Empir Software Eng (2015) 20:1126–1158
DOI 10.1007/s10664-014-9317-9

The impact of imperfect change rules on framework API evolution identification: an empirical study

Wei Wu · Adrien Serveaux · Yann-Gaël Guéhéneuc · Giuliano Antoniol

Published online: 22 July 2014

Abstract Software frameworks keep evolving. It is often time-consuming for developers to keep their client code up-to-date. Not all frameworks have documentation about the upgrading process. Many approaches have been proposed to ease the impact of non-documented framework evolution on developers by identifying change rules between two releases of a framework, but these change rules are imperfect, i.e., not 100 % correct. To the best of our knowledge, there is no empirical study to show the usefulness of these imperfect change rules. Therefore, we design and conduct an experiment to evaluate their impact. In the experiment, the subjects must find the replacements of 21 missing methods in the new releases of three open-source frameworks with the help of (1) all-correct, (2) imperfect, and (3) no change rules. The statistical analysis results show that the precision of the replacements found by the subjects with the three sets of change rules are significantly different. The precision with all-correct change rules is the highest while that with no change rules is the lowest, while imperfect change rules give a precision in between. The effect size of the difference between the subjects with no and imperfect change rules is large and that between the subjects with imperfect and correct change rules is moderate. The results of this study show that the change rules generated by framework API evolution approaches do help developers, even they are not always correct. The imperfect change rules can be used by developers upgrading their code when documentation is not available or as a complement to

Communicated by: Laurie Williams

W. Wu (✉) · A. Serveaux · Y.-G. Guéhéneuc
Ptidej Team, DGIGL, École Polytechnique de Montréal, Montréal, Canada
e-mail: wei.wu@polymtl.ca

A. Serveaux
e-mail: adrien.serveaux@telecom-st-etienne.fr

Y.-G. Guéhéneuc
e-mail: yann-gael.gueheneuc@polymtl.ca

W. Wu · G. Antoniol
Soccer Lab, DGIGL, École Polytechnique de Montréal, Montréal, Canada

G. Antoniol
e-mail: giuliano.antoniol@polymtl.ca

partial documentation. The moderate difference between results from subjects with imperfect and all-correct change rules also suggests that improving precision of change rules will still help developers.

Keywords Software maintenance · Usefulness · Framework API evolution · Change rule

1 Introduction

Software frameworks and libraries are widely used in software development to reduce costs. (Without loss of generality, we use the term “framework” to mean both frameworks and libraries in this article.) Indeed, developers evolve frameworks constantly to fix bugs, patch security vulnerabilities, and meet new requirements. In theory, the Application Programming Interface (API) of the new release of a framework should be *backward-compatible* with its previous releases, so that programs linked to the framework continue to work with the new release (German and Hassan 2009).

Yet, upgrading to new releases of frameworks requires significant effort due to changes to the APIs. Developers must dig into the documents and the source code of the new and previous releases of the frameworks to understand their differences and to make their programs compatible with the new releases. The longer they delay the upgrading, the more time consuming is upgrading, because there will be more changes to absorb.

To adapt the API changes in new releases of frameworks, developers usually perform four tasks: (1) find possible replacements of missing APIs, (2) find work-arounds for missing APIs if their replacements cannot be found, (3) implement the replacements or the work-arounds and, (4) test the implementation. Finding replacements for missing APIs (or identifying them as simply deleted, i.e., without replacements) avoids looking for unnecessary work-arounds and–or trying wrong replacements, thus reducing the overall time to adapt to new frameworks.

An effective way to help find the proper replacements is documentation. However, many frameworks are not sufficiently documented, especially when it comes to the changes between releases and the rules to adapt programs from an older release to a new one. The Java programming language provides the `@deprecated` annotation to help framework developers mark API changes, but developers do not always use these annotations to document how to update to the new releases. As examples, Google provides Android API difference reports[1] regularly but these reports only list the API removed or added, but do not include the information on how to replace the removed APIs. Eclipse informs developers about internal APIs and official APIs with Provisional API Guidelines,[2] using Javadoc annotations to describe changes and, sometimes, replacements to deprecated methods. By the definition, internal APIs can be changed without notice and documentation. However, a survey conducted by Businge et al. showed that 70 % of the developers of Eclipse-related projects used Eclipse internal APIs and only 3.3 % of them always followed Eclipse Provisional API Guidelines (Businge et al. 2013). Other ad hoc forms of upgrading documents include, for example, dedicated tutorials, such as the one helping Python devel-

[1] http://developer.android.com/sdk/api_diff/8/changes.html

[2] http://wiki.eclipse.org/Provisional_API_Guidelines#Before_the_API_freeze

opers to replace calls to the `os.system()` with functions provided by the `subprocess` module.[3] In general, very few companies explicitly document changes in APIs or provide such information to the public.

Therefore, many approaches have been developed to provide replacements for missing APIs. Some approaches require that the framework developers do additional work, such as providing explicit upgrade rules with annotations (Chow and Notkin 1996) or that they record API updates to the framework (Dig et al. 2007; Henkel and Diwan 2005; Kemper and Overbeck 2005). However, framework developers may not be able or willing to build change rules manually or use specific tools. Thus, to avoid the extra work for framework developers, other approaches automatically identify *change rules* that describe a matching between *target methods*, i.e. methods existing in the old release of a framework but not in the new one, and *replacement methods* in the new release (Antoniol et al. 2004; Dagenais and Robillard 2011; Demeyer et al. 2000; Dig et al. 2006; Fluri and Gall 2006; Godfrey and Zou 2005; Kataoka et al. 2001; Kim et al. 2005, 2007; Malpohl et al. 2000; Meng et al. 2012; Schäfer et al. 2008; Weißgerber and Diehl 2006; Wu et al. 2010; Xing and Stroulia 2006). The change rules generated by these approaches are imperfect or not all of them are correct. Their precision varies in the frameworks analyzed.

However, with the change rules generated by these approaches as upgrading documents, developers do not know if the change rules are correct until having used them to modify their client programs. Based on their understanding of the frameworks, if they believe that the change rules are correct, they will modify their code accordingly and test if the replacements indeed fix the errors and provide the expected behavior. For the missing methods whose change rules that they believe are incorrect or whose replacements do not pass the tests, developers must explore the documents or source code of the releases of the framework to search for the replacements manually.

Zhang and Budgen (2012) stated that "Software design is widely recognised as being a "wicked" or "ill-structured" problem, characterised by ambiguous specifications, no true/false solutions (only ones that are "better" or "worse" from a particular perspective), the lack of any "stopping rule" to determine when a solution has been reached, and no ultimate test of whether a solution meets the requirements." Therefore, there are no strict standards to assess the effectiveness of many software engineering activities but empirical studies. In the context of framework API evolution, to the best of our knowledge, there is no empirical study of the usefulness of the imperfect change rules (generated by tools or from other sources) to show that the imperfect change rules help developers to identify the replacements more accurately and faster than without change rules or, rather, that they confuse developers because they are not all correct. Although we could expect that using already-known all-correct change rules would help developers, it could actually slow them down in the case which developers would not be certain that the change rules are correct. Indeed, according to Fagar (1996), providing no information is actually better than providing the wrong information: it is less confusing and distracting. Knowing the usefulness of imperfect change rules could encourage and direct research on framework API evolution.

Therefore, we design and conduct an experiment to evaluate the usefulness of framework API evolution change rules. In the experiment, the subjects find the replacements of target methods with the help of all-correct, imperfect, and no change rules. Then, we measure the performance of the subjects by the precision of the replacement methods that they find and

[3] http://docs.python.org/2/library/os.html#os.system

the time that they spend. To limit the influence of a specific program on the results, we ask the subjects to analyze three medium size programs (JHotDraw v5.2–v5.3, JFreeChart v0.9.11–v0.9.12, and JEdit v4.1–v4.2). To limit the experiment time to about one hour, we did a pre-experiment evaluation of the required time and found that analyzing seven changed APIs in each program can meet the experiment time requirement. Thus, we randomly select seven change rules for each program. Among them, five correct rules and two incorrect change rules in the imperfect change rules of JEdit v4.1–v4.2 and six correct rules and one incorrect change rule for JFreeChart v0.9.11–v0.9.12 and JHotDraw v5.2–v5.3. The precision of the imperfect change rules is similar to that of the state-of-the-art approaches to framework API evolution.

We use a randomized, complete block design (Wohlin et al. 1999) in our experiment to minimize the number of subjects required and to lessen some threats to validity, discussed in Section 6. Under this design, each subject performs experiments with all the three programs and the three sets of change rules, but the orders of the combinations of programs and change rule sets are randomized. In total, 31 subjects participate in the experiment. The statistical analysis shows that the precision of the replacements of target methods found by the subjects with all-correct, imperfect, and no change rules are significantly different with average values of 82 %, 71 %, and 57 %, respectively. The effect size (Cliff's Delta) of the difference in precision between the subjects with no and imperfect change rules is large and that between the subjects with imperfect and all-correct change rules is moderate. Different from the precision values, the time that the subjects with the three treatments spend to find the replacement methods is not statistically different with average values of 24, 23, and 25 minutes (1,413, 1,338, and 1,479 seconds), respectively.

These results are evidence that change rules generated by framework API evolution approaches are useful, even when some of the change rules are incorrect. Yet, as expected, the higher precision the change rules have, the more help they provide. Thus, the imperfect change rules can be used as replacement for unavailable documentation or as complement to partial documentation. Developers of frameworks could also use them as starting point to build upgrading documentation. The difference between results between subjects with imperfect and all-correct change rules is moderate. Therefore, improving the precision of change rules will still help developer.

In the reminder of the paper, Section 2 introduces previous work. In Section 3, we describe the design and the execution of the experiment. The analysis results are presented in Section 4. Section 5 discusses related issues. Threats to validity are discussed in Section 6. Conclusions and future work are presented in Section 7.

2 Related Work

We now discuss the related works. First, we present approaches to frameworks API evolution, then introduce previous related empirical studies in software engineering.

2.1 Framework API Evolution Approaches

Several approaches help developers evolve their programs when the frameworks that they use change. We now discuss the differences between these approaches and ours according to the formulation presented in this paper.

2.1.1 Inputs

Existing approaches of capturing API-level changes require the framework developers to manually enter the change rules or to use a particular IDE to automatically record the changes. Chow and Notkin (1996) presented a method that requires the framework developers to provide change rules with the new releases. CatchUP! (Henkel and Diwan 2005) and JBuilder (Kemper and Overbeck 2005) record the refactoring operations in one release and replay them in another. MolhadoRef (Dig et al. 2007) also employs a record-and-replay technique for handling API-level changes in merging program releases. These approaches can provide accurate change rules because of the framework developers' involvement, which might not always be available.

