Empirical Software Engineering (2019) manuscript No.
(will be inserted by the editor)

RePOR: Mimicking humans on refactoring tasks. Are we there yet?

Rodrigo Morales[1] · **Foutse Khomh**[2] · **Giuliano Antoniol**[2]

Received: April 2019 / Accepted: –

Abstract Refactoring is a maintenance activity that aims to improve design quality while preserving the behavior of a system. Several (semi)automated approaches have been proposed to support developers in this maintenance activity, based on the correction of anti-patterns, which are "poor" solutions to recurring design problems. However, little quantitative evidence exists about the impact of automatically refactored code on program comprehension, and in which context automated refactoring can be as effective as manual refactoring. Leveraging RePOR, an automated refactoring approach based on partial order reduction techniques, we performed an empirical study to investigate whether automated refactoring code structure affects the understandability of systems during comprehension tasks. (1) We surveyed 80 developers, asking them to identify from a set of 20 refactoring changes if they were generated by developers or by a tool, and to rate the refactoring changes according to their design quality; (2) we asked 30 developers to complete code comprehension tasks on 10 systems that were refactored by either a freelancer or an automated refactoring tool. To make comparison fair, for a subset of refactoring actions that introduce new code entities, only synthetic identifiers were presented to practitioners. We measured developers' performance using the NASA task load index for their effort, the time that they spent performing the tasks, and their percentages of correct answers. Our findings, despite current technology limitations, show that it is reasonable to expect a refactoring tools to match developer code. Indeed, results show that for 3 out of the 5 anti-pattern types studied, developers could not recognize the origin of the refactoring (i.e., whether it was performed by a human or an

Rodrigo Morales
rodrigomorales2@acm.org

Foutse Khomh
foutse.khomh@polymtl.ca

Giuliano Antoniol
giulio.antoniol@polymtl.ca

[1] Department of Computer Science and Software Engineering, Concordia University, Montréal, Canada
[2] Département de génie informatique et génie logiciel, École Polytechnique de Montréal, Montréal, Canada

automatic tool). We also observed that developers do not prefer human refactorings over automated refactorings, except when refactoring Blob classes; and that there is no statistically significant difference between the impact on code understandability of human refactorings and automated refactorings. We conclude that automated refactorings can be as effective as manual refactorings. However, for complex anti-patterns types like the Blob, the perceived quality achieved by developers is slightly higher.

1 INTRODUCTION

In 1950, Alan Turing developed a test to assess a machine's ability to display behavior equivalent to that of a human being [38]. The evaluator (human) will be exposed to a blind conversation with a machine and another human, and by formulating questions (s)he will try to identify his interlocutor (i.e., whether it is the human or the machine). If the evaluator cannot distinguish between human and machine, the latter one is said to have passed the test. In this paper, we conduct an experiment inspired by the Turing test. We want to check whether RePOR our automatic software refactoring tool [20] can be as effective as human developers at least from the refactoring structure point of view. Indeed, one major limitation of RePOR, and existing refactoring tools, is the lacking of semantic and contextual information of identifiers. To have a fair comparison in our study, for a subset of refactoring actions, we used synthetic identifiers also for human refactored code and asked developers to judge the soundness of refactoring actions and code structure. It is important to underline, RePOR evaluation goal was not to verify if the refactoring results were deemed useful or necessary by the original developers or if original developers would have performed the same code changes. The overarching goal was the verify if RePOR was capable to produce refactoring actions and code comparable to human changes quality wise.

To this aim (1) we presented practitioners with refactorings performed manually by developers and refactorings performed by RePOR, and asked them to identify the origin of the refactorings. We also asked them to judge the design quality of the refactorings. Next, (2) we asked practitioners to perform a series of comprehension tasks on (both manually and automatically) refactored code and we assessed their performance.

Context: Software systems age as the result of deviations of their original design due to the implementation of new features [31], changes in the business logic, etc. One way to combat software system deterioration is to perform maintenance activities continuously; correcting poor design choices (*a.k.a.,* anti-patterns) [5] for example through refactorings [8]. Refactoring is a software maintenance activity that aims to reorganize code structure without altering system's behavior [8]. In fact, agile methodologies like *eXtreme Programming (XP)* encourages developers to interleave refactoring with their code tasks to ease software evolution. However, refactoring is a time-consuming activity because (1) developers have to identify software components that contain anti-patterns; (2) select the adequate refactorings to clean-up the code; and (3) select the best order to apply the refactorings determined in the previous step. Yet, the benefits of code refactoring to combat software aging can only be observed in the long term.

Previous works have studied refactoring practice in academic and industrial settings, and found that refactoring is a common practice, that the refactorings manually performed by developers differ from those applied using tool support, and that tool support is underused due to a lack of awareness [10, 25, 39]. In the last decade, tools

and frameworks supporting refactorings have been proposed [18, 29, 22, 23]. However, to the best of our knowledge, little effort has been taken to evaluate the impact on code understandability of automated-refactoring compared to manual refactoring. As a consequence, many developers express natural reluctance about applying automated refactoring to their code bases [15]. To accept or reject the notion that automated-refactoring can replace manual one, we propose to submit the refactorings applied by human and machine to the scrutiny of software developers, who will act as judges, similar to a Turing test, to assess the quality, pertinence, and understandability of automatically generated code structure. Our goal is to verify if RePOR challenges the idea that automated refactorings can't compete with refactorings performed by humans, in terms of quality, according to human judges.

