohlin et al. 2012].
(3) The chosen statistical tests allow the presence of a significant difference between independent groups to be verified, but do not provide any information about the magnitude of such a difference. To measure this difference (if present), we used the Cliff's d nonparametric effect-size measure [Kampenes et al. 2007]. Effect size is considered negligible for $|d| < 0.147$, small for $0.147 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [Kampenes et al. 2007]. We also computed the average percentage improvement for the defined dependent variables (i.e., time and correctness of understanding). For example, given the variable time and the methods SC and TD, we computed $\frac{Mean_{Time}(GD)-Mean_{Time}(SC)}{Mean_{Time}(SC)}\%$. If the obtained value was negative, participants spent on average less time with GD. Indeed, the lower the value, the lower the effort to comprehend source code with GD. As for correctness of understanding, it estimates the average percentage improvement in source-code comprehension. The higher the value, the better source-code comprehension with GD is.
(4) We also analyzed the statistical power of each performed test. Statistical power is defined as the probability that a statistical test will correctly reject a null hypothesis [Dybå et al. 2006]. Knowledge of statistical power can influence the planning, execution, and results of an empirical study (e.g., Arisholm et al. [2007], and Cohen [1988]). Statistical power can be also determined post-hoc [Dybå et al. 2006; Scanniello et al. 2014a]. The highest value is 1, while 0 the lowest. Therefore, the higher the statistical power value, the higher the probability to reject a null hypothesis when it is actually false. The value 0.80 is considered as a standard for adequacy [Ellis 2010]. The statistical power is computed as 1 minus the Type-II error (i.e., β-value). This type of error indicates that a null hypothesis is false, but that the statistical test erroneously fails to reject it. In the discussion of the results, we used the β-value when a statistical test was unable to reject a null hypothesis. The higher the β-value, the lower the probability of erroneously not rejecting a null hypothesis is.
(5) We used mosaic plots to graphically summarize the results for the confidence and perceived difficulties as well as the source of information the participants exploited to answer questions in the comprehension questionnaire.
(6) To summarize raw data and to support their discussion, we also used boxplots.

3.9. Differences and Issues

3.9.1. Differences among Experiments. The choice and numbers of factors to be varied among experiments is relevant because a large number of variations could make it less likely to trace observed results to the factor of interest (manipulated factor), and thus the result validity and interpretation could be greatly affected [Shull et al. 2008; Scanniello et al. 2014a]. In the following, we summarize variations among the four experiments in our family.

—*Participants*. Variation in the participants' experience and in environmental factors can contribute some confidence that the effect of the factor under study is not limited to one particular setting or experimental environment.
—*Training session*. Professionals did not carry out a training session on tasks similar to those used in the experiment.
—*Experiment material*. Sources of possible confusion in materials were removed after each experiment. As for E4-UniBZ, the experimental material was in English. Independently from the treatment, the participants in E4-UniBZ were asked to answer two additional questions to understand the *common sense* in the representation and use of design patterns. In particular, the following two open questions were asked.
  (i) Based on your experience and knowledge, please explain why (or why not) the documentation of design-pattern instances should improve source-code comprehensibility and maintainability.
  (ii) Please suggest how design-pattern instances should be documented to improve source-code comprehensibility and maintainability.

  We added this difference in the replication E4-UniBZ because we believe more significant comments might be given by those participants with an experience and maturity level higher than those of the participants in E2-UniSA and E3-UniBAS.

3.9.2. Documentation and Communication Issues. The success or failure of replications might be influenced by issues such as documentation [Shull et al. 2004] and communication among experimenters [Vegas et al. 2006]. To deal with these issues, we used laboratory packages, knowledge-sharing mechanisms, and communication media. In particular, with regard to documentation, original experimenters translated all the material (not initially written in English) from Italian into English. The investigator of the external replication (the fifth author) was provided with all the experiment material, and asked for clarifications and implemented modifications if they were needed. In addition, the experimenters shared: (i) the papers where the pilot study and original experiment were presented [Gravino et al. 2011, 2012] and (ii) a document to provide a common background.

We used instant messaging tools, mobile phones, teleconference tools, and email to establish a communication channel among experimenters. To reduce consistency issues across the experimenters, the results from the most important interactions were reported in a shared document where experiment decisions were recorded.

4. RESULTS

In this section, we present the obtained results following our analysis procedure.

4.1. Descriptive Statistics

Table V shows the values of mean and standard deviation for time and for correctness of understanding grouped by method. The distribution of the values for these variables grouped by method and experiment is graphically shown by the boxplots in Figure 3.

We can observe that, in all the experiments (except for E2-UniSA), the participants belonging to the group/treatment SC spent on average more time than the participants in the groups GD and TD. A small difference can be observed between the groups GD and TD in all the experiments with respect to the mean time to accomplish the task. Boxplots in Figure 3(a) show that the same trend is present among the experiments for each group (e.g., participants in E1-Prof spent more time than those in E4-UniBZ).

As for correctness of understanding, the participants in groups GD and TD achieved, on average, better scores. This does not hold in E4-UniBZ for the group TD. On the other hand, a small difference is observable in favor of the group TD for E1-Prof. Boxplots in Figure 3(b) show that more experienced participants benefit from

Table V. Descriptive Statistics for GD, TD, and SC

Experiment	Dependent variable	GD		TD		SC	
		Mean	St. Dev.	Mean	St.Dev.	Mean	St.Dev.
E1-Prof	Time	142.2	31.17	136.4	37.00	147.4	42.81
	Correctness of understanding	0.51	0.10	0.54	0.07	0.40	0.10
E2-UniSA	Time	85.5	24.089	93.44	22.07	83.5	33.98
	Correctness of understanding	0.406	0.116	0.43	0.1	0.31	0.12
E3-UniBAS	Time	112	30.836	125.125	21.397	148.857	40.268
	Correctness of understanding	0.379	0.073	0.349	0.131	0.384	0.111
E4-UniBZ	Time	94.6	13.686	77.8	20.777	108.4	28.5
	Correctness of understanding	0.511	0.034	0.602	0.177	0.388	0.124

Table VI. Results for Hn1_X (SC vs. GD vs. TD)

Experiment	Dependent variable	p-value
E1-Prof	Time	No (0.791)
	Correctness of understanding	**Yes** (0.035)
E2-UniSA	Time	No (0.943)
	Correctness of understanding	No (0.091)
E3-UniBAS	Time	No (0.097)
	Correctness of understanding	No (0.836)
E4-UniBZ	Time	No (0.343)
	Correctness of understanding	**Yes** (0.046)

documented pattern instances more than do less experienced participants. As for what concerns the treatment SC, there is a non-noteworthy difference among participants in all experiments.

4.2. Hypotheses Testing

4.2.1. Hn1_X: SC vs. GD vs. TD. The results for the null hypothesis Hn1_X are summarized in Table VI. The Kruskal-Wallis[8] test indicates that this null hypothesis can be rejected on correctness of understanding for E1-Prof and E4-UniBZ. The p-values are 0.035 and 0.046, respectively.

When three different groups are compared, a significance test only shows whether there is a significant difference in general [Wohlin et al. 2012]. A post-hoc analysis is needed afterwards, to evaluate between which conditions the significant effect occurs. Therefore we performed a post-hoc analysis on correctness of understanding for the experiments E1-Prof and E4-UniBZ, namely those experiments where a significant difference was observed. For each of these two experiments, we tested the following null hypotheses: (i) $Hn_{SC\,vs.D}$, that is, source-code comprehensibility does not increase when using D (i.e., TD or GD) with respect to SC; and (ii) $Hn_{TD\,vs.GD}$, that is, source-code comprehensibility does not increase when using GD with respect to TD. These hypotheses are one-sided on the basis of the results of the descriptive statistics. $Hn_{SC\,vs.D}$ is a parametrized null hypothesis.

The preceding hypotheses have been verified by the Mann-Whitney test [Conover 1998] (also known as the Wilcoxon rank-sum test). The results of the performed post-hoc analysis are summarized in Table VII and Table VIII.

[8]To evaluate whether data was not normally distributed, we applied the Shapiro-Wilk W test [Shapiro and Wilk 1965]. If this test returns a p-value smaller than the chosen threshold (0.05 in our case), data is considered to be non-normally distributed. The Shapiro-Wilk W test returned 0.02 as p-value for the group GD in E1-Prof. Although p-values larger than 0.05 were obtained in all other cases, we performed nonparametric analyses for uniformity reasons and to be as conservative as possible.