Dagenais and Robillard's SemDiff (2011) and HiMa of Meng et al. (2012) use software repository commits. Diff-CatchUp, developed by Xing and Stroulia (2007), uses the models of logical design of two releases of a program as input. Others approaches (Godfrey and Zou 2005; Kim et al. 2005, 2007; Schäfer et al. 2008; Wu et al. 2010) take the source code of evolved frameworks as input. Schäfer et al. (2008)'s approach also uses the client programs code as a part of its input.

2.1.2 Features

The features used by the approaches to framework API evolution are the more specific information extracted from the inputs, such as call-dependency relations or text similarity.

The features used by the approaches of capturing API-level changes (Chow and Notkin 1996; Henkel and Diwan 2005; Kemper and Overbeck 2005; Dig et al. 2007) are different presentations of the change rules manually added or automatically captured. These approaches have a specific model for each target method and its replacement.

Godfrey and Zou's (2005) and S. Kim et al.'s (2005) approaches use text similarity, software metrics, and call dependency relations to describe target methods and their replacements. Xing and Stroulia (2007) extract the differences between two releases of the logical design models using lexical and structural similarity, including text similarity, inheritance relations, usage dependencies, and association relations. Kim et al.'s (2007) compute LCS of the target methods and its replacement to measure the difference between them. SemDiff (Dagenais and Robillard 2011), Schäfer et al. (2008), and AURA (Wu et al. 2010) use call-dependency relations measured by confidence value and various presentations of text similarity. HiMa (Meng et al. 2012) uses call-dependency relations and natural-language-analysis-filtered comments of consecutive commits.

2.1.3 Matching Techniques

The approaches of capturing API-level changes only need simple matching techniques to match the collected change rules with the target methods, but the developers' involvement that they require might not be available.

The matching techniques of Godfrey and Zou's (2005) and Kim et al.'s (2005) approaches are based on origin analysis techniques. The former is semi-automatic while the latter is automatic. Diff-CatchUp (Xing and Stroulia 2007) defined three sets of heuristics for class, method, and fields, respectively to order the possible replacement methods. SemDiff (Dagenais and Robillard 2011) and Schäfer et al. (2008) first use confidence values to preselect the possible replacement methods, then use text similarity to rank them.

The difference between them is that the former focuses on how frameworks adapt to their own changes while the latter discovers the usage changes from client code. Kim et al.'s (2007) classifier leverages systematic renaming patterns using text similarity to match old APIs to new APIs. AURA (Wu et al. 2010) uses a multi-iteration algorithm combining call-dependency and text similarity analyses. HiMa's (Meng et al. 2012) generates initial change rules using processed revision comments, then expends and refines them using call-dependency analysis.

2.2 Empirical Study on Program Comprehension

Another field related to this work is the empirical studies on program comprehension. Lawrie et al.'s conducted an empirical study to investigate the influence of identifiers on program comprehension (Lawrie et al. 2007). Sharif et al. (2010) and Sharafi et al. (2012) use eye-tracking systems to compare camel-case and underscore identifier styles. The former focuses on accuracy and speed when developers recognize identifiers while the latter studies identifier comprehension from the point of view of genders. Some studies focus on diagram comprehension, such as UML. Cepeda and Guéhéneuc evaluated the impact of three visual presentations on design pattern comprehension with an eye-tracking system. Yusuf et al. conducted an experiment to study the characteristics of UML class diagrams that affect program comprehension (Yusuf et al. 2007). Soh et al. performed an experiment to assess the role of professional status and expertise on UML class diagram comprehension. There are also other empirical studies related to program comprehension. Abbes et al. performed an experiment to evaluate the impact of two anti-patterns (Blob and Spaghetti Code) on program comprehension (Abbes et al. 2011). Ali et al. use eye-tracking system to rank developer's preferred entities, such as class names, method names (Ali et al. 2012).

2.3 Summary

To the best of our knowledge, there is no empirical study to show the usefulness of the change rules generated by approaches to framework API evolution, i.e., if the imperfect change rules can help developers to identify the replacement methods more accurately or faster than without them, despite the popularity of empirical studies in software engineering. Thus, we designed and conducted this study.

3 Experiment

The goal of our experiment is to verify that the imperfect change rules help developers to find the replacements of target methods more accurately or faster than without such them. The usefulness of imperfect change rules describing API evolutions is the quality focus of our experiment and its perspective is to help developers to decide if they should use imperfect change rules when they upgrade their client programs to new releases of frameworks, when documentation is not available, complete, or up-to-date. The context of our experiment is finding the replacements of 21 target methods of three frameworks by 31 subjects with three treatments: (1) all-correct, (2) imperfect, and (3) no change rules. The correctness of the change rules is unknown to the subjects. We follow the guidelines described by Wohlin et al. (1999) to present our experiment.

3.1 Research Questions

We expect that the results of the experiment can answer the two research questions:

- RQ_1: Is there a difference between the precision of the replacements of the target methods found by the subjects with all-correct, imperfect, and no change rules?
- RQ_2: Is there a difference between the time that the subjects spend to find replacements with all-correct, imperfect, and no change rules?

In the experiment, although we do not give any time constraint to perform the tasks, we cannot expect all subjects to be able or willing to finish all of them. Therefore, we take both the time spent and the precision of the answers into account. The differences between our experiment and real context are discussed in Section 5.5.

3.2 Objects

The objects of our experiment are the source code of two releases of three frameworks written in Java. We choose three medium size programs as objects of our experiment. These three programs are among those that we and other researchers have analyzed: JEdit v4.1–v4.2, JFreeChart v0.9.11–v0.9.12, and JHotDraw v5.2–v5.3. JEdit is a text editor.[4] It has 37 releases between 2000 and 2013. Its API and implementation between its v4.1 and 4.2 changed dramatically. JFreeChart is a chart library.[5] There are 54 release between 2000 to 2013. The APIs between its v0.9.11 and v0.9.12 also changed a lot and there are many APIs in v0.9.12 with very similar names. The dramatic changes in the two programs are challenging for both developers and the approaches to identify API changes. JHotDraw is a GUI framework[6] developed by Gamma et al. to demonstrate the application of design patterns (Gamma et al. 1995). It is less active compared to the former two programs with only 12 releases between 2001 and 2011. Yet, between v5.2 and v5.3, they are several API changes. Previous approaches to identify API changes have all very high precision when applied on these two versions of JHotDraw.

We use these programs for four reasons. First, Java is supported by most framework API evolution approaches (Dagenais and Robillard 2011; Kim et al. 2007; Meng et al. 2012; Schäfer et al. 2008; Wu et al. 2010; Xing and Stroulia 2006). Second, by using programs with which we are familiar, we can more easily identify correct and imperfect change rules and better evaluate the results of the subjects on the tasks in the experiment. Third, these programs have different characteristics regarding framework API evolution. We believe that they are representative of typical API evolution patterns. Fourth, these programs are of medium sizes with numbers of lines of code ranging from 9,441 to 64,710, as shown in Table 1: the subjects do not have to spend too much time to explore them.

The source code of the three programs is provided to subjects in an Eclipse workspace. We also recommend the subjects to explore the source code with Eclipse. They are available online.[7] We present change rules and collect subjects' answers using a dedicated Web site (see Section 3.9).

[4] http://www.jedit.org

[5] http://www.jfree.org/jfreechart

[6] http://www.jhotdraw.org

[7] http://www.ptidej.net/download/experiments/emse13a

Table 1 Object systems

Frameworks	Releases	# Methods	# SLOC
JHotDraw	5.2	1,486	9,441
	5.3	2,265	14,612
JEdit	4.1	2,773	46,176
	4.2	3,547	59,804
JFreeChart	0.9.11	4,751	59,060
	0.9.12	5,197	64,710

3.3 Tasks

We also consider the difference between experiment and real context, when we design the tasks for the subjects. While finding replacements of missing methods, developers can use the compiler and their tests to verify if a replacement is correct. We do not provide client code in the experiment for two reasons. First, the client code could have helped the subjects to verify if the replacements were correct, but it cannot help developers find the correct replacements. Second, there would have been extra effort for the subjects to make the client code executable using the replacements. For example, if there was a new parameter in a replacement, to find out the proper value for that parameter would have taken time too. Such effort cannot be saved by change rules and we excluded it from our experiment.

Consequently, we design our experiment tasks as program comprehension to verify if the imperfect change rules can help subjects locate the replacements faster or more accurately than with all-correct or no rules. We measure both the precision of the answers of the subjects and the time that they spend. So, incorrect answers do not prevent us to evaluate subjects' performance.

We select a set of target methods from each framework and ask the subjects to find their replacements in the source code of the new release with all-correct, imperfect, and no change rules. For example, a subject is given a target method from JEdit v4.1, `void org.gjt.sp.jedit.gui.FloatingWindowContainer.save(org.gjt.sp.jedit.gui.DockableWindowManager.Entry)`, which does not exist in v4.2. She searches for its replacement by going through the source code of JEdit v4.1 and v4.2. According to our design, she may be provided with correct, incorrect, or no replacements methods, depending on her treatment. The subject does not know if the change rule is correct or not, therefore she must verify it by understanding the source code. In the end, she may or may not find that the actual replacement is `org.gjt.sp.jedit.GUIUtilities.saveGeometry(java.awt.Window,java.lang.String)`. We measure her performance by the precision of the replacement methods that she found and the time that she spent on the task.

3.4 Subjects

The subjects are volunteer software practitioners familiar with Java. Among the total 31 subjects, nine are B.Sc., four are M.Sc., and 16 are Ph.D. students in computer science or software engineering, two subjects are professional developers working in software development companies; 10 subjects are female and 21 are male. Their task distribution is shown in Table 18 in the end of the article.

3.5 Independent Variable

The independent variable of our experiment is the kind of set of change rules provided to subjects. We have three treatments:

- TR_c: All-correct change rules.
- TR_i: Imperfect change rules.
- TR_n: No change rule.

A change rule is the mapping between a target method and their replacements in the new release of a framework. It can be correct or incorrect if the replacement methods in the change rule can replace the target method or not. A set of imperfect change rules, TR_i, represents a set of change rules mixing correct and incorrect replacement methods. Usually, the change rules generated by framework API evolution approaches are imperfect. A set of all-correct change rules, TR_c, includes change rules manually verified to be correct, i.e., to provide correct replacement method(s) to a given target methods. We do not inform subjects that the change rules are imperfect or all-correct before performing the experiment. Because the change rules are randomly selected, we do not control the relations between the target methods and the type of change rules.

To build TR_c, TR_i, and TR_n, we first generate the change rules of the object systems with AURA (Wu et al. 2010). We choose AURA, because it is a state-of-the-art approach to framework API evolution. Our experiment studies the influence of all-correct, imperfect, or no change rules - using other tools or even building change rules manually do not affect the results of the study. Second, we randomly select five correct rules and two incorrect change rules for JEdit v4.1–v4.2 and six correct rules and one incorrect change rule for JFreeChart v0.9.11–v0.9.12 and JHotDraw v5.2–v5.3 to build the imperfect change rules, TR_i. The precision of TR_i is close to that of the state-of-the-art approaches to framework API evolution. Third, we manually correct the incorrect change rules to build TR_c and remove all the replacements from the change rules in TR_i to build TR_n. To limit the time of the experiment approximately to one hour, we perform a pre-experiment and choose 21 change rules, seven from each program.

