Improving the Success Rate of Applying the Extract Method Refactoring

Juan Pablo Sandoval Alcocer[a,*], Alejandra Siles Antezana[a], Gustavo Santos[b], Alexandre Bergel[c]

[a] *Departamento de Ciencias Exactas e Ingeniería*
Universidad Católica Boliviana "San Pablo", Cochabamba, Bolivia
[b] *Federal University of Technology - Paraná*
[c] *ISCLab, Department of Computer Science (DCC), University of Chile, Santiago, Chile*

Abstract

Context: Most modern programming environments support refactorings. Although refactorings are relevant to improve the quality of software source code, they unfortunately suffer from severe usability issues. In particular, the extract method refactoring, one of the most prominent refactorings, has a failure rate of 49% when users attempt to use it.
Objective: Our main objective is to improve the success rate of applying the extract method refactoring.
Methods: First, to understand the cause of refactoring failure, we conducted a partial replication of Vakilian's ICSE '14 study about usability issues of refactoring using IntelliJ IDEA. Second, we designed and implemented TOAD, a tool that proposes alternative text selection for source code refactoring for the Pharo programming language. Third, we evaluated TOAD using a controlled experiment against the standard Pharo code refactoring tool. Seven professional software engineers complemented with three undergrad students participated in our experiments.
Conclusion: The causes we identified of failed extract method refactoring attempts match Vakilian's work. TOAD significantly reduces the number of

*Corresponding Author
Email addresses: sandoval@ucbcba.edu.bo (Juan Pablo Sandoval Alcocer), aesa1@estudiantes.ucbcba.edu.bo (Alejandra Siles Antezana), abergel@dcc.uchile.cl (Alexandre Bergel)

failed attempts to run the extract method refactoring at a lower cognitive load cost.

Keywords: Refactoring, Usability

---

1. Introduction

Refactoring tools help developers to automatically perform many predefined source code transformations and refactorings [1, 3, 12]. However, besides the automated facilities provided by these tools, recent empirical studies show that developers face a variety of usability issues [16]. These issues range from unintuitive refactoring configuration to an unexpected outcome from what the developer wanted to achieve. Such usability issues ultimately discourage developers from using refactoring tools in the future.

First, to better understand usability issues that practitioners are experiencing, we conducted a partial replication of a study conducted by Vakilian *et al.* [16]. This first study allowed us to confirm that a wrong selection of code segments is a prominent pattern when one applies the Extract Method refactoring. We found that only 51% of the Extract Method refactoring attempts were successful, where incorrect source code selection is a major cause of misuse of the refactoring tool. Such a pattern leads to a negative experience that defeats the main goal of programming environments, which is to assist practitioners in conducting laborious and repetitive tasks. We hypothesize some heuristics to guide the developer to properly select the source code segment to be refactored.

And second, we designed and implemented *TOAD*, a tool that searches for appropriate source code selection. An appropriate code selection is the one that satisfies the necessary preconditions to perform a specific refactoring; in our case, the Extract Method refactoring.

In this extension, we present a controlled experiment, in which we empirically compare TOAD with Pharo's standard refactoring tool. Our results show that:

- When compared with the standard Pharo refactoring tool, TOAD significantly reduces the number of failed attempts to apply the Extract Method refactoring;
- Participants often consult alternative text selections (43% of the refactoring attempts) and they also used an alternative text selection

instead of their first one (27%) when applying an Extract Method refactoring;

- Offering optional alternatives in the refactoring tools does not overload the refactoring process and reduces the participants cognitive load.

Outline. This paper is structured as follows. Section 2 presents a replication study on IntelliJ IDEA that highlights the usability issues related to Extract Method refactoring. Section 3 introduces TOAD, a tool that recommends alternative source code selections for Extract Method. Section 4 describes a controlled experiment that compares TOAD with Pharo's standard refactoring tool. Section 5 discusses threats to validity and how we addressed them. Section 6 compares our work with related work. Section 7 concludes our overall results.

Previous work. This article is an extension of a short paper presented at the ICSE '19 Student Research Competition Paper [13]. We extended our previous papers in numerous different ways: (i) we doubled the number of participants in our partial replication study (from 5 to 10); (ii) we detailed the categorization of the patterns found during our partial-replication (iii) we performed a controlled experiment to compare TOAD and the Pharo Standard Refactoring Tool.

2. Of Usability and Refactoring Tools: A Partial-Replication Study

In 2014, Vakilian *et al.* [16] reported 15 categories of usability issues that developers experienced while interacting with Eclipse refactoring tools. Motivated by this work, we performed a user study to replicate it and to better understand these usability issues. In particular, we focused on the tools provided by *IntelliJ IDEA* for Extract Method refactoring. We selected *IntelliJ IDEA* to complement Vakilian's *et al.* work, which was done in *Eclipse*.

We qualify our experiment as a partial replication because we use a different programming language and programming environment.

2.1. Methodology

In our study we (i) identify the proportion of failed Extract Method attempts and (ii) classify these failed attempts. We use the following methodology:

S1 - selecting participants for our experiment;

S2 - defining relevant and representative tasks to be carry out by the participants;

S3 - executing and monitoring work sessions;

S4 - quantitatively and qualitatively analyzing the observations.

Our methodology differs from the one used in the original Vakilian's study [16]. Vakilian *et al.* designed a tool that monitored refactoring attempts. Since we focused on only one refactoring, we opted for a simpler methodology. Analyzing screen recordings and transcripts of each participant was considered enough to extract all the data necessary for our replication.