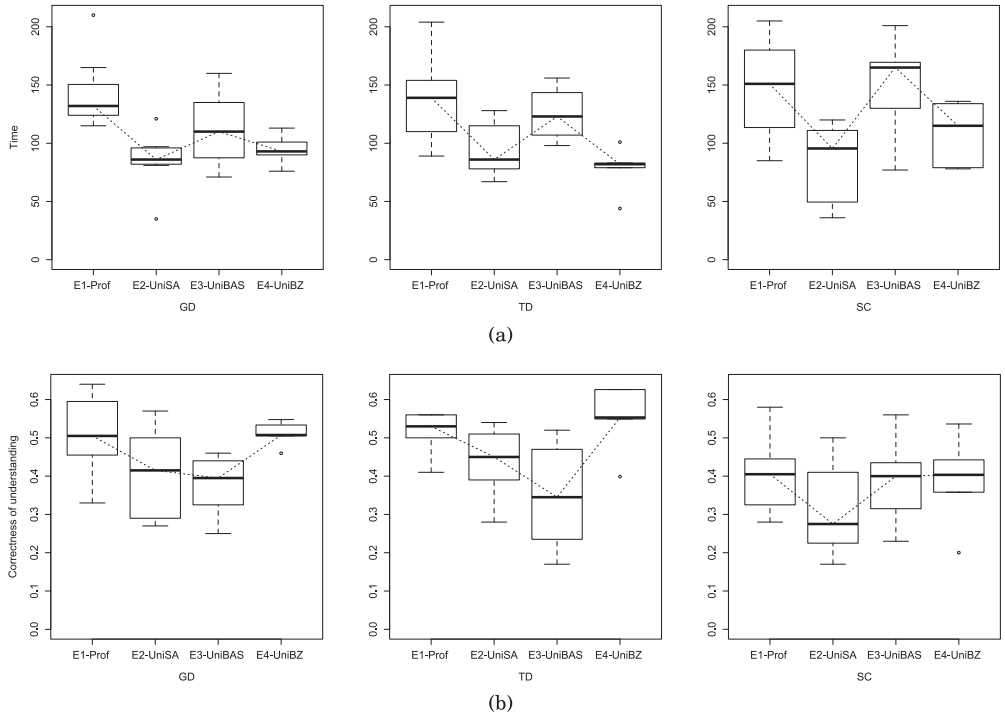


Fig. 3. Distributions for time (a) and correctness of understanding (b) grouped by method and experiment.

Table VII. Post-Hoc Analysis on Correctness of Understanding ($Hn_{SCvs.TD}$ and $Hn_{SCvs.GD}$ in E1-Prof and E4-UniBZ)

Experiment	Hypothesis	$p-value$	Cliff's d	Statistical Power	Average Improvement
E1-Prof	$Hn_{SCvs.TD}$	**Yes** (0.008)	−0.708	0.903	93.3%
	$Hn_{SCvs.GD}$	**Yes** (0.033)	−0.562	0.607	87%
E4-UniBZ	$Hn_{SCvs.TD}$	**Yes** (0.03)	−0.76	0.6	116.2%
	$Hn_{SCvs.GD}$	**Yes** (0.047)	−0.68	0.528	92.8%

Table VIII. Post-Hoc Analysis on Correctness of Understanding ($Hn_{TDvs.GD}$) in E1-Prof and E4-UniBZ)

Experiment	$p-value$	Cliff's d	β-value
E1-Prof	No (0.314)	−0.153	0.865
E4-UniBZ	No (0.072)	−0.6	0.782

The Mann-Whitney test indicated the presence of statistically significant differences in E1-Prof and E4-UniBZ for both $Hn_{SCvs.TD}$ and $Hn_{SCvs.GD}$ (see Table VII). The obtained p-values range in-between 0.008 (SC versus TD in E1-Prof) and 0.047 (SC versus GD in E4-UniBZ). In both experiments, the effect size is large and the values range in-between 0.562 and 0.76, and also the average improvement values are very high (from 87% to 116.2%). These results indicate better source-code comprehension for TD and GD than for SC, in practical terms. The statistical power values are not always adequate. This could be due to the number of participants in the groups.

Table IX. Results for Hn2_X

	Method	Experience	Method vs. Experience
Dependent variable		$p - value$	
Time	No (0.33)	**Yes** (<0.001)	No (0.495)
Correctness of understanding	**Yes** (<0.001)	**Yes** (<0.001)	No (0.157)

As for $Hn_{TDvs.GD}$, the Mann-Whitney test indicated that this null hypothesis can not be rejected (see Table VIII). Therefore, the difference between the participants who used graphically documented and textually documented pattern instances is not statistically significant. The β-values are close to the standard for adequacy (0.865 and 0.782). The effect size is either small (in E1-Prof) or large (in E4-UniBZ).

These results suggest that the documentation of pattern instances aids in correctness of understanding for object-oriented source code in E1-Prof and E4-UniBZ. In addition, we observed a non-noteworthy difference in comprehending source code when pattern instances are documented either graphically or textually.

4.2.2. Hn2_X: Participants' Experience Effect. Table IX summarizes the results from the two-way permutation test. The effect of participants' experience is statistically significant on time and no interaction between method and experience is present. As shown in Figure 3(a), professionals spent more time accomplishing the comprehension task when pattern instances are documented. There is a non-noteworthy difference in the other experiments for the group GD. As for the group TD, the participants in E3-UniBAS spent more time than the participants in the other two replications. The results from the two-way permutation test also indicate a positive effect of method and experience on the correctness of understanding. The p-values are less than 0.001, respectively. More experienced participants benefited more than less experienced ones when pattern instances are documented (see Figure 3(b)). No interaction between method and experience is shown on correctness of understanding.

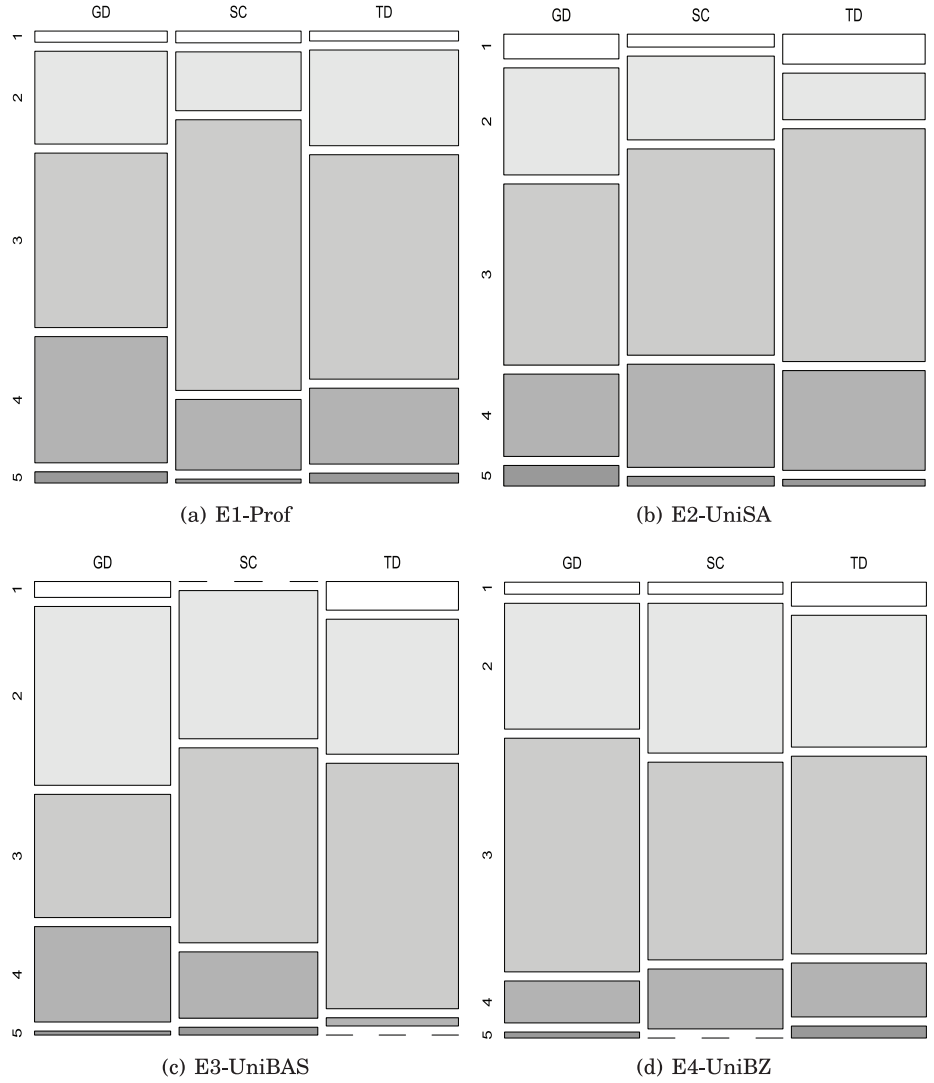
The data analysis results suggest that participants' experience has a statistically significant effect on source-code comprehensibility with respect to both the variables time and correctness of understanding, while there is no significant interaction effect.