3.6 Dependent Variable

The dependent variables of our experiment are the precision of the replacements that the subjects identify for the given target methods and the time that they spend on the task. We compute the precision using the equation below. For each target method, we compare the subjects' answers, i.e., given set of methods, against the manually-validated, expected set of replacement methods. Time is counted in seconds when subjects work on the tasks. Break time, if there were any, were excluded.

$$Precision = \frac{|\{CorrectReplacements\}|}{|\{GivenTargetMethods\}|}$$

3.7 Mitigating Variables

We collected several mitigating variables that could influence the results of our experiment:

- Subjects' knowledge of the object programs;
- Subjects' knowledge of Java;
- Subjects' knowledge of Eclipse;

- Subjects' experience in software engineering;
- Learning effect;
- Fatigue;
- Experiment environment.

For the first four mitigating variables, we asked the subjects to fill a questionnaire to provide their levels of knowledge on five-point Likert scales (Likert 1932) (bad, quite good, good, excellent or expert). Their knowledge levels are show in Table 2. We provide a tutorial about the functions related to our experiment to minimize the influence of the knowledge of Eclipse. We verify if there is any correlation between the dependent variables and the four mitigating variables during the result analysis. We limit learning effect and fatigue with a randomized complete block design (Wohlin et al. 1999). To minimize the influence of the environment, we require that the subjects complete the tasks in a quiet environment.

3.8 Hypotheses

The null hypotheses of our experiment are that there is no difference between the precision of the replacements of the target methods found by the subjects and the time that they spend with the help of all-correct, imperfect, and no change rules. For example:

- H_{p1}: There is no difference between the precision of the replacements of the target methods found by the subjects find without change rules and with imperfect change rules.

The completed null hypotheses are in Table 3.

3.9 Experiment Web Site

To conduct the experiment and abstract the presentation of the change rules from a particular tool format, we develop a dedicated Web site. The screenshots of the main Web pages are shown in Fig. 1. Page 1(a) is the paper where the subjects fill in background information. Page 1(b) is the main task page. For each target method, the page displays the target method on the left (e.g., `TextFigure.disconnect()`) and a set of candidate replacement methods, i.e., obtained from a change rule, on the right-bottom corner (e.g., `TextFigure.disconnect(Figure)`). The right-top box is for subjects to enter their answers.

The gathering of information from the subjects, such as background information and experiment question answers, occurs in a Web browser. All the data that subjects enter and the time that they spend on each question are stored in a database. The subjects do not need

Table 2 Subjects' knowledge levels

KnowledgeLevel	Topics					
	SE	Java	Eclipse	JHotdraw	JEdit	JFreechart
Bad	0	1	1	21	22	20
Quite Good	1	5	10	7	7	6
Good	18	19	16	3	2	3
Excellent	8	5	3	0	0	2
Expert	4	1	1	0	0	0

Table 3 Null hypotheses

H_{p1}	There is no	precision of the	without change rule	and	with imperfect change rules.
H_{p2}		replacement methods	without change rule	and	with correct change rules.
H_{p3}	difference	found by the subjects	with imperfect change rules	and	with correct change rules.
H_{t1}		time spent	without change rule	and	with imperfect change rules.
H_{t2}	between the		without change rule	and	with correct change rules.
H_{t3}		by the subjects	with imperfect change rules	and	with correct change rules.

paper or writing. The experiment Web site is accessible online[8] and its source code is also available online.[9]

3.10 Experiment Workspace

Besides using the experiment Web site to read the questions, the target and suggested replacement methods (if any), and to enter their answers, the subjects had also to explore the source code of the object programs to find the replacement methods. We provided an Eclipse workspace to each subject with the source code of JEdit v4.1–v4.2, JFreeChart v0.9.11–v0.9.12, and JHotDraw v5.2–v5.3, as shown in Fig. 2 and available online[9]. With Eclipse, the subjects could check the definitions and usage context of the target methods, read the source code of both versions of the source code of the frameworks, and identify the replacements.

3.11 Experiment Process

For each program, the subjects had to find the replacements for the seven target methods with all-correct, imperfect, or no change rules.

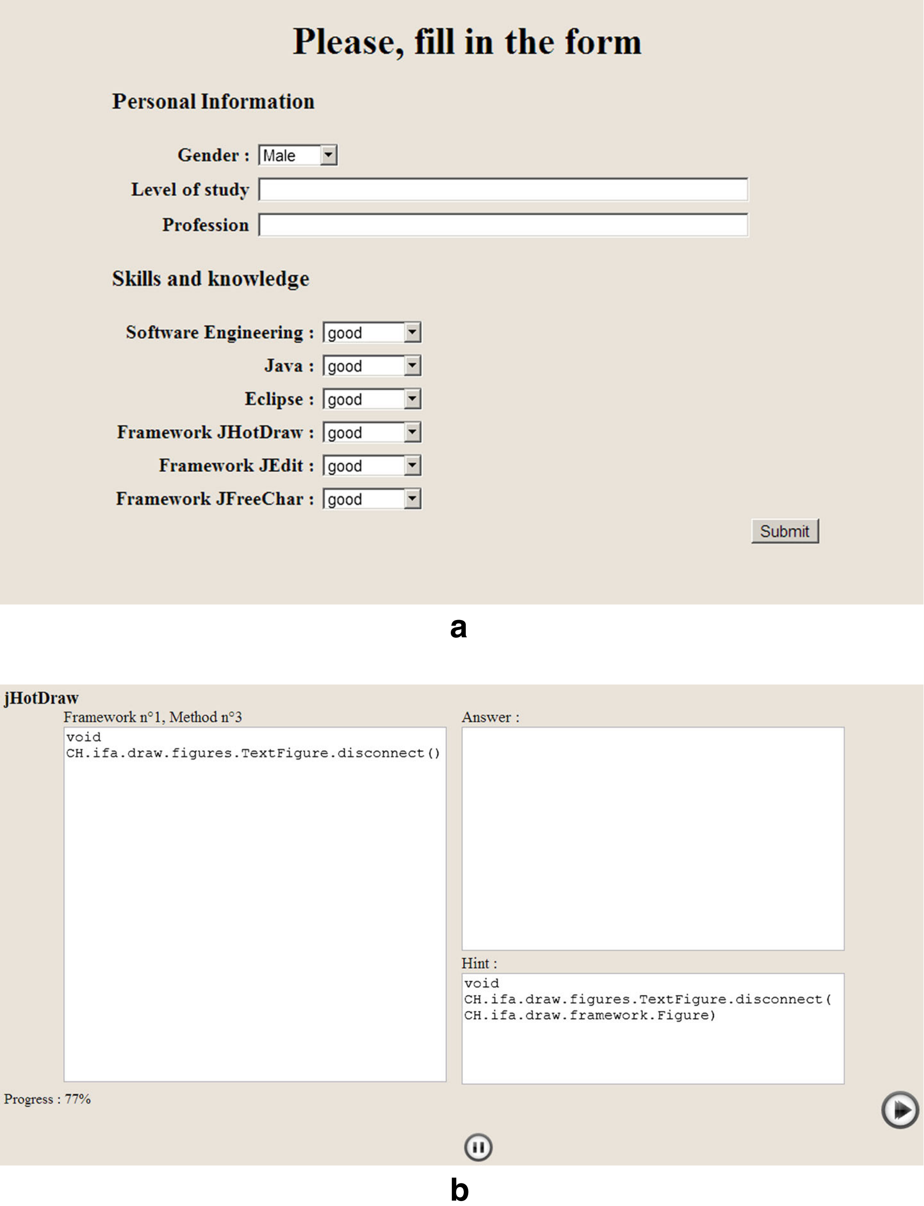
At the beginning of the experiment, each subject received two tutorials in PDF format. One tutorial explains the procedure of the experiment without revealing the purpose of our study: what subjects should do and how they fill in the answers of the questions. The other tutorial was about the use Eclipse to explore the source code of the frameworks, in case some subjects were not familiar with it. We asked the subjects to read these two documents carefully before the experiment. If the subjects had any questions, they could ask us in person if they performed the experiment in our laboratory or by email if they did it remotely.

When the subjects were familiarized with the experiment and Eclipse, we administered a pre-experiment questionnaire to collect their gender, level of study, profession, knowledge in software engineering, Eclipse, Java, and the frameworks used in the study, using the background information page of our Web site.

Then, they completed the tasks by exploring the source code of the frameworks, looking for the replacement methods of each target method displayed on the question pages. According to the treatment, they had either all-correct or imperfect replacements methods or no method at all to help them. When they found the replacement method(s), they copied the qualified names of the replacement methods from Eclipse or from the box at the bottom-right corner of the question pages and pasted them into the answer box on each question page. Eclipse provides a context menu to copy the qualified names of methods easily and

[8] http://web.soccerlab.polymtl.ca/serveaua/experiment/ with password: ptidejexp

[9] http://www.ptidej.net/download/experiments/emse13a



Please, fill in the form

Personal Information

Gender : Male

Level of study

Profession

Skills and knowledge

Software Engineering : good

Java : good

Eclipse : good

Framework JHotDraw : good

Framework JEdit : good

Framework JFreeChar : good

jHotDraw

Framework n°1, Method n°3

```
void
CH.ifa.draw.figures.TextFigure.disconnect()
```

Answer :

Hint :

```
void
CH.ifa.draw.figures.TextFigure.disconnect(
CH.ifa.draw.framework.Figure)
```

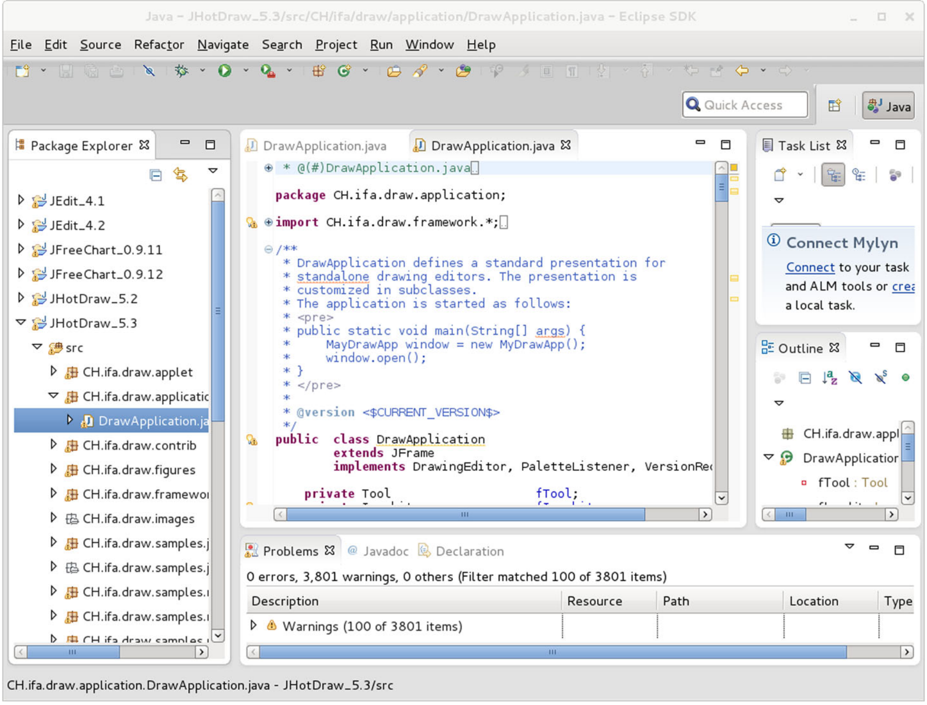
Progress : 77%

Fig. 1 Web site

our tutorial explained how to use this feature. If the subjects thought that the target method was simply deleted, i.e., without any replacement, they could fill in "null" in the answer box.