2.2. Experimental Setup

Participants. Seven engineers and three undergrad students participate in our user study. The engineers work in two Bolivian software companies. Work experience ranges from two to seven years. The three students were in their 5th and last year of University when we conducted the experiment. The students have experience in refactoring tools acquired during their Software Engineering lectures. Among the ten participants, three are women. Engineers have experience in a large spectrum of different projects, including web applications and domain specific libraries and interpreters.

Tasks. We previously selected five long methods from the *JFreeChart* open-source project. Their size varies from 130 to 356 lines of code. We randomly assigned one method to two participants separately (five methods, ten participants). We requested each participant to split up a long method into smaller ones using the tools provided by *IntelliJ IDEA*, *i.e.,* using the Extract Method refactoring. The methods are available online for inspection[1].

Data Collection and Analysis. We recorded the screen of each user work session. During the session, we asked participants to vocally and precisely describe their refactoring intentions and expectations. In total, we

[1] `https://github.com/Aleli03/LinksToMethods`

Table 1: Usability issues while using the extract method option by participants

Category	Participant										TOTAL
	1	2	3	4	5	6	7	8	9	10	
Change as Expected	7	2	7	8	7	4	13	5	5	0	58 (50.9%)
Invalid Source Code Selection	1	0	0	2	0	1	1	3	4	4	16 (14,0%)
Unexpected Source Code Change	2	1	0	0	2	1	1	0	0	6	13 (11,4%)
Ambiguous Return Value	0	4	0	6	0	0	1	1	0	0	12 (10,5%)
Confusing Messages	3	0	0	0	3	1	0	0	0	0	7 (6,1%)
Other	1	0	0	0	2	1	0	0	0	4	8 (7,0%)
TOTAL											**114 (100%)**

recorded 215 minutes of sessions, where participants performed 114 refactoring attempts. We consider a refactoring attempt to be each time the user selected the option "Extract Method" in the IDE. We analyzed and categorized each of these attempts according, but not restricted to Vakilian's *et al.* study. Recorded videos are also available online.[2].

2.3. Findings

Out of the 114 refactoring attempts, 58 (50.8%) concluded with no errors and the participants were satisfied with the refactoring result. We carefully analyzed each attempt and categorized the usability issues they experienced. Table 1 shows the usability issues we found with the participants. We describe each category as follows: each one has a title, a brief description, followed by an illustration example.

Invalid Code Selection. In 16 attempts (14%), participants selected a code segment that did not meet the refactoring preconditions, leading to a generic error message. For instance, consider the code selection in Figure 1 and its corresponding error message.

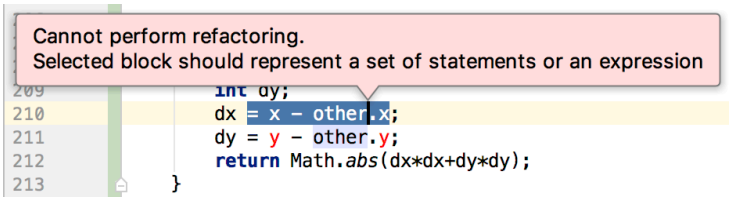

Figure 1: Example: Invalid Code Selection

[2]https://github.com/Aleli03/TOAD

In this particular case, the source code selection does not meet the preconditions because of the inclusion of the character "=". If the selection began after the character, there would not be an error.

Multiple Return Values. In 12 attempts (10.5%), participants selected a code segment with two or more variables being referenced externally. Consequently, the extracted method should have more than one return value, which is not syntactically correct. For instance, consider the code selection in Figure 2: to preserve the behavior the extracted method, it should return two values, as shown in the error message.

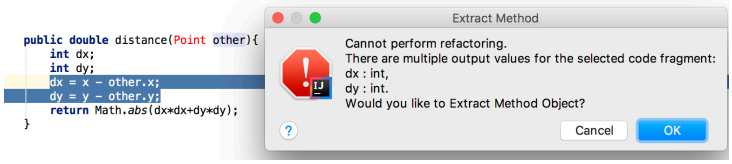

Figure 2: Example: Multiple Return Values

The attempts to perform the Extract Method refactoring failed, however IntelliJ IDEA proposes an alternative refactoring: "Extract Method Object".

Unexpected Source Code Change. In 13 attempts (11.4%), participants were not convinced by the result of the refactoring and they rolled back the changes. This mainly happened when the IDE suggested that the participants to extract and move the method to a new class through the "Extract Method Object" refactoring option (Figure 2). The fact that the IDE gives this option discourages the participants from continuing the refactoring.

Confusing Message. In 7 attempts (6.1%), participants got a confusing error message. For instance, one participant extracted a code snipped to be moved to a new method and provided the name of an existing one. IntelliJ IDEA provided an error message that was perceived as confusing (Figure 3). The participant canceled the attempt and provided a new name instead of opting for a guidance by the IDE. We define a confusing message a message that makes the participant cancel the current flow and change the selection or the method name.

Note that this category of failed refactoring was not reported in Vakilian's original work. Since we directly monitored participants we were able to

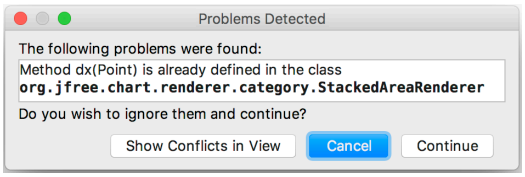

Figure 3: Example: Invalid Code Selection

identify confusing messages while Vakilian's instead monitored the actions performed in the IDE.