4.3. Confidence

Figure 4 shows mosaic plots for the confidence that the participants displayed regarding their own understanding of the source code in the comprehension task. The data are grouped by experience and method. The participants generally indicated "sure enough", "sure", and "very sure" as their own confidence level of source-code understanding. This holds in all the experiments and for all the methods (i.e., GD, SC, and TD). Indeed, the participants in E3-UniBAS expressed the worst level of confidence in understanding. A possible reason could be related to the participants' knowledge of high- and low-level design of object-oriented software systems and software development and maintenance.

4.4. Perceived Difficulty

Subjective judgment on the ease in obtaining information through the documentation of pattern instances (if present) and/or source code is visually summarized by the mosaic plots in Figure 5. Independently from the method, the most frequent answer given in all the experiments of our family is "on average" (i.e., 3). Although there is a non-noteworthy difference among the experiments in our family, it seems that less experienced participants found it less easy to obtain information from the documentation provided to accomplish comprehension tasks.



(a) E1-Prof (b) E2-UniSA

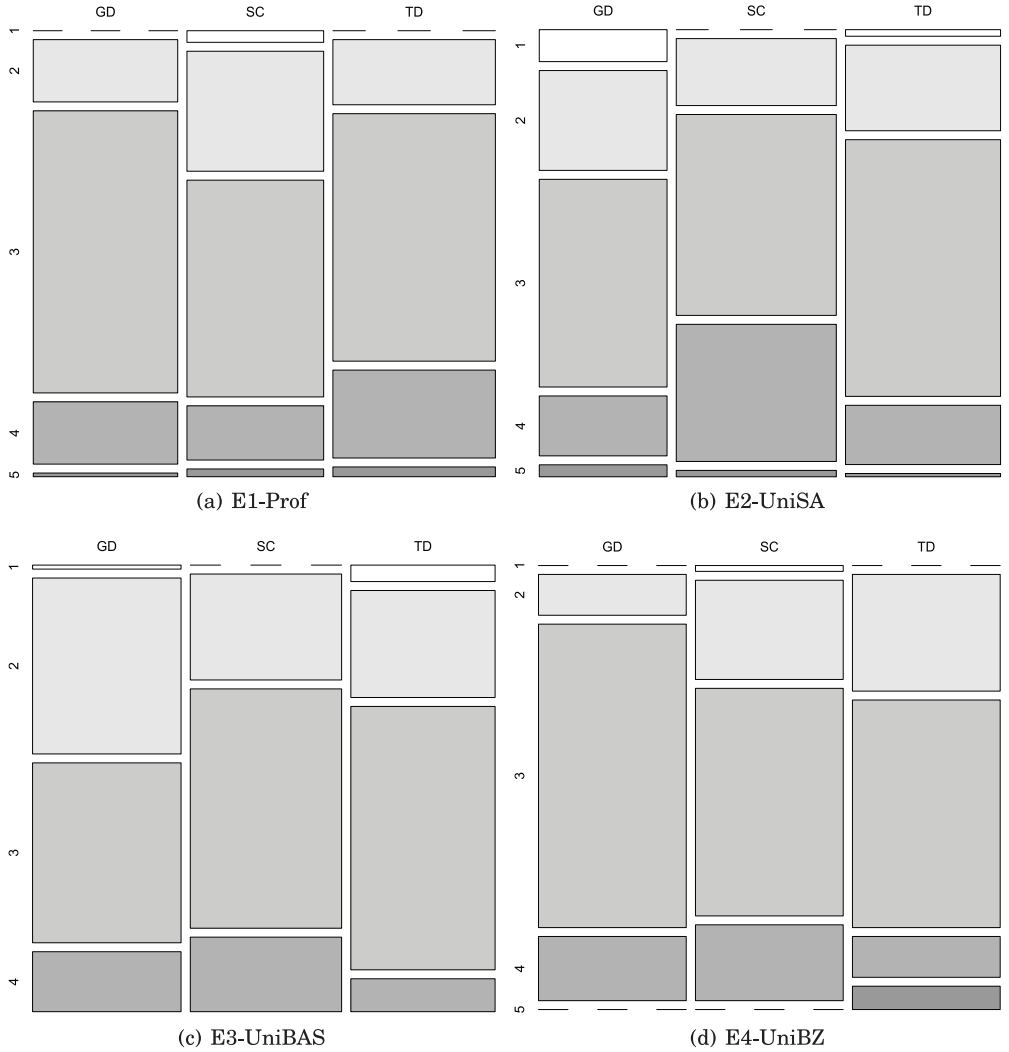
(c) E3-UniBAS (d) E4-UniBZ

Fig. 4. Mosaic plot for confidence (1: Unsure; 2: Not sure enough; 3: Sure enough; 4: Sure; 5: Very sure).

4.5. Source of Information

Figure 6 shows mosaic plots regarding the source of information for each experiment. The participants indicated source code as the main source of information for answering questions in the comprehension questionnaire. This finding held in all of the experiments except for the group GD in E1-Prof, where the participants indicated DPI as the main source of information. As for E3-UniBAS and E4-UniBZ, the difference between DPI and SoC is less evident for the group TD. The participants in these two experiments indicated that they did not use the Internet to answer the questions of the comprehension questionnaire.

To accomplish a source-code comprehension task, the participants perceived documented instances as a relevant, but not predominant source of information. This is



(a) E1-Prof (b) E2-UniSA

(c) E3-UniBAS (d) E4-UniBZ

Fig. 5. Mosaic plot for perceived difficulty (1: Very Difficult; 2: Difficult; 3: On Average; 4: Simple; 5: Very Simple).

a case where controlled experiments provide insight into the difference between perceived usefulness of a given method and the actual advantage of using it.

4.6. Post-Experiment Survey Questionnaire

The time needed to carry out experiments was considered appropriate (question S1). The median value was 1 (strongly agree) for all experiments except for E2-UniSA, where the median was 3 (neutral). The objectives (S2) were considered clear in all experiments except in E2-UniSA: the median values were 2 (agree) and 3 (neutral), respectively. The participants found comprehension tasks to be clear. The median value was 2 (agree) in all experiments. The answers to S4 suggest that the participants found comments to be clear. In particular, the median value was 1 (strongly agree) in E3-UniBAS, while it was 2 (agree) in all remaining experiments. All the participants