3.12 Analysis Method

We compared the subjects' answers with the all-correct rules and classified those as wrong if they were different from the all-correct rules.



```
 * @(#)DrawApplication.java

package CH.ifa.draw.application;

import CH.ifa.draw.framework.*;

/**
 * DrawApplication defines a standard presentation for
 * standalone drawing editors. The presentation is
 * customized in subclasses.
 * The application is started as follows:
 * <pre>
 * public static void main(String[] args) {
 *     MayDrawApp window = new MyDrawApp();
 *     window.open();
 * }
 * </pre>
 *
 * @version <$CURRENT_VERSION$>
 */
public  class DrawApplication
        extends JFrame
        implements DrawingEditor, PaletteListener, VersionRe

    private Tool                    fTool;
```

0 errors, 3,801 warnings, 0 others (Filter matched 100 of 3801 items)

Description	Resource	Path	Location	Type
Warnings (100 of 3801 items)				

CH.ifa.draw.application.DrawApplication.java - JHotDraw_5.3/src

Fig. 2 Source code workspace

We tested our hypotheses using Kruskal-Wallis test (Wohlin et al. 1999), which is applicable for non-parametric randomized, complete block designs. The hypotheses testing results are presented as Table 4. We chose Kruskal-Wallis test because we did not have to make any assumption on the distribution of the data collected in the experiment. The regular α value of single comparison Kruskal-Wallis test was 0.05. Our experiment had three object programs for each treatment. Therefore, we adjusted the α value to 0.01 according to the Bonferroni correction (Miller 1981). If the p-value of Kruskal-Wallis tests was smaller than 0.01, the results with different treatments were statistically different. We also

Table 4 Hypothesis testing results for precision

Treatements	Kruskal-Wallis *p*-value	Cliff's Delta
No-Imperfect-Correct	< 0.01 (4.496e-06)	N/A
No-Imperfect	< 0.01 (0.006293)	Large (0.619)
No-Correct	< 0.01 (5.269e-06)	Large (1.325)
Imperfect-Correct	< 0.01 (0.002763)	Moderate (0.443)

computed the Cliff's Delta d (Grissom and Kim 2005) of the results with different treatments to evaluate the effect size of their differences. Cliff's Delta is also a non-parametric and it does not require any knowledge of the distribution of the data. The effect size is small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$, respectively. We computed p-values and Cliff's Deltas of the results on all three frameworks and each framework separately to distinguish the potential effect of one particular framework.

We always applied the statistical tests on the average precision values of the subject's answers and average times spent by the subjects for subjects using two different treatments, for example subjects provided with all-correct change rules and subjects provided with no change rules. In the next section, for the sake of simplicity, we talk about precision and time to mean average precision values and average time spent by subjects.

4 Result Analysis

We now report the collected data and discuss the data analysis and the results of hypothesis testing. First, we report the general results on the precision of the subjects' answers and the time that they spent. Then, we discuss the results on the three frameworks separately.

4.1 Overall Data Analysis

Precision The distribution of the data on precision is shown in Fig. 3. On average, the precision values of the answers of the subjects with all-correct change rules is the highest with the value of 82 %. The next is that with imperfect change rules with the value of 71 %. The precision of the answers without change rule is the lowest with the value of 57 %. In Table 4, the hypothesis testing results show that there are statistically significant differences in the precision of the subjects' answers when they used all-correct, imperfect, and no change rules. In the third column, the value of Cliff Delta is moderate between the subjects with imperfect and all-correct rules and it is large between those with imperfect and no change rules. These results support the expectation that even imperfect change rules help developer find the correct replacements.

Therefore, we conclude that the subjects with all-correct change rules could find the replacements of the target methods more accurately than the subjects with the other two treatments. The subjects with imperfect change rules did not perform as well as the former, but significantly better than the subjects without change rule. However, even with all-correct change rules, the subjects still could not correctly answer all the questions. This last observation is evidence of the difficulty of framework API evolution. We further discuss this observation in Section 5.

Time Figure 4 shows the distribution of the time that subjects spent with the tree treatments. The subjects spent almost the same time to find the replacement methods with all-correct, imperfect, or no change rules. The subjects with imperfect change rules spent less time than the subjects with the other two treatments. On average, they spent 23 minutes (1,338 seconds) while the subjects with all-correct change rules and without change rule used 24 minutes (1,413 seconds) and 25 minutes (1,479 seconds), respectively. However, the hypothesis testing results do not confirm any statistically significance difference between the times spent with each treatment, as shown in Table 5.

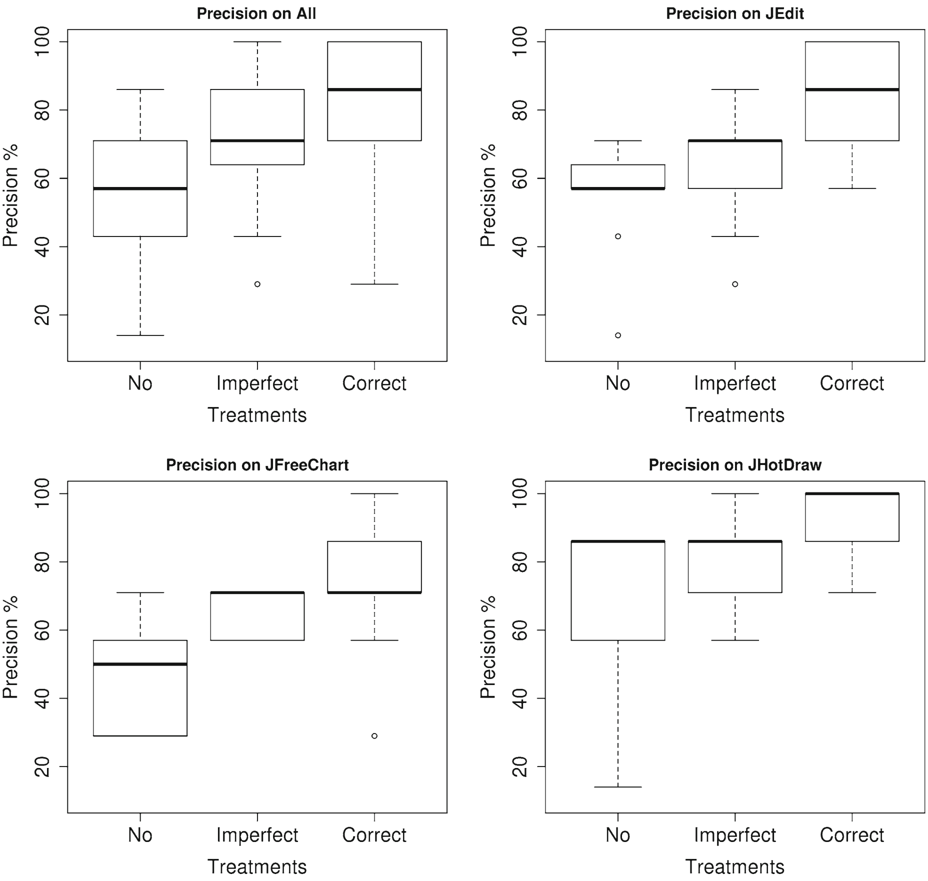


Fig. 3 Boxplots for precision

Discussion We examined the outliers in precision and time. There were two subjects with precision of 14 % on JHotDraw and JEdit, respectively, and three spent more than 46 minutes (2,800 seconds) on a single program: one on JHotdraw and two on JFreechart. There was no common subject between the two cases in precision and three cases in time. The results of these five subjects were "normal" when they worked on the other programs. Four of the five cases were on the first program that they analyzed in the experiment. We suspect that these outlier cases were caused by these subjects being not really familiar with the experiment tasks and tools. The other outlier case (more time on JFreechart) was on the third program. Probably, the subject productivity was compromised by tiredness.

4.2 Data Analysis per System

The results presented above show that the change rules helped the subjects to find the replacements of the target methods more accurately but not faster than without them. We want now to investigate if this conclusion applies to all three frameworks. Therefore, we also compare the results on different framework individually to see if the change rules helped similarly.

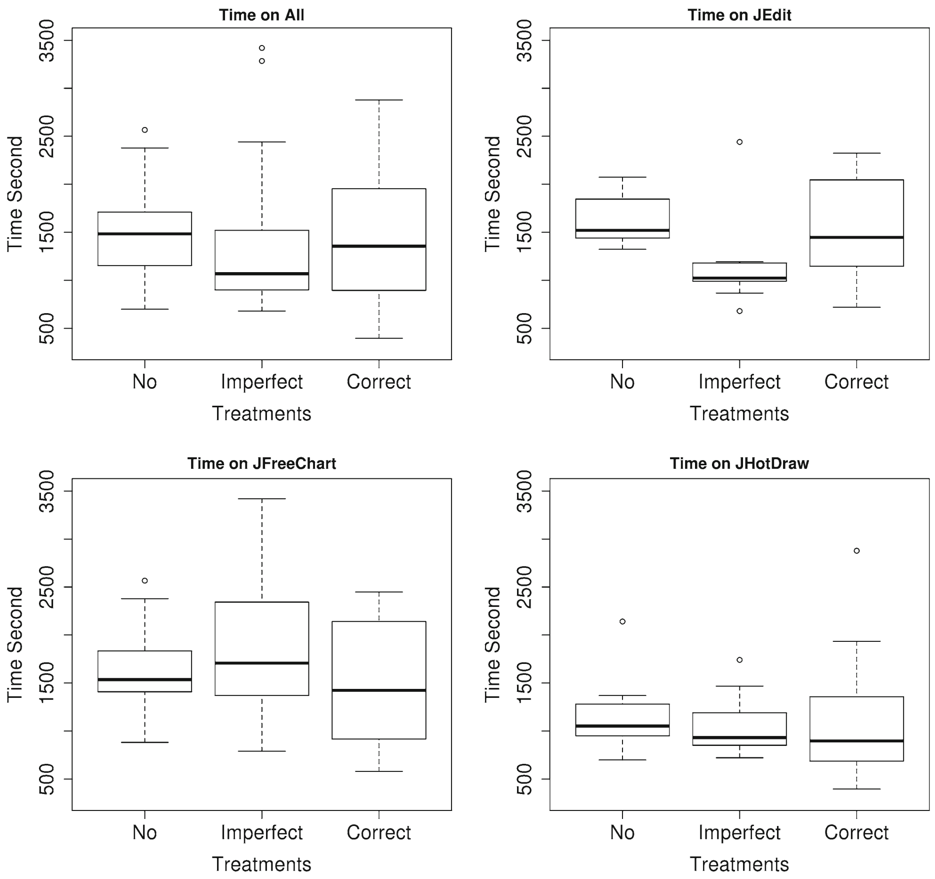


Fig. 4 Boxplots for time

4.2.1 *JHotDraw*

Precision For JHotDraw, the boxplot in Fig. 3 shows that the precision of the subjects' answers with imperfect change rules was better than that of the subjects without change rule and the precision of the subjects answers with all-correct change rules was the best, but the differences between the precision of subjects' answers with the three treatments are not statistically significant, according to the hypothesis testing results in Table 6.