Premise: Refactoring is conjectured in the literature to improve the design quality of systems. Despite the large number of studies on refactoring summarized in Section 2, few studies have empirically investigated the impact of automated refactoring tools on program comprehension. Yet, program comprehension is crucial to practitioners responsible of performing software maintenance. Hence, a better understanding of the conditions in which automated- is as suitable as manual-refactoring can promote the development of more human-like tools that perform automation efficiently, reducing maintenance costs.

Goal: We want to gather quantitative and qualitative evidences that automated-refactoring can succeed in improving the design quality of a system at least as well as a human being would do in similar conditions.

Study: We perform two experiments: (1) a preliminary refactoring survey (E_1) where we invited developers from Java mailing lists, and technical groups on social networks, to differentiate refactoring code changes (that aimed to remove anti-pattern's instances) generated by a software tool (machine) from those generated by freelance developers (human). For simplicity, we called this test in the consecutive "Turing test", acknowledging the differences on implementation, as a Turing tests typically require human interaction between human judges and machine in the form of natural language. The freelancers that refactored the code were hired from two well-known crowdsourcing marketplace websites[1]. We also asked the participants of the refactoring survey to rank the refactorings according to their quality. (2) We conducted a series of code comprehension experiments (E_2) where we studied whether the refactoring code changes generated by tool are more difficult to understand than those generated by developers. In E_1, we surveyed 80 developers using code from 10 different Java systems (two study groups, manual and automated code changes). In E_2, we hired 30 more additional developers, from the two aforementioned crowdsourcing marketplaces, to perform two comprehension tasks covering three out of the four categories of comprehension questions identified by Sillito et al. [36]. We measured the subjects' performance using: (1) the NASA task load index for their effort; (2) the time that they spent performing the tasks; and, (3) their percentages of correct answers.

Results: From the collected data, we observe that: (1) the ability of developers to recognize automatically-generated refactoring changes depends on the types of anti-patterns removed. For example, automatically generated refactoring changes to remove Blob, Spaghetti Code and Lazy Class anti-patterns are hard to distinguish, while those generated for correcting Long Parameter List and Speculative Generality anti-patterns are not. (2) Developers do not have preference between refactoring changes generated by

[1] Freelancer.com and Guru.com

humans or by machine (i.e., automated tools), except for those that correct Blob classes. (3) There is no statistically significant difference between the impact of automated- and manual- refactorings on program comprehension.

Relevance: Understanding the context in which automated-refactoring is beneficial to developers and the cases where a human supervision is required is crucial from the points of view of both researchers and practitioners. For researchers, our results debunk the myth that automated-refactoring is not reliable and-or cannot be safely performed without human intervention. At the same time, they also highlight the need to develop approaches able to create identifiers that are semantically and contextual dependent. For practitioners, our results build confidence in the adoption of automated-refactoring tools. We hope that our results serve as inspiration to both groups, and help them develop new tools and approaches to support software maintenance and evolution.

Organization: Section 2 relates our study with previous works. Section 3 presents the results of a preliminary study to assess the quality of the automated refactoring changes proposed by our tool. Section 4 describes the design of our empirical study. Section 5 presents the study results, while Section 6 discuss further the obtained results and their implications. Section 7 discusses threats to validity. Finally, Section 8 concludes the paper and discusses avenues for future work.

2 Related Work

Fowler [8] popularized the term refactoring, by defining some heuristics to improve the design of existing code while preserving its functionality. Brown [5] introduced the notion of anti-patterns as poor-design choices that hinders code evolution. Opdyke [28] is the first to formulate a set of pre- and post-conditions to automatize the refactoring of object-oriented systems. Mens et al. [17] published an extensive overview of existing research work of software refactoring, and there are still new works published every year.

In this section, we summarize some of the works related to refactoring and its impact on design quality and code comprehension, and relevant studies that compared manual and automatic refactoring.

2.1 Impact of Refactoring anti-patterns on design quality and code comprehension

Deligiannis et al. [7] proposed the first quantitative study of the impact of anti-patterns on software development and refactoring. Through a controlled experiment involving 20 students and two systems, they found that Blob classes hinder the evolution of software design and the subject's use of inheritance. This work did not evaluate the understandability of the code, neither the subjects' ability to successfully perform comprehension tasks on the systems studied.

Stroulia and Kapoor [41] investigated the impact of refactoring on metrics associated to size and coupling and found that those metrics improved after refactoring.

In another academic setting, Du Bois et al. [4] found that the refactoring of God Classes into a number of collaborating classes can improve code understandability. The participants were asked to perform simple refactoring to decompose God classes. They found that participants exhibited less difficulties to understand the refactored code.