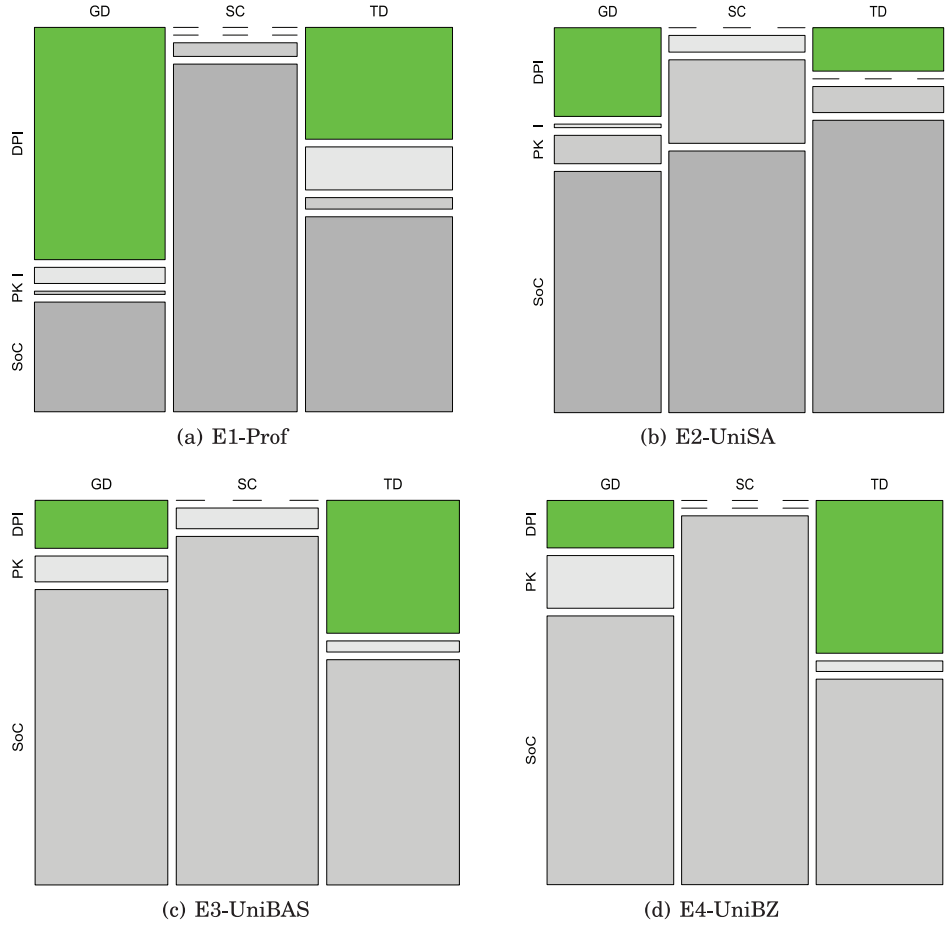


Fig. 6. Mosaic plot for source of information: (DPI) *design-pattern instances*, (PK) *previous knowledge*, (I) *internet*, or (SoC) *source code*.

agreed on the usefulness of the arrangement of the comprehension questionnaire in three parts (S5). In all experiments, the median value was 2 (agree).

As for E2-UniSA, those participants who carried out the experiment in the groups GD or TD expressed a neutral judgment (the median value) on the usefulness of pattern instances to accomplish the comprehension task (S6). The participants in other experiments agreed on the usefulness of pattern instances. In these experiments, the median value was 2. The participants found pattern instances to be well documented. The median value for S7 was 2 (agree) in all experiments.

Regarding S8, the median value was equal to D in all experiments. The participants indicated that the time they spent in analyzing source code ranged from 60% to 80% of the total time. The participants in E2-UniSa specified that the time to consult pattern instances was less than 20% (S9) of the total time to accomplish the comprehension task. All other participants specified a range between 20% and 40%.

4.7. Further Analyses

In the following, with respect to correctness of understanding, we compare the results of those participants who correctly identified the instances needed to answer the

Table X. Correctness of Understanding Grouped by CI and NCI

Experiment	Method	CI		NCI		p-value
		Mean	St.Dev.	Mean	St.Dev.	
E1-Prof	GD	0.671	0.309	0.499	0.359	**Yes** (0.004)
	TD	0.73	0.277	0.337	0.333	**Yes** (<0.001)
E2-UniSA	GD	0.721	0.252	0.392	0.339	**Yes** (<0.001)
	TD	0.694	0.342	0.34	0.358	**Yes** (<0.001)
E3-UniBAS	GD	0.602	0.328	0.268	0.315	**Yes** (<0.001)
	TD	0.626	0.265	0.35	0.363	**Yes** (0.006)
E4-UniBZ	GD	0.644	0.28	0.584	0.376	No (0.39)
	TD	0.725	0.298	0.422	0.324	**Yes** (<0.001)

questions in the comprehension questionnaire and those who did not. We conclude with the feedback that the participants in E4-UniBZ gave on how design-pattern instances should be documented to improve source-code comprehensibility and maintainability.

4.7.1. Correctly vs. Incorrectly Identified Instances. The answers to the questions in the comprehension questionnaire have been classified as: CI (*correctly identified* pattern instances) and NCI (*non-correctly identified* pattern instances). To understand whether the identification of design-pattern instances affects correctness of understanding, we compute the mean and standard deviation values on these two groups of answers. These statistics are summarized in Table X.

The descriptive statistics suggest that correctness of understanding improves when pattern instances are correctly recognized. This result held in all experiments with instances documented both graphically and textually. In addition, there is a non-noteworthy difference between the groups GD and TD regardless of whether the instances are correctly or incorrectly identified.

The results of a Mann-Whitney test for unpaired analyses indicate that the correctness of understanding is significantly greater when design-pattern instances are correctly identified with respect to when these instances are not correctly identified. This is true in all the experiments, the only exception being E4-UNiBZ on GD, where the statistical test returned a p-value equal to 0.39. It is also worth mentioning that the statistical power is always high when the Mann-Whitney test returns a p-value less than 0.05. The statistical power assumes values in-between 0.87 and 1. We also computed the effect size for GD and TD when pattern instances were correctly or not correctly recognized by participants. In particular, the effect size is small in E1-Prof on GD (Cliff's $d = 0.277$), while it is medium in E3-UniBAS on TD. In all the other experiments (except for E4-UNiBZ on GD), the effect size is large (values range between 0.432 and 0.623).

4.7.2. A Qualitative Look Inside the Use of Design-Pattern Instances. In this section, we discuss answers that the participants in E4-UniBZ gave to the two additional open questions presented in Section 3.9.1. The following are the main outcomes.

—The participants provided slightly different interpretations of why the documentation of pattern instances should make source-code comprehension and maintenance easier. However, there was consensus that instance documentation made software implementation more understandable (e.g., the link among classes can be found quickly) and that it implicitly clarifies the rationale behind the implementation. The participants did not find instance documentation to be useful as a communication mechanism among developers.
—Ten out of 15 participants stated that pattern instances should be graphically represented with UML class diagrams to improve source-code comprehensibility and

maintainability. This answer was given by 4 participants in the group SC and by 4 participants in the group TD, while it was given by 2 participants in the group GD. A possible justification is that the participants were familiar with the method of representing instances as proposed by Gamma et al. [1995] and thus, when instances were not documented in accordance with this method, they were less comfortable with comprehension tasks. When instances were documented in accordance with the method by Gamma et al. [1995], the participants believed that a different kind of documentation could better fit comprehensibility. We can conjecture that this finding might be due to how participants learned the basics of design-pattern development and not to the true benefits deriving from the use of one kind of documentation rather than another. This point might represent a future direction for our research.

4.8. Threats to Validity

To comprehend the strengths and limitations of our family of experiments, the threats that could affect results and their generalization are presented and discussed in this section. Despite our effort in mitigating as many threats as possible, some are unavoidable. We discuss the threats to validity using the guidelines proposed by Wohlin et al. [2012].

4.8.1. Internal Validity. In empirical studies, internal validity threats are relevant. They are even more so in a study like ours that tries to establish a causal relationship, because threats to internal validity may invalidate such a relationship.

—*Interaction with selection.* This has been mitigated by the experiment design. Each group of participants worked on one task only, with or without documented design-pattern instances. Furthermore, the participants in each experiment had similar background and experience.
—*Maturation.* The adopted experimental design should mitigate possible learning effects since participants worked on one task only. We have mitigated the fatigue effects allowing participants to take a break when passing from one group of questions in the comprehension questionnaire to the next.
—*Diffusion or imitation of treatments.* This threat concerns the information exchanged among participants within each experiment. We prevented this in several ways. While accomplishing each experiment, the participants were monitored by the experiment supervisors to prevent their communication with one another. The supervisors also ensured that no communication among participants occurred during each break. This point is less serious in E1-Prof, where at most two professionals were present in each laboratory session. The use of the Internet to exchange information might represent another bias. Participants could also use the Internet to get additional information on the experimental object. For example, the participants in group TD could have fetched from the Web the graphical representations of the pattern instances in the experimental object. To deal with threats related to the use of the Internet, we monitored participants while performing experimental tasks. As an additional measure to prevent the diffusion of material among participants, we asked participants to return material when each experiment was accomplished. This allowed reducing biases related to the communication among participants in different experiments.

4.8.2. External Validity. External validity primarily refers to the generalizability of the obtained results. Questions about external validity may arise in controlled experiments. However, this kind of empirical strategy is often conducted in the early steps of a long-term empirical investigation to reduce failure risks [Arisholm et al. 2006].