Time Regarding the time that subjects spent on JHotDraw, Fig. 4 shows that the average values of the subjects' time with the three treatments were slightly different. Similar to the general result on time, the hypothesis testing results (Table 7) show that the differences are not statistically significant. The change rules did not help to find the replacement methods faster or slower, while adapting to the new release of JHotDraw.

Discussion JHotDraw is a program developed by Gamma et al. to demonstrate the application of design patterns (Gamma et al. 1995). It was elegantly designed and consistently

Table 5 Hypothesis testing results for time

Treatements	Kruskal-Wallis *p*-value	Cliff's Delta
No-Imperfect-Correct	> 0.01 (0.1378)	N/A
No-Imperfect	> 0.01 (0.02611)	N/A
No-Correct	> 0.01 (0.3348)	N/A
Imperfect-Correct	> 0.01 (0.6272)	N/A

coded. When we studied its v5.2 and v5.3, we found that it is very straightforward to navigate and understand its source code. Between JHotDraw v5.2 and v5.3, the change rules helped subjects, but it did not take much more time for subjects to complete the task without them. Indeed, the hypothesis testing results on JHotDraw show that there is no statistically significant differences between the subjects with the three treatments.

4.2.2 JFreeChart

Precision The change rules helped subjects to find replacements for the target methods of JFreeChart more accurately. The boxplot in Fig. 3 shows that the precision of the subjects' answers with correct change rules was the best and the precision of the subjects' answers with imperfect change rules is better than that without change rule. There is much less overlap between the precision values of the subjects' answers with the three treatments than that on JHotDraw and JEdit. The hypothesis testing results confirm that the differences between the subjects' answers with no vs. all-correct and no vs. imperfect change rules are statistically significant with large effect size and the difference between the subjects' answers with imperfect and all-correct change rules is not significant as shown in Table 8.

Time Similar to JHotDraw, Fig. 4 shows that the average values of the times spent by the subjects with the three treatments are only slightly different. The subjects with imperfect change rules have the largest value while the subjects with all-correct change rules have

Table 6 Hypothesis testing results for precision on JHotDraw

Treatements	Kruskal-Wallis *p*-value	Cliff's Delta
No-Imperfect-Correct	> 0.01 (0.01738)	N/A
No-Imperfect	> 0.01 (0.5376)	N/A
No-Correct	> 0.01 (0.01008)	N/A
Imperfect-Correct	> 0.01 (0.02132)	N/A

Table 7 Hypothesis testing results for time on JHotDraw

Treatements	Kruskal-Wallis *p*-value	Cliff's Delta
No-Imperfect-Correct	> 0.01 (0.3632)	N/A
No-Imperfect	> 0.01 (0.2353)	N/A
No-Correct	> 0.01 (0.2207)	N/A
Imperfect-Correct	> 0.01 (0.7223)	N/A

the smallest. The hypothesis testing results in Table 9 show that there are no significant differences. The change rules did not help save time to find the replacement methods while upgrading to the new release of JFreeChart.

Discussion JFreeChart is a Java chart library to generate different types of charts. Between v0.9.11 and v0.9.12, there were many methods with similar names in the two versions, such as `DefaultBoxAndWhiskerCategoryDataset.getMedianValue(int, int)` and `DefaultBoxAndWhiskerXYDataset.getMedianValue(int, int)`. It would be time-consuming to go over all of these similar methods to find possible replacement methods and verify them. Therefore, with the change rules, subjects used much less time to find the replacements than those without them.

Because of the differences between a real context and our experiment, the subjects without change rules spent almost the same time as the subjects with imperfect and all-correct change rules, but with lower precision in their answers. Although we asked them to take as much time as they needed, it seems that they just spent the same effort as the subjects with the other treatments and did not verify their answers thoroughly. The hypothesis testing results show that the precision of the subjects' answers with imperfect and all-correct change rules are statistically better than that without change rules, while the time that they spent is not significantly different.

Table 8 Hypothesis testing results for precision on JFreeChart

Treatements	Kruskal-Wallis *p*-value	Cliff's Delta
No-Imperfect-Correct	< 0.01 (0.001069)	N/A
No-Imperfect	< 0.01 (0.004354)	Large (1.60919)
No-Correct	< 0.01 (0.001702)	Large (1.80291)
Imperfect-Correct	> 0.01 (0.08238)	N/A

Table 9 Hypothesis testing results for time on JFreeChart

Treatements	Kruskal-Wallis *p*-value	Cliff's Delta
No-Imperfect-Correct	> 0.01 (0.5209)	N/A
No-Imperfect	> 0.01 (0.5676)	N/A
No-Correct	> 0.01 (0.4288)	N/A
Imperfect-Correct	> 0.01 (0.3198)	N/A

4.2.3 JEdit

Precision Similar to the precision for JFreeChart, the boxplot in Fig. 3 shows that the average value of precision of the subjects's answers with imperfect change rules is better than that of the subjects without change rule and that the precision of the subjects' answers with all-correct change rules is the best. In Table 10, the hypothesis testing results of the precision for JEdit shows that there is no statistically significant difference between the subjects' answers with imperfect and no change rules while the difference between the subjects' answers with all-correct and no change rules is statistically significant with large effective size.

Time The distribution of the times that subjects spent on JEdit is different than those on JHotDraw and JFreeChart as shown in Fig. 4. First, the subjects with imperfect change rules used the least time to answer. Second, the times spent by the subjects with the three treatments were very different. The hypothesis testing results (Table 11) confirm that only the difference between times spent by the subjects with imperfect and no change rules is significant with a large effective size. The differences between the times spent by the subjects with imperfect and all-correct change rules and with all-correct and no change rules are not significant.

Discussion JEdit is a free text editor supporting plugins and syntax highlighting for more than 200 programming languages. When we analyzed JEdit, we found that the implementations of v4.1 and v4.2 changed dramatically. The major differences between the two versions

Table 10 Hypothesis testing results for precision on JEdit

Treatements	Kruskal-Wallis *p*-value	Cliff's Delta
No-Imperfect-Correct	< 0.01 (0.00329)	N/A
No-Imperfect	> 0.01 (0.1959)	N/A
No-Correct	< 0.01 (0.001198)	Large (1.98641)
Imperfect-Correct	> 0.01 (0.02876)	N/A

Table 11 Hypothesis testing results for time on JEdit

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	< 0.01 (0.008001)	N/A
No-Imperfect	< 0.01 (0.001939)	Large (0.7322)
No-Correct	> 0.01 (0.4812)	N/A
Imperfect-Correct	> 0.01 (0.04125)	N/A

make the upgrading process more difficult for the subjects. The precision between the subjects with all-correct and no change rules are significantly different, but not between the subjects with imperfect and no change rules. However, the time that subjects spent with imperfect and no change rules is statistically different. The subjects with imperfect change rules spent much less time than the subject without change rule with similar precision in their answers. So, the change rules did save some effort for JEdit.

One result on JEdit that we did not expect is that the average time spent by the subjects with all-correct change rules was more than that for the subjects with imperfect change rules. To explain this observation, we first checked if there was any correlation between the results and the mitigating variables, but the answer was negative. Then, we checked if the ten subjects with all-correct change rules on JEdit also spent more time than the other subjects when they worked on JHotDraw and JFreechart with imperfect and no change rules. We found that these subjects did spend 39 % more time than the others, on average. We applied Kruskal-Wallis test to the time spent by these ten subjects and others on other tasks. The p-values are 0.20, 0.20, 0.05 and 0.83 on JHotdraw with imperfect and no change rules and JFreechart with imperfect and no change rules, respectively. These results show that, in three of the four other tasks, these ten subjects spent more time than the other subjects with more the 80 % confidence. The only exception was when they worked on JFreechart with no change rule. We suspect that all these ten subjects reached the upper bound of the time that they could spend on one program in our experiment, because JFreechart's code is more difficult to comprehend.

4.3 Summary

Based on the results of the experiment and the statistical analyses, we answer the research questions as follows:

- RQ_1: Is there a difference between the precision of the replacements of the target methods found by the subjects with all-correct, imperfect, and no change rules?

  Answer. Yes, the precision values of the subjects' answers with the help of all-correct, imperfect, and no change rules shows statistically significant differences. The subjects with all-correct change rules have the higher value, the next is with imperfect change rules, and the subjects without change rules have the lowest value. The effect size of the differences in precision values between the subjects with no and imperfect change rules is large and that between the subjects with imperfect and all-correct change rules is moderate.

- RQ_2: Is there a difference between the time that the subjects spend with all-correct, imperfect, and no change rules?

  Answer: No, there is no significant difference between the times spent by the subjects with all-correct, imperfect, and no change rules.

These results show that the change rules generated by framework API evolution approaches are useful. Yet, different to upgrading to new releases of frameworks in a real context, the subjects could only spend limited times on the tasks, no matter how serious they were. So, they could given wrong answers. In a real context, the change rules will help developers to adapt their client code to new releases of frameworks faster, because they must work on the upgrading until they have 100 % precision. As we discussed after presenting the results on each object program, the effect of the change rules can vary on a specific framework.

5 Discussion

We now discuss the influence of the mitigating variables and other issues impacting the results of our study.