Others. In 8 of the attempts (7%), the IDE offered some options before applying the refactoring, which confused the participants. For instance, a popup recommended participants to apply the same refactoring to another similar source code in the same class. However, after a few other attempts, the participants got familiar with it. Two participants selected a code segment and a refactoring option, but after a few minutes they canceled the refactoring. In three attempts, the participants were curious about the refactoring option "type parameter", but after proceeding it no change is perceived by the participants.

Overall, participants did not expect additional options. So, when an expected popup menu appears for the first time, they thought that the menu was not relevant to achieve their goals. As such, the participants quickly closed the popup. However, the refactoring was not performed, so they tried to do it again, and the second time the participants read the popup carefully and got familiar with it.

Summary: **Only 51% of the extract method refactoring attempts are successful. The most prominent usability issues are related to the source code selection.**

3. TOAD: A Tool for Recommending Refactoring Alternatives

To tackle the usability issues related to source code selection described in our replication study, we propose *TOAD*, a tool that proposes alternative text selection for source code refactoring. *TOAD* is a refactoring tool which provides refactoring alternatives implemented in the Pharo programming language. Our hypothesis is that providing valid source code selection

alternatives improves the success rate of applying the extract method refactoring. As we saw in the previous section, a great portion of failing attempts are caused by invalid source code selections.

TOAD starts to operate when the user selects a code segment and then applies an Extract Method refactoring. *TOAD* uses the code selection as input and searches code sections that are syntactically correct and meet the refactoring preconditions. Within the scope of this paper, we only consider the Extract Method preconditions.

TOAD follows a three-step algorithm to show five valid source code selection alternatives.

- *Step 1: Source Selection Candidate* – First, *TOAD* searches for all possible code selection candidates in the method under analysis. For this, we use a string search based brute force algorithm. Consider that a method source code has n tokens and the extract method refactoring has m preconditions $P = p_1, \ldots, p_m$. A selection candidate may be represented as a pair (i, j) where $0 \leq i \leq j \leq n$, where the source code between the interval (i, j) meet all preconditions in P. For our implementation in Pharo, we use the standard refactoring preconditions and the standard string tokenizer to identify (i, j) combinations.

- *Step 2: Filtering* – We filter out selection candidates that do not contain any character of the user selection. We focus on candidates that overlap the code selected by the user (the input).

- *Step 3: Refactoring Alternatives –TOAD* only provides to the user the five closest source selection candidates, prioritizing the ones that overlap a great portion of the initial user selection. They are sorted so that the closest alternative is shown in the middle, with shorter alternatives on the left and longer ones on the right. In case there are less than five alternatives, *TOAD* shows all of them. We limited the alternatives to five, so as to not overload users, but our algorithm it is not tied to this number.

This approach could easily generate a large number of refactoring alternatives, from one statement to the entire method to be refactored. Since we do not want to overload to the user with too many options, we design a user interface that only shows five alternatives. These alternatives may be selected using the buttons at the bottom of the interface. The

middle button gives an alternative closest to the source code selection (*i.e.*, Figure 4 marked with 1), buttons on the right offer alternatives involving a larger code section than the selected one (*i.e.*, Figure 4 marked with 2), buttons on the left one offer alternatives involving less code than the selected one (*i.e.*, Figure 4 marked with 3).

Figure 4 shows an example of *TOAD* usage. The selected source code does not meet the conditions to apply Extract Method correctly (*i.e.*, the selected code is neither a set of statements nor an expression). *TOAD* (i) produces five different selections that are relatively close to the original selection made by the user and (ii) previews of the refactoring results for each alternative. Figure 4 (right side) shows three alternatives that *TOAD* found that are close to the user' selection in Figure 4 (left side). *TOAD* also shows a preview of the extracted method and how the original method would be modified.