—*Interaction of selection and treatment.* The use of students as participants may affect external validity [Carver et al. 2003; Ciolkowski et al. 2004; Hannay and Jørgensen 2008]. To deal with this threat, we conducted replications with professionals and PhD students. Professionals were Italian junior/senior software developers. The majority of the professionals (15 out of 25) worked for companies located in Southern Italy. The remaining participants worked for companies located in Central (8) and Northern (2) Italy. Southern and Central Italy are over represented with respect to Northern Italy. This point might represent a threat to external validity. Regarding the size of the sample (i.e., number of professionals), it was hard to find a larger number of participants because of their limited availability and restrictions from their employers. PhD students were enrolled in an international PhD course in computer science. We were not able to find a larger number of PhD students because of their limited availability. As for the participants in E2-UniSA and E3-UniBAS, they were trained on UML and design-pattern development, and had taken several programming courses (e.g., procedural and object-oriented programming). In addition, some of the participants in E2-UniSA (i.e., about 40%) had an internship in industry as a final part of their Bachelor degree. However, working with students also has various advantages. For example, the students' prior knowledge was rather homogeneous [Verelst 2004]. To increase our confidence in the results, we are also promoting further independent/external replications by making a laboratory package available on the Web [Scanniello et al. 2014b]. This will allow researchers to easily conduct replications of our experiments [Lindvall et al. 2005]. External differentiated replications will further increase confidence in our results. These kinds of replication will reduce the likelihood of the researcher bias possibly present in our experimental procedure [Sjøberg et al. 2005].
—*Interaction of setting and treatment.* In our study, the size and complexity of the experimental object (i.e., documentation and system itself) may affect result validity. We selected a portion of an open-source software system large enough to be considered not excessively easy. Larger experimental objects could overload participants, thus biasing the experimental results. In addition, the source code of the experimental object was not familiar to participants, thus allowing to simulate a realistic comprehension task. Another possible threat to external validity related to the experimental object is that JHotDraw is well designed and design patterns are mostly intentionally used. This might make the experimental object not representative. Future work is needed to verify whether our results can be generalized in the case of poorly designed software systems developed without intentional use of design patterns. The use of UltraEdit might represent another threat to the validity of the results, as might the translation of the experimental material. However, it is worth mentioning that these two possible threats might affect the results for all three treatments (SC, GD, and TD). However, it is still possible that some of the participants were more familiar with UltraEdit than others, especially in E1-Prof and E4-UniBZ. Identifier renaming might also bias the results. To deal with this concern, source-code modifications were performed by one of the authors (the first), who was not involved in the definition of the comprehension questionnaire.

4.8.3. Construct Validity. Construct validity concerns the link between the concrete experimental elements and the abstract concepts of the experimental goal. Some construct validity threats are also related to the experiment design and to social factors.

—*Interaction of different treatments.* To mitigate this threat, we adopted a one-factor-with-three treatments design [Wohlin et al. 2012]. Each participant experimented wiht only one method on the same experimental object.

Table XI. Results of Cronbach's α Test for Questionnaires Reliability

Variable	E1-Prof	E2-UniSA	E3-UniBAS	E4-UniBZ
Source of Information	Acceptable (0.85)	Acceptable (0.95)	Acceptable (0.84)	–
Perceived Difficulty	Acceptable (0.86)	Acceptable (0.86)	Acceptable (0.9)	Acceptable (0.84)
Confidence	Acceptable (0.82)	Acceptable (0.79)	Acceptable (0.89)	Acceptable (0.83)

"–" signifies that the Cronbach's α test has not been computed because all the participants indicated source code (see also Figure 6(d)).

—*Confounding constructs and level of construct.* The procedure used to divide the participants into the groups shown in Table III could affect the validity of the obtained results. We are aware that the use of a different criterion to assess participants' working experience and ability could lead to different results. The fact that the participants in our family of experiments received the documentation in their native language (i.e., the participants in E1-Prof, E2-UniSA, and E3-UniBAS), while others did not (i.e., the participants in E4-UniBZ) might also affect the validity of results. Another threat is related to the fact that some of the participants received a training session on design-pattern development, while others did not. Professionals did not receive training for time constraints.

—*Evaluation apprehension.* We mitigated this threat because the participants were not evaluated on the results achieved in the experiments. All students in E2-UniSA and E3-UniBAS were equally rewarded with one extra point in the exam grade, regardless of their actual performance. All participants in our family of experiments were unaware of the objectives of our family of experiments and of our experimental hypotheses.

—*Experimenters' expectancies*. Questions in the comprehension questionnaire were formulated in order to favor none among groups GD, TD, or SC. The reliability of the comprehension questionnaires was tested by applying the Cronbach's α test [Cronbach 1951]. In Table XI, we summarize the results obtained for each set of the closed questions intended to measure the three subjective variables (i.e., source of information, perceived difficulty, and confidence). All the obtained values were higher than the acceptable minimum threshold [Maxwell 2002] ($\alpha = 0.70$). The post-experiment survey questionnaire was designed to capture the participants' perception of the tasks. We designed this questionnaire using standard methods and scales [Oppenheim 1992].

—*Inadequate pre-operational explication of construct*. The constructs were not sufficiently defined before they were translated into measures or treatments. To deal with this kind of threat to construct validity, we took advantage of the framework we introduced in Section 2.

4.8.4. Conclusion Validity. Conclusion validity concerns issues that may affect the ability of drawing a correct conclusion.

—*Reliability of measures.* This threat is related to how the dependent variables were measured. The method to assess correctness of understanding is widely used in experiments with purposes similar to ours (e.g., Ricca et al. [2010]). In addition, one of the authors (i.e., the second) not involved in the task definition built the comprehension questionnaire and analyzed the participants' answers to this questionnaire. Regarding the second considered dependent variable, we asked the participants to write their start-and stop-times on their questionnaire. We qualitatively validated the provided information. Even if this method of measuring the task completion time can be considered questionable [Hochstein et al. 2005], this practice is very common [Huang and Holcombe 2009; Ceccato et al. 2014; Scanniello et al. 2014a].

—*Random heterogeneity of participants.* We drew fair samples and conducted our experiments with participants belonging to these samples. Regarding practitioners, their experience could be heterogeneous.
—*Fishing and the error rate.* Our hypotheses have been rejected considering proper p-values.
—*Statistical tests.* We opted for nonparametric tests (e.g., Mann-Whitney test for unpaired analyses) to statistically reject the defined null hypotheses. We used a non parametric test also when a parametric test could be applied. This choice might increase the risk of Type-II errors.

5. DISCUSSION

We first discuss the results of each experiment individually and then together. The section is concluded with some practical implications from our family of experiments.

5.1. Individual Experiments

We observed a clear improvement in correctness of understanding deriving from the use of documented instances. This finding holds for all experiments in the family, with the only exception of E3-UniBAS, namely that experiment having participants with the lowest level of experience. For E1-Prof and E4-UniBZ, the difference in correctness of understanding is statistically significant. The participants in these experiments were those with the highest level of experience. On the basis of the observed results, we can conclude that: if the maintainer has an adequate level of experience with design-pattern development and computer programming, source-code comprehensibility is statistically greater when instances are textually or graphically documented as opposed to not having these instances documented at all. As for E2-UniSA, the average improvement achieved by the participants with graphically documented instances as opposed to those having instances not documented at all was 30.1%. The average improvement in comprehension was 38.7% when instances were textually documented. In other words, source-code comprehensibility is greater, but not statistically significant, when design-pattern instances are textually or graphically documented rather than with respect to not having them documented at all. This implies that a maintainer benefits from the documentation of design-pattern instances also in the case where he/she has a Bachelor degree and knows the basics of high- and low-level design of object-oriented software systems, software development, and software maintenance.

The presence of documented instances did not significantly affect the time to accomplish comprehension tasks. Indeed, the participants spent slightly less time in accomplishing a task when the implemented instances were documented either textually or graphically. This result might be considered unexpected because having more documents/information to read and interpret should require more time to execute a task. This is even more evident for graphically documented instances. A justification for this result is that explicitly reported instances allow finding implementation details quicker, and this reduces the time to comprehend source code.

In all the experiments, we observed that the average values for correctness of understanding were higher when participants were able to correctly identify those pattern instances needed to answer the comprehension questionnaire. In many cases (7 out of 8), we observed that correctness of understanding is significantly greater when design-pattern instances are correctly identified (see Table X). This finding suggests it is important to help the maintainer to easily recognize pattern instances in source code because this would improve source-code comprehensibility.

As for E1-Prof, we observed that the participants in the group GD indicated pattern instances as the first source of information. In contrast, the participants in the group TD indicated source code as the first source of information. Independently from the group,

source code is the first source of information in all the other experiments of our family. This slight difference in the results achieved by the participants in the groups GD and TD could be due to the fact that, in the latter case, pattern instances are documented in source code. Then, some participants could have considered documented pattern instances as an integral part of source code.