5.1 Mitigating Variables

For mitigating variables like fatigue and learning effect, we used a randomized, complete block design to minimize their influences. There are five other mitigating variables whose influence we could not neutralize: gender, degree, knowledge of software engineering, Java, and Eclipse, because we did not have this information until the subjects performed the experiment. Therefore, we verified if there was any correlation between the precision of the subjects' answers (the times spent by the subjects) and the five mitigating variables, using permutations tests. A permutation test (Baker 1995) is non-parametric and does not require normal data distribution.

We also consider gender as a mitigating variable, because the differences between male and female in problem solving activities have been studied before (Beckwith et al. 2005; Meyers-Levy 1989; O'Donnell and Johnson 2001; Sharafi et al. 2012). We want to see if the results of our experiment are correlated with gender.

Tables 12, 13, 14, 15 and 16 show that the precision of the subjects' answers are not correlated to the five mitigating variables. (The time that the subjects spent are not correlated to the five mitigating variables either and the results can be found online.[10])

5.2 NASA Task Load Index

Besides the two objective measurements of subjects' performance (precision and time), we also asked the subjects to fill in the NASA Task Load Index (NASA-TLX) (Hart and Stavenland 1988) after each program to give their subjective evaluation of the effort to complete the tasks. The NASA-TLX is a subjective measurement of subjects' workload in human–machine environments, using six sub-scales: Mental Demands, Physical Demands, Temporal Demands, Own Performance, Effort and Frustration, as shown in Fig. 5. Each sub-scale is measured from 1 to 20. Level 1 represents the lightest or the best while level 20

[10] http://www.ptidej.net/download/experiments/emse13a

Table 12 Precision: two-way permutation test by change rule type and knowledge in software engineering

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9961.83	4980.91	500000.00	<0.01
SE	1	18.39	18.39	232090.00	0.81
Treatment:SE	2	1042.32	521.16	500000.00	0.20
Residuals	87	27276.64	313.52		

means heaviest or worst. We computed the average value of the six sub-scales as the overall NASA-TLX value (Hart and Staveland 1988).

Because the NASA-TLX depends on the subjects' personal feeling, we did not use it as a dependent variable, but it is interesting to see the relations between NASA-TLX value and the two objective measurements of subject performance. The distribution of the NASA-TLX values (see Fig. 6) and the hypothesis testing results on NASA-TLX values (see Tables 17 and 18) show that there is no statistical difference between the subjects with all-correct, imperfect, and no change rules. We argue that the main reason for observing statistically-similar NASA-TLX values is that the subjects' feelings vary dramatically between subjects.

We cross-referenced the NASA-TLX values and the times that the subjects spent. For the same NASA-TLX value, the times could be very different. For example, 11 subjects gave a NASA-TLX value of 7 for their tasks, but the times that they spent varied between 14 minutes (849 seconds) and 40 minutes (2,379 seconds). Similarly, we observed a subject with a NASA-TLX value of 2 who spent 40 minutes (2,378 seconds) on the tasks.

We suspect that the randomized, complete block design of the experiment amplified the variance of the subjects' feelings regarding their workload. If we had used only one object program, the subjects could have reported consistent feelings between the treatments. However, an experimental design with one object program would have been a serious threat to the generalizability of the results. The lesson learned is that it is risky to use only subjective measurements as dependent variables in experiments. Subjects may choose different values for similar personal feeling on same tasks, especially when the experiment design is complex.

5.3 Change Rule Types

The change rules can be classified in different ways. While considering code element changes, the change rules can be return type, package, class, method, and parameter changes (assuming frameworks written in Java). The change rules used in the experiment cover all these code element change types. While considering the mapping between changed API and its replacements, the types of the change rules include one-replaced-by-one, one-replaced-by-many, many-replaced-by-one, and simply-deleted. In our experiment, besides one-replaced-one change rules as illustrated in Section 3.3, the randomly selected change rules in our experiment also include one-replaced-by-many and simply-deleted

Table 13 Precision: two-way permutation test by change rule type and knowledge in java

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9961.83	4980.91	500000.00	<0.01
Java	1	64.57	64.57	500000.00	0.64
Treatment:Java	2	83.80	41.90	408504.00	0.88
Residuals	87	28188.98	324.01		

Table 14 Precision: two-way permutation test by change rule type and knowledge in eclipse

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9961.83	4980.91	500000.00	<0.01
Eclipse	1	55.64	55.64	469963.00	0.68
Treatment:Eclipse	2	291.67	145.84	500000.00	0.64
Residuals	87	27990.04	321.72		

rules. For the target methods of one-replaced-by-many change rules, the answers of subjects are correct only when they find all the replacements. Subjects can answer "null" as replacement when they think that the target methods are simply-deleted.

The goal of our experiment is to investigate the influence of all-correct and imperfect change rules in comparison to no change rules. The influence of different types of change rules is an interesting future study.

5.4 Skeptical Subjects

The results of the experiment show that the subjects were serious and skeptical about the provided change rules. Because they do not know a priori, even with the correct change rules, most of the subjects spent almost the same time as the subjects without them to find the replacement methods. The hypothesis testing results on time confirm this conclusion statistically. Moreover, the average precision of the subjects' answers with all correct change rules is just 82 %. Even with correct change rules, the subjects did correctly answer all the questions during the time that they spent for our experiment even though we did not give any time limitation to the subjects. This observation is a further evidence of the difficulty of framework API evolution.

5.5 Experiment vs. Real Tasks

The differences between our experiment and framework upgrading tasks in a real context are caused by the goal of this study, which focuses only on a part of the framework upgrading tasks. In the experiment, the subjects are volunteers. No matter how serious they are, they can only spend a limited amount of time on the experiment. We cannot expect all of them to be able or willing to finish all the tasks correctly if they take too long. In a real context, developers first would try to find the replacements of missing methods. If they could not find any proper replacements, then they would need to find a work-around. Finally, whether with replacements or work-arounds, they would test their programs after changes. If the test passes, then the upgrading task would be done, else they would keep working on it or they would stay with the previous release of the framework.

Our study focuses on the impact of imperfect change rules on framework API evolution identification. We want to investigate if the imperfect change rules still help developers

Table 15 Precision: two-way permutation test by change rule type and gender

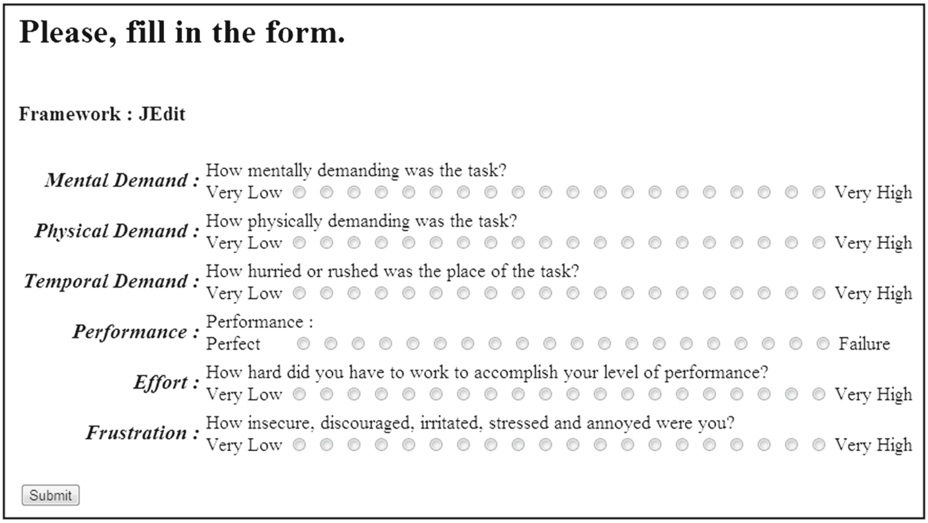
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9545.85	4772.92	500000.00	<0.01
Gender	1	582.55	582.55	500000.00	0.17
Treatment:Gender	2	156.21	78.11	500000.00	0.78
Residuals	87	27598.60	317.23		

Table 16 Precision: two-way permutation test by change rule type and degrees

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9956.99	4978.49	500000.00	<0.01
Degree	2	91.86	45.93	500000.00	0.86
Treatment:Degree	4	2014.42	503.60	500000.00	0.17
Residuals	84	26231.07	312.27		

find the replacements of missing methods caused by software evolution, such as class or parameter changes. It does not cover finding work-arounds and testing. If we had asked the subjects to perform real framework upgrading tasks, finding work-arounds and testing would have increased the precision and the time spent. However, following previous works on change-rules for framework APIs (Dagenais and Robillard 2011; Kim et al. 2007; Meng et al. 2012; Schäfer et al. 2008; Wu et al. 2010; Xing and Stroulia 2007), we assume that providing more accurate replacement methods can only help developers by reducing (1) the time spent understanding the new framework, (2) the time spent performing the changes necessary to adapt their programs, and (3) the overall time spent in the cycle {search-for-replacement, change, test}. Yet, in addition to the threats to the validity to our experiment discussed in Section 6, it is possible that other factors impact the accuracy and time of the overall cycle and future work is necessary (1) to study these factors in details, (2) to understand how time is spread throughout the cycle, and (3) to assess the impact of the change rules (and of their accuracy) on the whole cycle.

We argue that finding replacements for missing methods is a program comprehension task and, consequently, we designed our experiment as other program comprehension experiments, asking the subjects to answer questions based on their understanding of the frameworks. In our experiment, the subjects were volunteers. No matter how serious they were, they could only spend a limited amount of time on the experiment. We could not ask them to perform real framework upgrading tasks, because they could take very long time. Therefore, subjects may have given answers even if they were not 100 % confident.


Please, fill in the form.