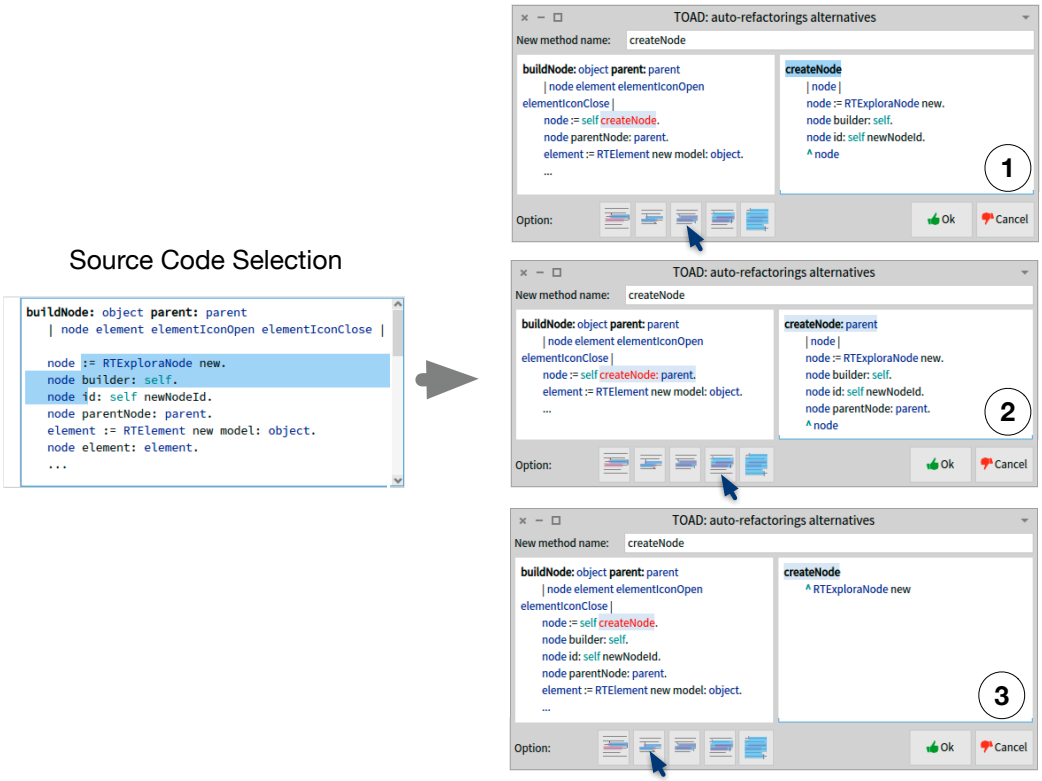

Figure 4: *TOAD* in action: five adequate selections are suggested from an invalid selection made by a user. The figure shows three of these suggestions.

Note that *TOAD* implements a simple approach to display the refactoring alternatives to the user. The goal of TOAD is to show that a simple approach may reduce failing attempts while refactoring. TOAD is available online [3].

4. Evaluation: a Controlled Experiment

This section describes the experiment we conducted to evaluate how *TOAD*'s recommendations were perceived by practitioners. We conducted a controlled experiment and used Pharo's standard refactoring tool as a baseline. We opted for a controlled experiment to verify whether the use of *TOAD* may be related to a reduction of failed attempts to perform the Extract Method refactoring.

4.1. Experimental Setup

Participants. We selected ten participants that have experience in the Pharo programming language. All of them are either authors of a Pharo project or deeply involved in the development of a Pharo core functionality. This is an important requirement for our experiment because we aim to reduce bias related to a poor understanding of the proposed refactoring tasks. Ensuring that participants have a fair comprehension of the code they will work with is therefore relevant.

Table 2 details participants' experience in software development, Pharo and refactoring tools. Seven out of ten participants have experience using refactoring tools. Participants are currently working in academia and industry. Four are from Bolivia and six from Chile. These participants are different from the ones that participated in our replication study.

Projects under Study. We asked each participant to select a Pharo project they were working on. Table 3 shows the projects we used in the experiment. Note that two participants (P6 and P9) were working on the *Roassal* project. Therefore, they performed refactoring tasks on the same project. All the other participants used different projects.

Setup. We previously determined the size of each method for each project under analysis. We then ordered the methods from the largest to the smallest. Each participant picked four methods to refactor. The list was later proposed

[3] `https://github.com/AleliO3/TOAD`

Table 2: Participants (Pharo Exp. = Pharo Experience; Prev. Exp. Ref. = Previous Experience with Refactoring Tools)

ID	Main Activity	Soft. Dev. Experience (years)	Pharo Exp. (years)	Prev. Exp. Ref.	Country
P1	Software Engineer	2	1	✓	Bolivia
P2	Software Engineer	0.5	0.5	✓	Bolivia
P3	Undergrad Student	5	1	✗	Bolivia
P4	PhD. Student	4	4	✓	Chile
P5	Software Engineer	6	2	✓	Bolivia
P6	PhD. Student	6	2	✗	Chile
P7	PhD. Student	6	6	✗	Chile
P8	Software Engineer	5	5	✓	Chile
P9	Software Engineer	10	5	✓	Chile
P10	PhD. Student	15	6	✓	Chile

Table 3: Project under Study (Part. = Participants)

Project Name	Short Description	Part.
GitMultipileMatrix	A stacked adjacency matrix to visualize software evolution	P1
TestDeviator	A test case generation technique for GraphQL APIs	P2
DrTest	An extendable, plugins-based UI for testing Pharo projects	P3
Regis	A Conference Registration Website	P4
SmallSuiteGenerator	A Test Case Generator for Pharo	P5
Roassal	A script system for advanced interactive visualizations.	P6, P9
Live Robot Programming	A live programming language for robot behaviors using nested state machines	P7
KerasBridge	A Pharo bridge for Keras (a deep learning library)	P8
GToolkit Documenter	A tool for creating and consuming live documents inside the development environment	P10

to each of the participants and their respective projects. The refactoring task consisted of splitting up each of the four methods using the Extract Method refactoring. Note that we excluded methods from our list that are either data holders or tests. Only methods having a logic relevant to the application under analysis were proposed.

Task and Treatments. Our experiment compares the refactoring experience between two treatments: *PharoStandard*, the standard Extract Method refactoring available on Pharo, and *TOAD*, the tool for recommending Extract Method refactoring alternatives. Participants were requested to refactor four large (previous selected) methods, two of them using the Pharo standard Extract Method tool and two using TOAD.

Work Session. For each treatment {*PharoStandard, TOAD*}, we followed the next steps.

1. *Learning material* – We provided learning material to the participants. We also performed a demo to let participants get familiar with the tools.

2. *The task* – We requested each participant to refactor two large methods. We describe the method selection for each project in the experiment setup.

3. *Video Recording* – We recorded each refactoring session. We asked participants to follow the think-aloud protocol by vocally describing their refactoring intentions and expectations. The think-aloud protocol is a method used to collect data by encouraging participants to say whatever comes into their mind as they complete the task.