5.2. Participants' Experience

Participants with the greater experience had a more correct understanding of source code. In addition, more experienced participants benefit more than less experienced ones when instances are documented either textually or graphically. As for time, we observed no noteworthy data pattern.

We observed that those participants who used textually documented instances and with a given experience (i.e., at least a Bachelor degree) obtained, on average, better results with respect to those participants who used graphically documented instances (see Table V). The effect size is also in favor of TD. It is negligible in E2-UniSA, small in E1-Prof, and large in E4-UniBZ. In the case of the most experienced participants (E1-Prof and E4-UniBZ), we also found that the average time to accomplish comprehension tasks is lower for TD than for SC and GD. These outcomes suggest that those participants who accomplished comprehension tasks with TD achieved an improved comprehension of source code with less effort. This is even more interesting if we compare TD with SC: the participants analyzed more information in less time and obtained an improved correctness of understanding of the source code. As for E2-UniSA, the participants who accomplished comprehension tasks with GD achieved improved comprehension of source code with greater effort with respect to those who accomplished tasks with TD. The outcomes suggest that a given experience is needed to benefit from the documentation of design-pattern instances in the comprehension of source code and to reduce the average time to accomplish comprehension tasks when this kind of documentation is given in addition to source code.

5.3. Implications

To assess the implications of our family of experiments, we adopted a perspective-based approach [Basili et al. 1996]. In particular, we focus here on the practitioner/consultant (simply practitioner, in the following) and researcher's perspectives [Kitchenham et al. 2008].

—The use of textually or graphically documented instances improves the correctness of understanding of source code, given that the maintainer has an adequate level of experience. This result is clearly relevant from the researcher's perspective. For example, the researcher could be interested in studying how source-code comprehensibility is affected when using together both textually and graphically documented pattern instances. The practitioner could also be interested in our results because it is worth documenting pattern instances to get an improved source-code comprehension. However, this outcome opens a managerial dilemma: are the additional effort and cost to create and maintain the documentation of pattern instances adequately paid back by improved source-code comprehensibility? Indeed, from a managerial point of view, the adoption of graphical and textual documentation as a means to represent pattern instances should take into account the costs it will introduce. Furthermore, what method is less expensive in representing design-pattern instances? On the basis of our findings, these points represent future directions for research.
—The presence of documented instances does not seem to be a cause of distraction for the maintainer (Hn1_Time was not rejected in all the experiments) while performing comprehension tasks. In each experiment, the use of textual and graphical instances

reduces the average time to comprehend source code. This result is obviously relevant for the practitioner, but also for the researcher. In particular, the researcher could be interested in investigating how to better support the maintainer in quickly finding implementation details from the pattern instances to reduce the effort to comprehend source code.

—Benefits deriving from the documentation of pattern instances appear dependent on the participants' experience. This result is practically relevant for the professional: maintainers with different experiences shall benefit from the documented instances to a different degree. This aspect is clearly relevant also for the researcher, who could be interested in studying the possible experience threshold to benefit from the use of instance documentation in source-code comprehension. From the researcher's perspective, this outcome also adds an empirical voice to the debate over the use of students as valid proxies in developer-based studies [Carver et al. 2003; Höst et al. 2000; Arisholm and Sjøberg 2004]. That is, can students be considered as appropriate as professionals in controlled experiments? On the basis of our results, it seems that students can if they have sufficient knowledge of the technology/method that is being evaluated. Therefore, the use of students might be an issue as long as we are interested in evaluating the adoption of a technology/method by expert software engineers and if they do not have a sufficient knowledge of such technology/method. On the other hand, the use of students might be valuable when our goal is to study ease of use of either a technology/method and its possible benefit without or with limited training or experience (e.g., Erra and Scanniello [2010], and Scanniello and Erra [2014]).

—The experiment with PhD students replicates the findings of the original experiment. This implication is related to the implication in the prior bullet-point. In fact, PhD students can be as appropriate as professionals in controlled experiments. This implication is relevant for the researcher because it suggests that PhD students might substitute for professionals in developer-based studies. Although this point deserves future investigation, we can postulate that our findings pose the basis for future work in this direction.

—Comprehensibility improves when pattern instances are correctly recognized in source code. This opens the following future research directions: (i) investigating issues that led to certain patterns being better comprehended and recognized than others, (ii) defining new methods/notations to make the identification of pattern instances in source code easier, and (iii) investigating source-code comprehensibility when graphically documented instances are complemented by other notations (e.g., UML sequence diagrams or textual documentation). The results from our family of experiments delineate these future research directions.

—Since comprehensibility improves when source code is complemented by textually or graphically documented instances, the number of defects should decrease correspondingly as a consequence of a maintenance operation. This is true provided that the maintainer has an adequate level of experience (i.e., at least a Bachelor degree). The use of documented instances should also positively impact the communication among developers. These implications are relevant for the practitioner.

—The majority of the participants found the source code to be the most relevant source of information to perform comprehension tasks. This finding is relevant for the researcher who could be interested in assessing whether and how this concern affects the benefits stemming from the use of documented pattern instances.

—We considered pattern instances to be graphically documented using UML class diagrams. For each instance, we exploited the same graphical layout as proposed by Gamma et al. [1995]. The use of different layouts might lead to different results. This aspect is relevant for the practitioner interested in understanding the best

way for representing pattern instances, and for the researcher interested in investigating how and why a different layout for a pattern instance affects source-code comprehensibility.

—The diffusion of a new technology/method is made easier when empirical evaluations are performed and their results show that such a technology/method solves actual issues [Baldassarre et al. 2014; Pfleeger and Menezes 2000]. Therefore, the results from our family of experiments could increase the practice of documenting pattern instances in the software industry. In addition, even if those costs related to software development, maintenance, and evolution might increase, the introduction or the enforcement of the systematical use of documented instances should not require a complete and radical process change in a given company. This point has particular interest for the practitioner.

6. RELATED WORK

Software maintenance is essential in the evolution of software systems and represents one of the most expensive, time-consuming, and challenging phases of the whole development process. Maintenance starts after the delivery of the first version of the system and lasts much longer than the initial development process [Bennett and Rajlich 2000; Zelkowitz et al. 1979]. As shown in the survey by Erlikh [2000], the cost needed to perform maintenance operations ranges from 85% to 90% of the total cost of a software project. Whatever the maintenance operation, the greater part of its cost and effort is due to the comprehension of source code [Mayrhauser 1995]. In particular, Pfleeger and Atlee [2006] estimated that up to 60% of software maintenance is spent on comprehension. There are several reasons that make comprehension even more costly and complex, namely the size of software system and its available documentation [Selfridge et al. 1993].

Only few studies have been conducted to assess the support that the documentation of pattern instances provides to software maintenance [Prechelt et al. 2002; Gravino et al. 2011]. For example, Prechelt et al. [2002] studied whether pattern instances textually documented in source code (through comments) improve maintainers' performance in the execution of maintenance tasks with respect to well-commented source code without explicit references to pattern instances. The study involved 74 German graduate students and 22 undergraduate students from the USA, who performed tasks on small Java and C++ programs, respectively. For example, the size of these programs ranged from 495 to 762 lines of code (comment lines included), only two kinds of pattern instances were present in each program, and maintenance operations were executed on sheet papers. The results suggested that maintenance tasks supported by explicitly documented pattern instances were completed faster or with fewer errors. This is the study most similar to ours. However, there are many differences that make the study by Prechelt et al. [2002] different from the one we present in this article. The main difference is that we conducted a family of controlled experiments involving participants having four different levels of experience: from Bachelor students to software professionals. In addition, we analyzed the effect of graphically documented pattern instances, we used a larger experimental object from a real open-source software system (2149 lines of code, comment lines included, with 10 pattern instances implemented), and the participants conducted their task inside a more realistic environment, thus reducing threats to external validity. The results achieved in our long-term investigation together with those from Prechelt et al. [2002] demonstrate the usefulness of explicitly documenting instances when dealing with source-code comprehension. Therefore, the main "take away" lesson from our family of experiments and from the two experiments by Prechelt et al. [2001, 2002] is that documenting instances can improve source-code comprehensibility. That is, it is not only important to use design patterns to develop

an object-oriented software system, but so even more important to properly document the instances implemented in its source code. These results are perhaps not overly surprising, but are acceptable if not significant, as evidence needs to be verified/reaffirmed through empirical studies [Basili et al. 1999; Kitchenham 2008; Kitchenham et al. 2002; Shull et al. 2008].