Framework : JEdit

Mental Demand : How mentally demanding was the task?
Very Low ○ Very High

Physical Demand : How physically demanding was the task?
Very Low ○ Very High

Temporal Demand : How hurried or rushed was the place of the task?
Very Low ○ Very High

Performance : Performance :
Perfect ○ Failure

Effort : How hard did you have to work to accomplish your level of performance?
Very Low ○ Very High

Frustration : How insecure, discouraged, irritated, stressed and annoyed were you?
Very Low ○ Very High

Fig. 5 NASA-TLX

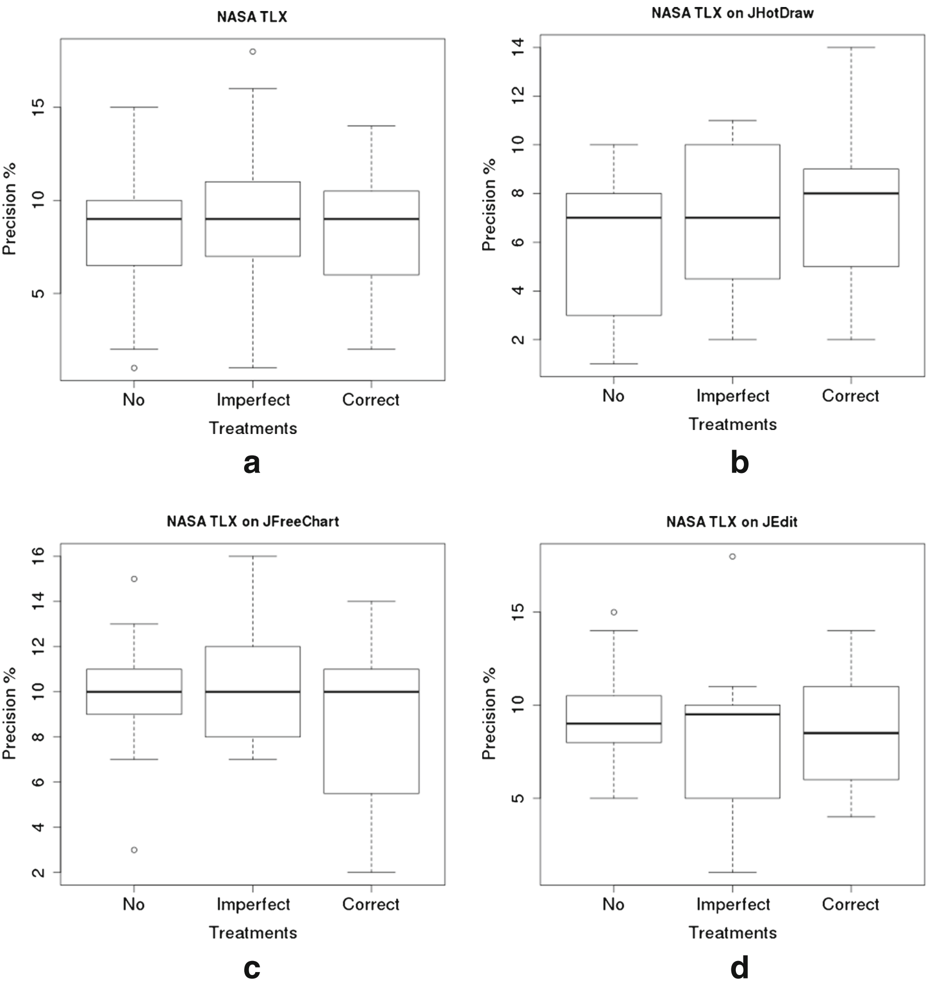


Fig. 6 Boxplots for NASA-TLX

Although we asked the subjects to take as much time as they needed, the time that the subjects spent ranges from 40 to 120 minutes and the precision of their answers lies between 14 % and 100 %. The average times for choosing a replacement method are 24, 23, and 25 minutes (1,413, 1,338, and 1,479 seconds). The times spent by the subjects with all-correct, imperfect and no change rules are not statistically different. These times are quite short and likely to be shorter than the times necessary to test the changes. On the one hand, such times show the importance of using API change rules to reduce the overall time of choosing and testing a change. On the other hand, such times do not show the total times needed to complete the changes to satisfy the tests in the absence of change rules, which may be longer than the average times with no change rules, because developers must find a work-around and not just provide a possible replacement, as in our experiment.

There is another difference between a real context and our experiment. In a real context, developers must have client code to test the replacements. In our experiment, we did not

Table 17 Hypothesis testing results for NASA-TLX

Treatements	Kruskal-Wallis p-value			
	General	JHotDraw	JFreeChart	JEdiit
No-Correct-Imperfect	> 0.01 (0.978)	> 0.01 (0.6182)	> 0.01 (0.6777)	> 0.01 (0.8287)
No-Imperfect	> 0.01 (0.9379)	> 0.01 (0.507)	> 0.01 (1)	> 0.01 (0.7228)
No-Correct	> 0.01 (0.9211)	> 0.01 (0.3443)	> 0.01 (0.4208)	> 0.01 (0.4989)
Imperfect-Correct	> 0.01 (0.8209)	> 0.01 (0.7191)	> 0.01 (0.4954)	> 0.01 (1)

provide such client code for two reasons. First, the test code could have helped the subjects to verify if the replacements were correct, but it cannot help developers find the correct replacements. Second, there could have been extra effort for the subjects to make the client code executable using the replacements. For example, if there was a new parameter in a replacement, to find out the proper value for that parameter could have taken time too. Such effort cannot be saved by change rules and we excluded it from our experiment.

5.6 Subjects vs. Professional Developers

In our experiment, most subjects are students. From their answers, we can see that they understood the frameworks well because most of their answers are correct. Still, they are different from professional developers. We can see subjects as inexperienced developers. Change rules may help experienced and inexperienced developers in different ways. For experienced developers, their experience can help them to distinguish the correct and wrong change rules. So, change rules let them quickly focus on "difficult" changes. For inexperienced developers, change rules can reduce the general comprehension effort by guiding them to a relatively small part of the frameworks. Only further studies with professional and expert subjects could help identify differences, if any.

6 Threats to Validity

Some threats limit the validity of our study. We discuss these threats and how we accepted or mitigated them following the guidelines provided by Wohlin et al. (1999).

Construct Validity Construct validity verifies that the observation really reflects the theory, i.e., if the treatment reflects the cause and the outcome reflects the effect. We wanted to evaluate if change rules help subjects find the replacements of target methods more accurately and faster. We used correct, imperfect, and no change rules as the treatments. We used the precision of the subjects' answers and the time that they spent as the outcomes. Some mitigating variables could affect the outcomes and we minimized their influence as described in Section 3.7. We did not observe correlations between the mitigating variables and the outcomes in the collected data. Thus, we argue that the treatments and outcomes reflect the cause and the effect. We use our tool, AURA (Wu et al. 2010), to collect raw change rules from which we selected 21 correct change rules, verified manually. Moreover, we used a dedicated Web site to present the change rules to the subjects. Therefore, we used AURA only to make it easier for us to build the sets TR_c and TR_i. The experiment does not depend on AURA, its algorithms or the formats of its change rules. We could have used another tool but choose AURA by convenience and due to his high precision on the three object programs.

Internal Validity Internal validity verifies that the outcome is really caused by the treatment. To overcome threats to internal validity, first, we selected three object programs to avoid that the experiment results depended on the properties of a single program. Second, we used a randomized, complete block design to avoid maturation threats, such as fatigue and learning effect. Third, we did not provide feedback to the subjects' answers, so the subjects could complete subsequent tasks using the answers of previous questions. Fourth, we chose Eclipse, a popular IDE as tool to explore the source code to avoid instrumentation threat. Among the 31 subjects, only one reported poor skills with Eclipse. We provided a tutorial about the Eclipse features required for the experiment to help such inexperienced subjects. Fifth, because wrong answers may be due to misunderstood tasks, we provided detailed experiment instructions and asked the subjects to read them carefully before answering questions. The experiment results showed that the subjects well understood the tasks. Sixth, asking subjects for their genders could lead to anxiety and decreased performances (stereotype threat). However, our study shows that gender could not explain any differences among subjects and, therefore, that this threat does not impact the results of our study. Last, no subject participated in the development of the experiment design, documents, tasks, and Web site. We also asked the subjects not to talk about the experiment with other people before the end of the study to avoid diffusion threat.

Conclusion Validity Conclusion validity verifies that the relation between the outcome and the treatment can be proved statistically. The null hypotheses are well defined. In total, 31 subjects participated in the experiment. We applied Kruskal-Wallis test (Wohlin et al. 1999), which is proper for randomized, complete block design to verify the significance level and Cliff's Delta (Grissom and Kim 2005) to evaluate the effect size. Both Kruskal-Wallis test and Cliff's Delta do not make any assumption on the distribution of data collected during the study.

External Validity External validity verifies that the results of a study are generalizable. We identify two threats to external validity. The first is the interaction between selection and treatment. In our experiment, the subjects acted as software developers. Yet, because of their limited number and specific demographics, they may not represent generally software developers accurately. However, all the 31 subjects who participated in the experiment were university-level students or had a degree in computer science or software engineering. Most of them had a good knowledge in Java, Eclipse, and software engineering. Among 29 subjects, 20 of them were graduate students. Thus, we believe that they represent junior software developers. The second threat is interaction of setting and treatment. We only used three Java programs and 21 target methods in our experiment. They may not reflect the whole framework API evolution problem. Considering the popularity of frameworks in Java and the variety of the three object programs, we believe this setting is not a major threat to external validity. We also chose the 21 target methods randomly, so there was no bias for this factor in our experiment design.

7 Conclusion

Software frameworks are indispensable in software development to reuse existing code and reduce development and testing costs. They constantly evolve to fix bugs and add new features. It is often time-consuming for developers to keep their code up-to-date with new framework versions due to the pace of software evolution. However, the upgrading process is not always documented. Thus, approaches have been proposed to lessen the impact of non-documented (or partially documented) framework evolution on developers by identifying change rules between two releases of a framework. Usually, the change rules are imperfect, i.e., not 100 % percent correct. To the best of our knowledge, there was no empirical study to show the usefulness of the change rules, i.e., to show whether imperfect change rules can help developers to identify the replacements more accurately or faster than without them. We designed and conducted an experiment to evaluate the precision of the replacement methods that subjects find and the time that they spend with all-correct, imperfect, and no change rules.

The dependent variables of the experiment were the precision of the replacement methods that the subjects found and the times that they spent with and without the help of change rules. We chose randomly 21 target methods as the independent variable of the experiment and defined three treatments: all-correct, imperfect, and no change rules. To limit the influence of a specific program on the results and to control experimental time, we chose three medium-size programs (JHotDraw v5.2–v5.3, JFreeChart v0.9.11–v0.9.12, and JEdit v4.1–v4.2) and seven target methods for each program. Because of the numbers of treatments and object programs, we use a randomized, complete block design (Wohlin et al. 1999) to minimized the number of subjects required and to overcome some threats to validity discussed in Section 6.

In total, 31 subjects participated in the experiment. In general, the statistical analysis results showed that the precision values of subjects' answers with all-correct, imperfect, and no change rules are significantly different with average values of 82 %, 71 % and 57 %, respectively. The effect size Cliff's Delta of the differences between the precision of the subjects' answers with no and imperfect change rules is large and that between the subjects with imperfect and correct change rules is moderate. The time that the subjects spent with the three treatments to find the replacements methods is not statistically different with average values of 24, 23, and 25 minutes (1,413, 1,338, and 1,479 seconds), respectively.

In conclusion, the results of our study show that the change rules generated by framework API evolution approaches are useful. The higher precision the change rules have, the better help they provide. Thus, the imperfect change rules can be used instead of unavailable documentation or as complement to partial documentation. Developers of frameworks could also use them as starting point to build upgrading documentation. The experiment results also suggest that approaches that generate change rules may not need perfect precision but rather possibly should seek other ways to help developers, for example through better integration with development environment or through generating work-arounds for simply deleted methods. Consequently, while future work includes experiments to investigate the influence of types of change rules and the influence of change rules with different values of precision, it also should investigate other ways to help developers, including the behaviors between male and female, professional and student, novice and expert subjects when they perform framework API evolution identification tasks, formats of change rules, mining code samples to generate work-arounds.