4. *Task Load Index* – After each session, we requested each participant to fill in the NASA Task Load Index [4] [4] to assess the cognitive workload using six dimensions: mental demand, physical demand, temporal demand, performance, effort, and frustration. Each participant rates her/his perceived workload across these six dimensions to determine an overall workload rating. Participants assign to each dimension a value between 1 (low) to 20 (high).

[4] `https://humansystems.arc.nasa.gov/groups/TLX/`

5. *Participants Feedback* – At the end of each session we informally interviewed each participant and asked open questions to not pressure participants into giving an answer we expected.

Video Analysis. We analyzed each video session and categorized all the Extract Method attempts based on categories defined in our replication study. Additionally, we measured how many times users inspect the alternative options and when they apply another alternative rather than the initially selected in the source code.

Pilot. Before running the work sessions with all participants, we performed a pilot with a refactoring expert that has academic and industrial backgrounds. This experiment helps us make some improvements:

- *Source Code To Refactor* – In the pilot, we provided to the participant the longest methods of Roassal project and asked her/his to refactor them. We noticed that: i) the user was not confident enough to refactor an unfamiliar method, and ii) some methods did not contain any logic (*i.e.,* methods that contain long meta description) and some of them are expected to be long (i.e., script examples). For this reason, we refined our way of selecting the candidate methods to refactor. We developed a script to list the methods that have the most lines of code in the project and discarded the ones that do not have any application logic. Then we let the participant decide which ones are good candidates for refactoring.

- *Learning Material* – To complete the Extract Method refactoring, one must provide a new method name. Although apparently simple and intuitive, we encountered some issues related to this. A candidate provided the argument names in addition to the method name. However, both the standard Pharo refactoring tool and TOAD expect the method name only, with semicolons to denote the location of an argument, and without spaces. Therefore, we updated the learning material to indicate the only the method name is compulsory. Additionally, we encouraged the pilot participant to refactor a toy method using the refactoring tools as a preliminary and learning task.

4.2. Results

4.2.1. Observations

Considering all refactoring sessions, our participants performed 204 Extract Method attempts. Table 4 summarizes the attempts for each participant.

Table 4: Participants extract method refactoring attempts

Participant	Pharo Standard Attempts				TOAD Attempts			
	Success	Fail	Total	Success Rate (%)	Success	Fail	Total	Success Rate (%)
P1	6	7	13	46%	8	2	10	80%
P2	1	13	14	7%	6	2	8	75%
P3	6	4	10	60%	5	2	7	71%
P4	13	2	15	87%	12	8	20	60%
P5	5	4	9	56%	6	0	6	100%
P6	8	5	13	61%	3	0	3	100%
P7	8	15	23	35%	4	2	6	67%
P8	4	2	6	67%	8	2	10	80%
P9	6	5	11	54%	4	0	4	100%
P10	5	3	8	62%	5	3	8	63%
TOTAL	**62**	**60**	**122**	**51%**	**61**	**21**	**82**	**74%**

Success Rate. All participants have a different number of attempts. We define a metric *Success Rate*, which is the ratio between the number of times the participants use the extract method refactoring tool successfully and the total number of times the open the tool.

In total, 122 attempts were done using the Pharo Standard tool, from which only 51% of the attempts were considered successful. The remaining attempts ended with the participant canceling or closing the tool window, or undoing the changes. The remaining attempts, 82, were performed using *TOAD*, which represents 32% fewer attempts than with the Pharo standard tool. With TOAD, 74% of the attempts were successful. Figure 5 presents the distribution of the success rate among all participants.

We run the Mann Whitney statistical test (also called Wilcoxon rank-sum test) over the success rate of the two treatments (two-tailed, confidence level = 95%). The null hypothesis is that the distributions of both groups are

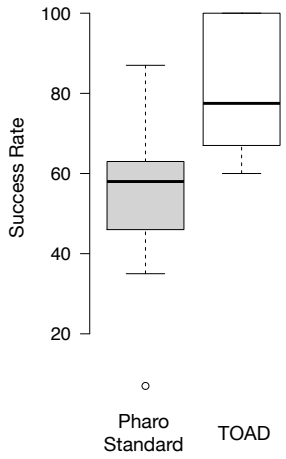

Figure 5: Extract Method Success Rate

identical.[5] The test indicates that the distribution is statistically different (P-value = 0.0154, Mann-Whitney U=18.50). **We can reject the null-hypothesis and conclude that there is a causal effect between the treatment (*i.e., PharoStandard* or *TOAD*) independent variable and the success rate dependent variable.**

Categorization of Attempts. To better understand the reasons behind failed refactoring attempts, we carefully analyzed and categorized each one of the attempts. Table 5 details the categories of failed attempts and their frequencies.

- *Invalid source code selection* – Attempts where Pharo provides error the messages such as "invalid source code selection" at the bottom left of the window. This error only appears when one uses the standard Pharo tools since TOAD proposes only valid alternatives.

- *Errors* – Both treatments have some errors that appeared during our experiment. In the case of the Pharo Tool, it mainly shows two errors: i) a null pointer exception and ii) cannot extract assignments to

[5]Having the two identical distributions means that there is a 50% probability that a success rate randomly picked for Pharo Standard is greater than a success rate randomly picked for *TOAD*.