The research presented in this article is also different from other investigations conducted to examine software design patterns [Prechelt et al. 2001; Bieman et al. 2003; Cepeda Porras and Guéhéneuc 2010; Jeanmart et al. 2009; Khomh and Guéhéneuc 2008; di Penta et al. 2008; Vokác 2004; Vokác et al. 2004; Krein et al. 2011], since we pursued a different goal. In particular, the focus here is on the kind of documentation for pattern instances and on source-code comprehensibility. For example, Prechelt et al. [2001] investigate whether the use of design patterns is useful or harmful. To this end, their experiment involved four different, rather small, C++ programs and six software design patterns. Each of these programs existed in two versions: one based on design patterns and the other implementing a simplified design without pattern instances. This empirical investigation was next replicated by Vokác et al. [2004] and Krein et al. [2011]. All the investigations reinforce the conclusion that each kind of design pattern has its own nature and proper place of use. The results achieved in these investigations also suggest that some kinds of design patterns are much easier to understand and use than others. The implication is that design patterns are not universally good or bad they must be used in a way that matches the problem and maintainer. In this respect, our work complements these investigations by explicitly considering the source-code comprehension that four kinds of maintainers achieved when source code is developed exploiting design patterns and when the instances implemented in the source code are textually and graphically documented.

7. CONCLUSION

In this work, we have presented results from a family of four controlled experiments carried out with students and practitioners. We used controlled experiments because a number of confounding and uncontrollable factors might be present in real project settings. The achieved results suggest that at least a given experience is needed to benefit from the documentation of design-pattern instances as complementary information to comprehend source code. For professionals and PhD students, the use of graphically and textually documented pattern instances significantly improves correctness of understanding of source code. We did not observe any significant difference between textually and graphically documented pattern instances with respect to both correctness of understanding and task completion time. Our results also suggest that correctness of understanding improves when pattern instances are correctly identified. It seems crucial to ease the identification of instances in source code.

Possible future directions for our research are: (i) performing different kinds of empirical investigations (e.g., case studies); (ii) investigating the effect of documentation for specific design patterns on source-code comprehensibility; (iii) studying new representations for documenting pattern instances; and (iv) investigating whether providing textual documentation alongside the graphical improves source-code comprehensibility.

ELECTRONIC APPENDIX

The electronic appendix to this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors would like to thank all the participants in the family of experiments.

REFERENCES

S. M. Abrahao, C. Gravino, E. I. Pelozo, G. Scanniello, and G. Tortora. 2013. Assessing effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Trans. Softw. Engin.* 39, 3.

C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. 1977. *A Pattern Language - Towns, Buildings, Construction*. Oxford University Press.

J. Aranda, N. Ernst, J. Horkoff, and S. Easterbrook. 2007. A framework for empirical evaluation of model comprehensibility. In *Proceedings of the International Workshop on Modeling in Software Engineering (MISE'07)*. IEEE Computer Society, 7–13.

E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. 2006. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Trans. Softw. Engin.* 32, 6, 365–381.

E. Arisholm, H. Gallis, T. Dyba, and D. I. K. Sjøberg. 2007. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Trans. Softw. Engin.* 33, 2, 65–86.

E. Arisholm and D. I. K. Sjøberg. 2004. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Trans. Softw. Engin.* 30, 8, 521–534.

H. U. Asuncion, F. Francois, and R. N. Taylor. 2007. An end-to-end industrial software traceability tool. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*. ACM Press, New York, 115–124.

R. Baker. 1995. Modern permutation test software. In *Randomization Tests*, Marcel Dekker, 391–402.

M. T. Baldassarre, J. Carver, O. Dieste, and N. Juristo. 2014. Replication types: Towards a shared taxonomy. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*. ACM Press, New York, 18:1–18:4.

V. Basili, F. Shull, and F. Lanubile. 1999. Building knowledge through families of experiments. *IEEE Trans. Softw. Engin.* 25, 4, 456–473.

V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, L. S. Sørumgard, and M. V. Zelkowitz. 1996. The empirical investigation of perspective-based reading. *Empir. Softw. Engin.* 1, 2, 133–164.

V. R. Basili and H. D. Rombach. 1988. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. Softw. Engin.* 14, 6, 758–773.

M. I. Bauer and P. N. Johnson-laird. 1993. How diagrams can improve reasoning. *Psychol. Sci.* 4, 372–378.

K. H. Bennett and V. T. Rajlich. 2000. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering (ICSE'00)*. ACM Press, New York, 73–87.

G. Bergersen, D. I. K. Sjøberg, and T. Dyba. 2014. Construction and validation of an instrument for measuring programming skill. *IEEE Trans. Softw. Engin.* 40, 12, 1163–1184.

G. R. Bergersen, J. E. Hannay, D. I. K. Sjøberg, T. Dyba, and A. Karahasanovic. 2011. Inferring skill from tests of programming performance: Combining time and quality. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'11)*. IEEE Computer Society, 305–314.

J. Bieman, G. Straw, H. Wang, P. Munger, and R. Alexander. 2003. Design patterns and change proneness: An examination of five evolving systems. In *Proceedings of the 9th International Software Metrics Symposium (METRIC'03)*. IEEE Computer Society, 40–49.

B. Bruegge and A. H. Dutoit. 2003. *Object-Oriented Software Engineering: Using UML, Patterns and Java*. 2nd Ed. Prentice-Hall.

D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. 2011. Empirical evidence about the UML: A systematic literature review. *Softw. Pract. Exper.* 41, 4, 363–392.

J. Carver, L. Jaccheri, S. Morasca, and F. Shull. 2003. Issues in using students in empirical studies in software engineering education. In *Proceedings of the 9th International Symposium on Software Metrics (METRIC'03)*. IEEE Computer Society, 239–250.

M. Ceccato, M. di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. 2014. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empir. Softw. Engin.* 19, 4, 1040–1074.

G. Cepeda Porras and Y.-G. Gueheneuc. 2010. An empirical study on the efficiency of different design pattern representations in UML class diagrams. *Empir. Softw. Engin.* 15, 5, 493–522.

M. Ciolkowski, D. Muthig, and J. Rech. 2004. Using academic courses for empirical validation of software development processes. In *Proceedings of the EUROMICRO Conference (EUROMICRO'04)*. IEEE Computer Society, 354–361.

J. Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences*. L. Erlbaum Associates.

M. Colosimo, A. De Lucia, G. Scanniello, and G. Tortora. 2009. Evaluating legacy system migration technologies through empirical studies. *Inf. Softw. Technol.* 51, 2, 433–447.

W. J. Conover. 1998. *Practical Nonparametric Statistics*. 3rd Ed. Wiley.

L. Cronbach. 1951. Coefficient alpha and the internal structure of tests. *Psychometrika* 16, 3, 297–334.

F. Q. B. Da silva, M. Suassuna, A, A. C. C. Franc, A. M. Grubb, T. B. Gouveia, C. V. F. Monteiro, and I. E. Dos Santos. 2014. Replication of empirical studies in software engineering research: A systematic mapping study. *Empir. Softw. Engin.* 19, 3, 501–557.

M. Di Penta, L. Cerulo, Y.-G. Gueheneuc, and G. Antoniol. 2008. An empirical study of the relationships between design pattern roles and class change proneness. In *Proceedings of the International Conference on Software Maintenance (ICSM'08)*. IEEE Computer Society, 217–226.

T. Dyba, V. B. Kampenes, and D. I. K. Sjøberg. 2006. A systematic review of statistical power in software engineering experiments. *Inf. Softw. Technol.* 48, 8, 745–755.

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

L. Erlikh. 2000. Leveraging legacy system dollars for e-business. *IT Profess.* 2, 17–23.

U. Erra and G. Scanniello. 2010. Assessing communication media richness in requirements negotiation. *IET Softw.* 4, 2, 134–148.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.

O. S. Gomez, N. Juristo, and S. Vegas. 2010. Replications types in experimental disciplines. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'10)*. ACM Press, New York, 3:1–3:10.

C. Gravino, M. Risi, G. Scanniello, and G. Tortora. 2011. Does the documentation of design pattern instances impact on source code comprehension? Results from two controlled experiments. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'11)*. IEEE Computer Society, 67–76.

C. Gravino, M. Risi, G. Scanniello, and G. Tortora. 2012. Do professional developers benefit from design pattern documentation? A replication in the context of source code comprehension. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12)*. Springer, 185–201.

Y.-G. Gueheneuc. 2007. P-mart: Pattern-like micro architecture repository. In *Proceedings of the EuroPLoP Focus Group on Pattern Repositories (EuroPLoP'07)*.