Table 18 Task distribution (C-Correct, I-Imperfect, N-No, JHD-JHotDraw, JFC-JFreeChart, JE-JEdit)

Order	Treatment	Subject																														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	change rules	I	I	I	N	N	N	C	N	C	C	I	I	N	N	N	N	I	C	N	I	C	C	C	N	C	N	C	C	C	I	I
	System	JHD	JHD	JHD	JE	JFC	JE	JFC	JHD	JE	JFC	JE	JFC	JHD	JHD	JFC	JE	JFC	JHD	JHD	JFC	JE	JFC	JHD	JHD	JE	JFC	JFC	JHD	JHD	JE	JFC
2	change rules	C	C	N	I	C	I	I	C	I	I	N	N	I	C	I	C	N	N	C	N	N	N	I	C	N	I	N	I	I	N	N
	System	JE	JFC	JFC	JHD	JHD	JFC	JE	JE	JHD	JHD	JFC	JE	JE	JFC	JHD	JFC	JE	JE	JFC	JHD	JFC	JE	JE	JFC	JHD	JHD	JE	JE	JFC	JHD	JHD
3	change rules	N	N	C	C	I	C	N	I	N	N	C	C	C	I	C	I	C	I	I	C	I	I	N	I	I	C	I	N	N	C	C
	System	JFC	JE	JE	JFC	JE	JHD	JHD	JFC	JFC	JE	JHD	JHD	JFC	JE	JE	JHD	JHD	JFC	JE	JE	JHD	JHD	JFC	JE	JFC	JE	JHD	JFC	JE	JFC	JE

Acknowledgments The authors would like to thank all the anonymous subjects, who contributed their effort and time for this work, and the anonymous reviewers for their comments that greatly improve the quality of this article.

References

Abbes M, Khomh F, Guéhéneuc Y-G, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Proceedings of the 2011 15th European conference on software maintenance and reengineering, ser. CSMR '11. IEEE Computer Society, Washington, DC, USA, pp 181–190

Ali N, Sharafi Z, Guéhéneuc Y-G, Antoniol G (2012) An empirical study on requirements traceability using eye-tracking. In: Proceedings of the international conference on software maintenance, ser. ICSM 2012

Antoniol G, Penta MD, Merlo E (2004) An automatic approach to identify class evolution discontinuities. In: IWPSE '04: Proceedings of the principles of software evolution, 7th international workshop. IEEE Computer Society, pp 31–40

Baker RD (1995) Modern permutation test software. In: Edgington E (ed) Randomization tests. Marcel Dekker Incorporated

Beckwith L, Burnett M, Wiedenbeck S, Cook C, Sorte S, Hastings M (2005) Effectiveness of end-user debugging software features: are there gender issues?. In: Proceedings of the SIGCHI conference on human factors in computing systems, ser. CHI '05, New York, NY, USA, pp 869–878

Businge J, Serebrenik A, van den Brand M (2013) Analyzing the eclipse api usage: putting the developer in the loop. In: CSMR, pp 37–46

Chow K, Notkin D (1996) Semi-automatic update of applications in response to library changes. In: Proceedings of the 1996 international conference on software maintenance. ser. ICSM 1996. IEEE Computer Society, Washington, DC, USA, p 359

Dagenais B, Robillard MP (2011) Recommending adaptive changes for framework evolution. ACM Trans Softw Eng Methodol 20(4):19:1–19:35

Demeyer S, Ducasse S, Nierstrasz O (2000) Finding refactorings via change metrics. In: OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM, New York, NY, USA, pp 166–177

Dig D, Comertoglu C, Marinov D, Johnson R (2006) Automated detection of refactorings in evolving components. In: ECOOP '06: Proceedings of the 20th European conference on object-oriented programming. Springer, Berlin

Dig D, Manzoor K, Johnson R, Nguyen TN (2007) Refactoring-aware configuration management for object-oriented programs. In: ICSE '07: Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp 427–436

Fagard RH, Staessen JA, Thijs L (1996) Advantages and disadvantages of the meta-analysis approach. J Hypertens 14(2)

Fluri B, Gall HC (2006) Classifying change types for qualifying change couplings. In: ICPC '06: Proceedings of the 14th IEEE international conference on program comprehension. IEEE Computer Society, Washington, DC, USA, pp 35–45

Gamma E, Helm R, Johnson RE, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley

German DM, Hassan AE (2009) License integration patterns: addressing license mismatches in component-based development. In: ICSE '09: Proceedings of the 2009 IEEE 31st international conference on software engineering. IEEE Computer Society, Washington, DC, USA, pp 188–198

Godfrey MW, Zou L (2005) Using origin analysis to detect merging and splitting of source code entities. IEEE Trans Softw Eng 31(2):166–181

Grissom R, Kim J (2005) Effect sizes for research: a broad practical approach. Lawrence Erlbaum Associates

Hart SG, Stavenland LE (1988) Development of NASA-TLX (Task Load Index): results of empirical and theoretical research, pp 139–183

Hart SG, Staveland LE (1988) Nasa task load index (tlx) v 1.0. http://humansystems.arc.nasa.gov/groups/TLX/downloads/TLX.pdf

Henkel J, Diwan A (2005) Catchup: capturing and replaying refactorings to support api evolution. In: ICSE '05: Proceedings of the 27th international conference on Software engineering. ACM, New York, NY, USA, pp 274–283

Kataoka Y, Ernst MD, Griswold WG, Notkin D (2001) Automated support for program refactoring using invariants. In: Proceedings of the IEEE international conference on software maintenance, ser. ICSM 2001. IEEE Computer Society, Washington, DC, USA, p 736

Kemper C, Overbeck C (2005) What's new with jbuilder. In: JavaOne Sun's 2005 worldwide java developer conference

Kim M, Notkin D, Grossman D (2007) Automatic inference of structural changes for matching across program versions. In: ICSE '07: Proceedings of the 29th international conference on software engineering. IEEE Computer Society, Washington, DC, USA, pp 333–343

Kim S, Pan K, Whitehead EJ Jr. (2005) When functions change their names: automatic detection of origin relationships. In: WCRE '05: Proceedings of the 12th working conference on reverse engineering. IEEE Computer Society, Washington, DC, USA, pp 143–152

Lawrie D, Morrell C, Feild H, Binkley DW (2007) Effective identifier names for comprehension and memory. ISSE 3(4):303–318

Likert R (1932) A technique for the measurement of attitudes. Arch Psychol 22(140):1–55

Malpohl G, Hunt JJ, Tichy WE (2000) Renaming detection. In: ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering. IEEE Computer Society, Washington, DC, USA, p 73

Meng S, Wang X, Zhang L, Mei H (2012) A history-based matching approach to identification of framework evolution. In: Proceedings of 34th international conference on software engineering, ser. ICSE 2012, pp 353–363

Meyers-Levy J (1989) Gender differences in information processing: a selectivityinterpretation. In: Cafferata P, Tybout A (eds) Cognitive and affective responses to advertising. Lexington Books

Miller RGJ (1981) Simultaneous statistical inference, 2nd ed. Springer

O'Donnell E, Johnson E (2001) The effects of auditor gender and task complexity on information processing efficiency. Int J Audit 5(2):91–105

Schäfer T, Jonas J, Mezini M (2008) Mining framework usage changes from instantiation code. In: ICSE '08: Proceedings of the 30th international conference on software engineering. ACM, New York, NY, USA, pp 471–480

Sharif B, Maletic JI (2010) An eye tracking study on camelcase and underscore identifier styles. In: Proceedings of the 2010 IEEE 18th international conference on program comprehension, ser. ICPC '10. IEEE Computer Society, Washington, DC, USA, pp 196–205

Sharafi Z, Soh Z, Guéhéneuc Y-G, Antoniol G (2012) Women and men - different but equal: on the impact of identifier style on source code reading. In: ICPC, pp 27–36

Weißgerber P, Diehl S (2006) Identifying refactorings from source-code changes. In: ASE '06: Proceedings of the 21st IEEE/ACM international conference on automated software engineering. IEEE Computer Society, Washington, DC, USA, pp 231–240

Wohlin C, Runeson P, Höst M (1999) Experimentation in software engineering: an introduction. Springer

Wu W, Guéhéneuc Y-G, Antoniol G, Kim M (2010) Aura: a hybrid approach to identify framework evolution. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 1, ser. ICSE '10. ACM, New York, NY, USA, pp 325–334. [Online]. Available: doi:10.1145/1806799.1806848

Xing Z, Stroulia E (2006) Refactoring detection based on umldiff change-facts queries. In: WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering. IEEE Computer Society, Washington, DC, USA, pp 263–274

Xing Z, Stroulia E (2007) API-evolution support with diff-CatchUp. IEEE Trans Softw Eng 33(12):818–836

Yusuf S, Kagdi H, Maletic JI (2007) Assessing the comprehension of uml class diagrams via eye tracking. In: Proceedings of the 15th IEEE international conference on program comprehension, ser. ICPC '07. IEEE Computer Society, Washington, DC, USA, pp 113–122

Zhang C, Budgen D (2012) What do we know about the effectiveness of software design patterns? Trans Softw Eng 38(5):1213–1231. [Online]. Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5975176

Wei Wu is Ph.D candidate in Software Engineering at École Polytechnique de Montréal under the supervision of Prof. Yann-Gaël Guéhéneuc and Prof. Giuliano Antoniol. He received his M.S. in Computer Science at Université de Montréal under the supervision of Prof. Yann-Gaël Guéhéneuc in 2010. His research interests are Software Evolution, Empirical Software Engineering, Software Maintenance, Sotware Quality, Software Testing and Data Mining.

Adrien Serveaux is software engineer at Orange in France. He holds a master degree in telecommunication, network and computer science from Télécom Saint-Etienne, France. From March to September 2012, he did his internship in Ptidej Team at the Ecole Polytechnique de Montréal, Canada.

Yann-Gaël Guéhéneuc is full professor at the Department of computer and software engineering of École Polytechnique de Montréal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. He is IEEE Senior Member since 2010. In 2009, he was awarded the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from École des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals, including IEEE TSE, Springer EMSE, ACM/IEEE ICSE, and IEEE ICSM. He is currently (2013-2014) in sabbatical in Korea, working with colleagues at KAIST, Yonsei U., and SNU.

Giuliano Antoniol (Giulio) Canada Research Chair in Software Change and Evolution, served as program chair, industrial chair, tutorial, and general chair of international conferences and workshops. He contributed to the program committees of more than 30 IEEE and ACM conferences and workshops. He is a member of the editorial boards of five software engineering journals and he acts as referee for all major software engineering journals.

He is currently Full Professor at the Polytechnique Montreal, where he works in the area of software evolution, empirical software engineering, software traceability, search based software engineering, software testing and software maintenance.