Table 5: Categorization of Extract Method attempts

Pharo Standard Tool	
Category	**Frequency**
Change as expected	62 (50,82%)
Invalid source code selection	21 (17,21%)
First was sent to nil	15 (12,30%)
Canceled Operations	11 (9,02%)
Cannot extract assignments to temporaries	9 (7,38%)
New parameter required	2 (1,64%)
TOAD	
Category	**Frequency**
Change as expected	61 (74,39%)
Cascade message not allowed	11 (13,41%)
Operation cancelled	7 (8,54%)
Different argument order required	2 (2,44%)
New parameter required	1 (1,21%)

temporaries, which is an error triggered by the IDE itself, and not the refactoring engine.

- *Operation cancelled* – When participants change their mind and simply cancel the refactoring by pressing the *cancel* button in the user interface.

- *Missing Options* – In three attempts, participants wanted to add a method argument while creating the new extracted method, but Pharo did not allow this feature, despite the fact that our learning material explicitly mentions this restriction. Therefore, after extracting a new method, they manually added the argument they wanted. In the case of TOAD, users wanted to change the order of the arguments, but our prototype does not provide this capability. So, the participant manually performed this transformation.

When compared with the standard Pharo refactoring tool, *TOAD* significantly reduces the number of failed attempts to apply the Extract Method refactoring.

Refactoring Alternatives. From the text selection provided by a participant, *TOAD* offers five possible text selections and their

corresponding preview of the refactoring. When *TOAD* opens, the first proposed selection is the one made by the participant (or a close selection if the provided one is incorrect). After entering the name for the new extracted method, the user can either accept the selection provided by *TOAD* or choose an alternative one.

Table 6 details the number of times each participant inspected *TOAD* and how many times the participants changed their refactoring intentions to one alternative provided by *TOAD*. Overall, participants inspected an alternative text selection in 43% of attempts and they selected an alternative selection in 27% of attempts.

Table 6: Participants Refactoring Alternative Usage Frequency

Participant	**Total Attempts**	**Alternative Inspection**	**Alternative Selection**
P1	10	5	2
P2	8	3	1
P3	7	5	4
P4	20	8	7
P5	6	5	2
P6	3	2	2
P7	6	2	0
P8	10	0	0
P9	4	3	3
P10	8	2	1
TOTAL	**82**	**35 (43%)**	**22 (27%)**

Participants consulted, in a large proportion, alternative text selections (43% of the refactoring attempts) and used an alternative text selection instead of their selection (27%) when doing an extract method refactoring.

4.2.2. Task Load & Participants Feedback

The NASA-Task Load Index (TLX) is a widely used technique for measuring subjective mental workload [4]. The NASA TLX is a questionnaire in which participants estimate their cognitive workload using numerical scales. Six different scales are considered: mental demand, physical demand, temporal demand, performance, effort, and frustration. Figure 6 summarizes the participant perception about the task they were asked to do.

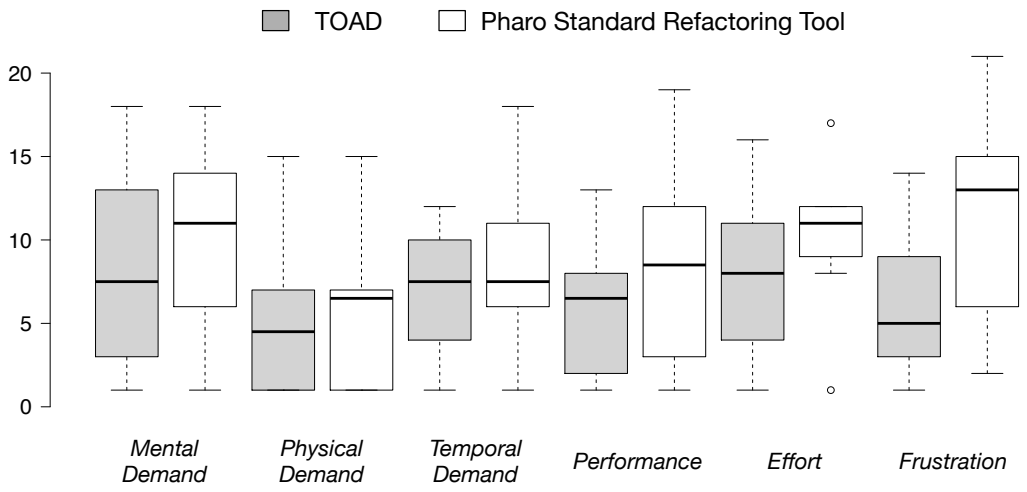

Figure 6: Task Load: Participant Perceptions

Mental Demand. Five participants indicated that the task was more demanding while using the Pharo standard tool. Some participants reported some effort in selecting a correct code segment for extraction, mainly because the method they were refactoring was complex to analyze. Four of them indicated they had the same mental demand with both tools. One of them indicated that it was easier to refactor with the standard tool since the code was easy to understand and he/she knew exactly what to extract.

Think-aloud protocol. We encouraged the participants to follow the think-aloud protocol to collect information that were not apparent from the interactivity. Some participants felt uncomfortable with this, and preferred to stay quiet instead.

Performance and Temporal Demand. Since we did not impose a time limit on the participants, all the participants performed the activity until they were satisfied with the new version of the method. There is no notable difference in terms of performance and temporal demand. All of them indicated that they performed the task at almost the same level of success.

Effort and Frustration. Nine out of ten participants indicated that they required less effort using *TOAD* and had less frustration. They manifested that *TOAD* helped focus on which parts they should refactor instead of

focusing on whether the source code selection met the refactoring preconditions. Even when some of them did not choose the alternatives provided by *TOAD*, the tool shows the closest code segment that satisfies the preconditions.