Y.-G. Gueheneuc and G. Antoniol. 2008. Demima: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Engin.* 34, 5, 667–684.

J. Hannay and M. Jørgensen. 2008. The role of deliberate artificial design elements in software engineering experiments. *IEEE Trans. Softw. Engin.* 34, 2, 242–259.

J. Heer and M. Agrawala 2006. Software design patterns for information visualization. *IEEE Trans. Visual. Comput. Graph.* 12, 853–860.

L. Hochstein, V. R. Basili, M. V. Zelkowitz, J. K. Hollingsworth, and J. Carver. 2005. Combining self-reported and automatic data to improve programming effort measurement. *ACM SIGSOFT Softw. Engin. Notes* 30, 5, 356–365.

M. Host, B. Regnell, and C. Wohlin. 2000. Using students as subjects—A comparative study of students and professionals in lead-time impact assessment. *Empir. Softw. Engin.* 5, 3, 201–214.

L. Huang and M. Holcombe. 2009. Empirical investigation towards the effectiveness of test first programming. *Inf. Softw. Technol.* 51, 1, 182–194.

ISO. 1991. Information technology–Software product evaluation: Quality characteristics and guidelines for their use. ISO/IEC IS 9126. ISO, Geneva.

S. Jeanmart, Y.-G. Gueheneuc, H. Sahraoui, and N. Habra. 2009. Impact of the visitor pattern on program comprehension and maintenance. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09)*. IEEE Computer Society, 69–78.

A. Jedlitschka, M. Ciolkowski, and D. Pfahl. 2008. Reporting experiments in software engineering. In *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjoberg, Eds., Springer, 201–228.

N. Juristo and A. Moreno 2001. *Basics of Software Engineering Experimentation*. Kluwer Academic.

V. Kampenes, T. Dyba, J. Hannay, and I. Sjoberg 2007. A systematic review of effect size in software engineering experiments. *Inf. Softw. Technol.* 49, 11–12, 1073–1086.

F. Khomh and Y.-G. Gueheneuc. 2008. Do design patterns impact software quality positively? In *Proceedings of the 12th European Conference on Software Engineering and Maintenance (CSMR'08)*. 274–278.

B. Kitchenham. 2008. The role of replications in empirical software engineering - A word of warning. *Empir. Softw. Engin.* 13, 2, 219–221.

B. Kitchenham, H. Al-khilidar, M. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu. 2008. Evaluating guidelines for reporting empirical software engineering studies. *Empir. Softw. Engin.* 13, 97–121.

B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El emam, and J. Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Engin.* 28, 8, 721–734.

J. L. Krein, L. J. Pratt, A. B. Swenson, A. C. Maclean, C. D. Knutson, and D. L. Eggett. 2011. Design patterns in software maintenance: An experiment replication at Brigham Young University. In *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research (RESER'11).* 25–34.

M. Lindvall, I. Rus, F. Shull, M. V. Zelkowitz, P. Donzelli, A. M. Memon , V. R. Basili, P. Costa, R. T. Tvedt, L. Hochstein, S. Asgari, C. Ackermann, and D. Pech. 2005. An evolutionary testbed for software technology evaluation. *Innovat. Syst. Softw. Engin.* 1, 1, 3–11.

M. Lindvall, and K. Sandahl 1996. Practical implications of traceability. *Softw. Pract. Exper.* 26, 10, 1161–1180.

K. Maxwell. 2002. *Applied Statistics for Software Managers*. Software Quality Institute Series, Prentice Hall.

A. V. Mayrhauser. 1995. Program comprehension during software maintenance and evolution. *IEEE Comput.* 28, 44–55.

J. Mcdermid. 1991. *Software Engineer's Reference Book*. Butterworth-Heinemann, Linacre House, Jordan Hill, Oxford, UK.

A, M. G. Mendonc, J. C. Maldonado, M. C. F. D. Oliveira, J. Carver, S. C. P. F. Fabbri, F. Shull, G. H. Travassos, E. N. Hohn, and V. R. Basili . 2008. A framework for software engineering experimental replications. In *Proceedings of the 13th International Conference on Engineering of Complex Computer Systems (ICECCS'08).* IEEE Computer Society, 203–212.

OMG. 2005. Unified modeling language (UML) specification, version 2.0. Tech. rep., Object Management Group. http://www.omg.org/spec/UML/2.0/.

A. N. Oppenheim. 1992. *Questionnaire Design, Interviewing and Attitude Measurement.* Pinter, London.

S. Pfleeger and J. Atlee 2006. *Software Engineering - Theory and Practice*. 3rd Ed. Ellis Horwood.

S. L. Pfleeger and W. Menezes. 2000. Marketing technology to software practitioners. *IEEE Softw.* 17, 1, 27–33.

L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta. 2001. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Trans. Softw. Engin.* 27, 12, 1134–1144.

L. Prechelt, B. Unger-lamprecht, M. Philippsen, and W. Tichy. 2002. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Softw. Engin.* 28, 6, 595–606.

F. Ricca, M. Di penta, M. Torchiano, P. Tonella, and M. Ceccato. 2010. How developers' experience and ability influence Web application comprehension tasks supported by UML stereotypes: A series of four experiments. *IEEE Trans. Softw. Engin.* 36, 1, 96–118.

F. Ricca, G. Scanniello, M. Torchiano, G. Reggio, and E. Astesiano. 2014. Assessing the effect of screen mockups on the comprehension of functional requirements. *ACM Trans. Softw. Engin. Methodol.* 24, 1, 1:1–1:38.

G. Salton and M. J. Mcgill. 1983. *Introduction to Modern Information Retrieval*. McGraw Hill, New York.

M. Scaife and Y. Rogers. 1996. External cognition: How do graphical representations work? *Int. J. Human-Comput. Stud.* 45, 2, 185–213.

G. Scanniello and U. Erra. 2014. Distributed modeling of use case diagrams with a method based on think-pair-square: Results from two controlled experiments. *J. Vis. Lang. Comput.* 25, 4, 494–517.

G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-lemus, and G. Tortora. 2014a. On the impact of UML analysis models on source code comprehensibility and modifiability. *ACM Trans. Softw. Engin. Methodol.* 23, 2.

G. Scanniello, C. Gravino, M. Risi, G. Tortora, and G. Dodero. 2014b. Design pattern instances comprehension. http://www2.unibas.it/gscanniello/DP/.

G. Scanniello, C. Gravino, and G. Tortora 2013. An early investigation on the contribution of class and sequence diagrams in source code comprehension. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13).* IEEE Computer Society, 367–370.

P. Selfridge, R. Waters, and E. Chikofsky. 1993. Challenges to the field of reverse engineering. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'93).* IEEE Computer Society, 144-150.

S. Shapiro and M. Wilk 1965. An analysis of variance test for normality. *Biometrika* 52, 3–4, 591–611.

F. Shull, J. C. Carver, S. Vegas, and N. J. Juzgado. 2008. The role of replications in empirical software engineering. *Empir. Softw. Engin.* 13, 2, 211–218.

F. Shull, M. G. Mendoncca, V. Basili, J. Carver, J. C. Maldonado, S. Fabbri, G. H. Travassos, and M. C. Ferreira. 2004. Knowledge-sharing issues in experimental software engineering. *Empir. Softw. Engin.* 9, 1–2, 111–137.

J. Sillito, G. C. Murphy, and K. De volder. 2008. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Engin.* 34, 4, 434–451.

D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. Liborg, and A. C. Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE Trans. Softw. Engin.* 31, 9, 733–753.

S. Vegas, N. Juristo, A. Moreno, M. Solari, and P. Letelier. 2006. Analysis of the influence of communication between researchers on experiment replication. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06).* ACM, New York, 28–37.

J. Verelst. 2004. The influence of the level of abstraction on the evolvability of conceptual models of information systems. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'04).* IEEE Computer Society, 17–26.

M. Vokac. 2004. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Softw. Engin.* 30, 12, 904–917.

M. Vokac, W. F. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin. 2004. A controlled experiment comparing the maintainability of programs designed with and without design patterns- A replication in a real programming environment. *Empir. Softw. Engin.* 9, 3, 149–195.

C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A Wesslen. 2012. *Experimentation in Software Engineering*. Springer.

M. V. Zelkowitz, A. C. Shaw, and J. D. Gannon. 1979. *Principles of Software Engineering and Design*. Prentice-Hall.

Received March 2014; revised November 2014; accepted December 2014