Total Cognitive Load. Figure 7 presents the distribution of the total cognitive load. After using a treatment we asked the participant to fill the NASA TLX. We collected two data points for each participant: the cognitive footprint for *TOAD* and another footprint for the *PharoStandard* refactoring tool. Each participant has an overall score defined as the sum of each workload scale. A low value indicates that the task was not cognitively demanding while a high value indicates that the task was cognitively demanding.

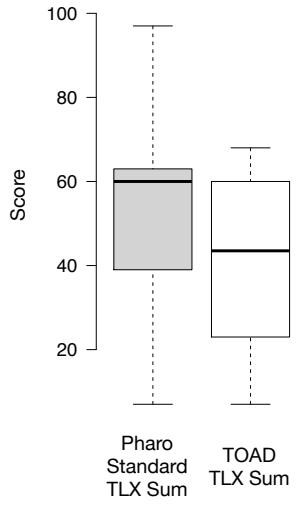

Figure 7: Total cognitive load (lower is better)

We executed Wilcoxon matched-pairs signed-rank test over the total cognitive load (two-tailed, confidence level = 95%). The null hypothesis is that the distribution of the cognitive load across participants is the same for both *TOAD* and *PharoStandard*. The test indicated that the distribution is statistically different (P-value = 0.0391, Sum of signed ranks (W) = 35). **Therefore, we can reject the null hypothesis and conclude that there is a causal effect between the treatment (*i.e.,***

***PharoStandard* or *TOAD*) independent variable and the total cognitive load.**

Adding the refactoring alternatives as an option in the refactoring tools does not perceptibly overload the refactoring process and reduce the participants' cognitive load.

5. Threats to Validity

5.1. Internal Validity

Participants. Participants' experience with Pharo may influence the results. If we compare Table 1 and Table 4, we see that P2 (0.5 years of experience) has a significantly high failure rate using Pharo tools (93%), but a much lower failures rate (25%) using TOAD. P10 (15 years of experience) has an almost identical failure rate using Pharo tools (38%) and TOAD (37%). This suggests that experience has a relevant influence on the results.

Baseline. We compare our tool against the Pharo standard refactoring tool which has a different way to display the refactoring preview than *TOAD* has. During the feedback session, we asked participants about the source code preview. Although they manifested that *TOAD* has a good preview visualization, they did not feel that the preview influences the refactoring task. Also, two participants manifested that they focused on analyzing the source code they selected instead of the preview both tools provide.

Refactored Method. The size and complexity of candidate methods to refactor may be a bias in our results. To mitigate this issue, we first selected the longest methods in each project together with each participant. Then, we randomly assigned the order in which these methods should be refactored without favoring any tool.

Required task effort. We did not want to pressure participants in a way that could hamper the representativity of our experiment. As a consequence, we did not monitor the time to complete those tasks. From our pre-experimental pilot and after having run the experiment, we did not find evidence that the tasks require different efforts to complete. We therefore conclude that the tasks are comparable.

Learning Effect. To reduce the learning effect between experiments, during each session, we randomly selected which treatment will be used first. In the end, we had five participants that used *TOAD* at first, and five participants that used the Pharo Standard Tool.

TOAD Implementation. *TOAD* implements a simple approach to show a number of source code selection alternatives for extracting methods. There are different options to compute the selection alternatives and show them to the participant. However, the goal of *TOAD* is to show that a simple approach is enough to gain in usability.

Unexpected Errors. As reported in the result section, both TOAD and the standard Pharo refactoring tool raised a number of errors during the experiment. From one side, the Pharo tool reports a number of null pointer exceptions, on the other side TOAD report a number of parsing errors due that the standard Pharo parser could not handle particular AST combinations with cascade messages. However, we believe that these errors do not jeopardise our results since these errors rarely appear and in almost the same frequency.

Cognitive load. The comparison of the survey of cognitive load suggests that TOAD has lower cognitive load. It is likely that looking for alternate code selections reduces cognitive load, while previewing the alternate refactoring increases the cognitive load. However, this increase, if it exists, it is not perceptible by the participant.

5.2. External Validity

Pharo Programming Language. Our experiment focused on the Pharo programming language. Although refactoring tools have similar user interfaces and options along most IDEs in different programming languages, we have no evidence about the impact of refactoring alternatives in other IDEs. However, as we see in our replication study, *IntelliJ IDEA* has similar usability issues than the Pharo Standard refactoring tools. Therefore, our findings will be useful for IDEs other than Pharo.

Other Refactoring Options. The notion of refactoring alternative presented in this article is likely to be applicable to other refactorings, including pushing / pulling refactoring, extract temporal variables. We have no evidence that it is not the case. However we leave this as future work.

6. Related Work

Previous researchers investigated the lack of trust in refactoring tools. In addition to usability problems investigated in our replication study, developers do not understand what most of the refactorings do [11], and they regularly face overly strong preconditions in current IDEs [8]. Consequently, developers prefer to perform changes manually and sometimes repetitively, even though there is automated support in the IDE.

Previous studies on the usability of refactoring tools [10, 15, 16] record Eclipse IDE usage information from developers, for instance, which commands they executed, failures from automated refactorings, the context of the failure, among others. Complementary to these studies, we video-recorded refactoring sessions from participants, where we asked participants to speak about their refactoring intentions and expectations (*i.e.*,, following the think aloud protocol). Additionally, we experimented with the IntelliJ IDE and the Pharo Programming Environment.

These challenges lead researchers to improve the experience of wizard-based refactoring, such as the ones proposed in most IDEs. Lee *et al.* [5] proposed drag-and-drop actions to invoke the refactoring tool. Ge *et al.* [2] proposed to detect manual refactoring actions and recommend additional code changes to automatically complete the refactoring. Maruyama and Hayashi [6] proposed to record a failed refactoring attempt; then the tool automatically resumes the refactoring when preconditions are satisfied by further code editions.

Identifying refactoring opportunities have also been proposed in the past. Tsantalis *et al.* [14] proposed an approach to help developers to identify code fragments that may be extracted to new methods. Mkaouer *et al.* [7] proposed a tool that suggests refactoring opportunities to developers based on their feedback and introduced code changes.

Our tool takes a different stance by guiding the developer to achieve a correct code selection. Murphy-Hill and Black [9] proposed a similar approach to assist code selection by highlighting the entirety of a partially selected statement, such as the one presented in Figure 4. However, the correct selection of a statement may not be sufficient to meet all the preconditions of the Extract Method refactoring. External access to variables and ambiguous return values may also trigger refactoring errors. TOAD searches and proposes multiple code selections including the one selected by developer, and not only the closest selection. The recommended

code selections are previously tested, thus the tool guarantees that the preconditions are satisfied.

7. Conclusions

This article investigated the usability issues from the classical Extract Method refactoring. We first presented a replication study to highlight and analyze usability issues that developers face using *IntelliJ IDEA*'s refactoring tools. Our results are comparable with the original study by Vakilian *et al.* [16], in which they found similar usability issues. Their experiment was conducted on Eclipse and they did not associate usability issues with the practitioner's perception, such as unexpected source code modifications or confusing error messages.

We then described *TOAD*, our solution to address the issue of wrongly selecting code to be refactored. *TOAD* recommends refactoring alternatives, which tackles some of these usability issues, such as invalid code selections and unexpected source code changes. Our evaluation of *TOAD* indicates that it has a significant impact on reducing the number of failed refactoring attempts, while participants benefit from a decrease of cognitive load when compared with the standard Pharo refactoring engine.

As a future work, we plan to research on the consequences of having a preview window. In particular, we will measure how the preview window impacts practitioners when picking particular alternate refactorings.

Acknowledgements

We are deeply grateful to Lam Research(4800054170 and 4800043946) and the FONDECYT project 1200067 for having partially sponsored the work presented in this article. We thank Renato Cerro for his help in reviewing an early draft of the manuscript.

References

[1] Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA.

[2] Ge, X., DuBose, Q.L., Murphy-Hill, E., 2012. Reconciling manual and automatic refactoring, in: 2012 34th International Conference on Software Engineering, ACM. pp. 211–221.

[3] Griswold, W.G., 1992. Program Restructuring As an Aid to Software Maintenance. Ph.D. thesis. Seattle, WA, USA. UMI Order No. GAX92-03258.

[4] Hart, S.G., Staveland, L.E., 1988. Development of nasa-tlx (task load index): Results of empirical and theoretical research. Human mental workload 1, 139–183.

[5] Lee, Y.Y., Chen, N., Johnson, R.E., 2013. Drag-and-drop refactoring: Intuitive and efficient program transformation, in: 2013 35th International Conference on Software Engineering, ACM. pp. 23–32.

[6] Maruyama, K., Hayashi, S., 2017. A tool supporting postponable refactoring, in: Proceedings of the 39th International Conference on Software Engineering Companion, IEEE Press. pp. 133–135.

[7] Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., Ó Cinnéide, M., 2014. Recommendation system for software refactoring using innovization and interactive dynamic optimization, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ACM. pp. 331–336.

[8] Mongiovi, M., Gheyi, R., Soares, G., Ribeiro, M., Borba, P., Teixeira, L., 2018. Detecting overly strong preconditions in refactoring engines. IEEE Transactions on Software Engineering 44, 429–452.

[9] Murphy-Hill, E., Black, A.P., 2008. Breaking the barriers to successful refactoring: Observations and tools for extract method, in: Proceedings of the 30th International Conference on Software Engineering, ACM. pp. 421–430.

[10] Murphy-Hill, E., Parnin, C., Black, A.P., 2012. How we refactor, and how we know it. IEEE Transactions on Software Engineering 38, 5–18. doi:10.1109/TSE.2011.41.

[11] Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D., 2013. A comparative study of manual and automated refactorings, in: Proceedings of the 27th European Conference on Object-Oriented Programming, Springer-Verlag, Berlin, Heidelberg. pp. 552–576.

[12] Opdyke, W.F., 1992. Refactoring Object-oriented Frameworks. Ph.D. thesis. Champaign, IL, USA. UMI Order No. GAX93-05645.

[13] Siles Antezana, A., 2019. Toad: A tool for recommending auto-refactoring alternatives, in: Companion Proceedings of the 41th International Conference on Software Engineering.

[14] Tsantalis, N., Chatzigeorgiou, A., 2011. Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software 84, 1757 – 1782.

[15] Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E., 2012. Use, disuse, and misuse of automated refactorings, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA. pp. 233–243. URL: http://dl.acm.org/citation.cfm?id=2337223.2337251.

[16] Vakilian, M., Johnson, R.E., 2014. Alternate refactoring paths reveal usability problems, in: Proceedings of the 36th International Conference on Software Engineering, ACM. pp. 1106–1116.