Murphy et al. [26] performed an empirical study on refactoring practice and the use of refactoring tool support. By analyzing the commit history of more than 39,000 Eclipse developers, with their interaction traces (using Mylyn plug-in), they found that: (1) refactoring tooling support is rarely used; (2) that 40% of refactorings assisted by tools occur on batch; (3) developers prefer interleaving refactoring with other code activities; (4) the kind of refactorings performed with tools differs from the kind of refactorings performed manually. However, this work did not study the impact on code comprehension of refactorings performed by machine and human, and what is the take of developers on automatically refactored code. These questions that were left unanswered served as motivation to perform our study.

Abbes et al. [1] performed an empirical study to determine the impact of Blob and Spaghetti Code anti-patterns on program comprehension. They performed three experiments, each of them with 24 subjects and three different Java systems. They measured the subjects performance using three metrics: NASA TLX, tasks' completion times, and percentage of correct answers. They found that the occurrence of one single instance of anti-pattern does not impact significantly the comprehensibility of the code, while the combination of two anti-patterns does impact comprehension negatively.

Moser et al. [24] analyzed the impact of refactoring on productivity in agile environments. They measured developer's productivity by dividing lines of code and effort (measured in time). They reported a statistically significant increase in productivity after refactoring along with some improvements in complexity and coupling.

Kim et al. [14] performed a case study in an industrial setting at Microsoft on the benefits of refactoring. They found that developers perceive manual refactoring as expensive and sometimes risky, and that refactoring changes loosely match the literature definition of semantics-preserving code transformations. They also reported that the top 5 percent of preferentially refactored modules in Windows 7 reported higher reduction in the number of inter-module dependencies and several complexity metrics, but as a consequence larger increase in module's size compared to the remaining 95% of the refactored modules.

All these studies corroborate the idea that refactoring anti-patterns improve code design quality in several dimensions. However, all of them focused on manual refactoring.

2.2 Manual compared to automatic refactoring studies

Negara et al. [27] performed an empirical study to compare semi-automatic (IDE tool-assisted) refactoring and manual refactoring operations of 23 participants from the academy and industry in a controlled experiment. They reported that developers performed 11% more refactorings manually than using tool support; less experienced developers use less tool support than experienced ones; that developers perform large refactoring changes (*e.g.,* extract method) with and without tool support, and that the size of a refactoring is not a decisive factor.

Szőke et al. [37] performed a case study with a R&D company to study the effects of semi-automatic refactoring on code maintainability. They corroborate the statement by Murphy et al. [26] that the quality of machine-generated code might differ from manual refactoring when developers are prompted to provide input to the tool to decide between a list of refactorings, and when they blindly select the default machine propositions. They conclude that companies could achieve a considerable increment of code maintainability by only applying automatic refactorings. Note that they did

not compare an equal number of manual and automatic refactorings, but commanded developers to perform refactoring manually and then developed a tool to reproduce manual refactoring behavior followed by developers. The level of automation of their proposed tool is not disclosed.

Note that none of the studies in this category compared the understandability of manual and automatic refactoring on the same source code, and–or developers' perceived quality of the refactored code.

2.3 Evaluation of Refactoring process

Kataoka et al. [13] proposed an approach to measure the effect of refactorings on the maintainability of the code, using coupling metrics. Although, they find that coupling metrics can be useful to asses the degree of maintainability improvement of a refactoring change, they also recognized that the type of refactorings that can be evaluated using coupling metrics is limited,

In a recent preliminary work by Arima et al. [2], a novel technique is proposed to assess the *naturalness* of refactored code automatically, using probabilistic language models. *Code naturalness* is a numerical value that measures how natural a given word sequence is for the model. As oracle they used a curated dataset of commits belonging to JUnit, where the refactorings were clearly identified by the authors of the commits. Their approach achieved 68% of accuracy on 28 refactoring operations. This study does not evaluate the impact of refactoring, but tried to propose a new metric to quantitatively measure how well does a refactoring matches the text content of a software system. In the future, this technique could serve for further comparisons of automatically and manually refactored code in a quantitative way, and to improve existing automatic refactoring tools.

3 Preliminary Study

In this section we performed a preliminary study to assess the quality of refactorings suggested by RePOR. It is about recognizing if a refactoring code change applied to remove an instance of the aforementioned anti-pattern types was performed by a developer, or an automatic tool. We also asked participants to rank the refactoring code changes according to their perceived quality.

3.1 Preliminary Research Questions

Our preliminary research questions stem from our goal of checking whether or not automated tools can do as good a job as manual developers. Looking at quality and checking whether people could easily distinguish them are just two indicators that we used to achieve that goal. We want to check if the two types of refactorings (i.e., refactoring made by humans and refactorings made by a tool) are significantly different structurally. Our reasoning is that if the two types of refactorings are not too different structurally, then automated tools are likely to be adopted more easily by practitioners. Although, there exist other alternative approaches, to assessing quality and checking if refactorings can be easily differentiated based on their origin (i.e., tools vs humans),

we believe that our analysis can provide key information on whether or not developer perceive refactoring performed by humans and tools to be equivalent.

We state the preliminary questions as follows.
PQ1: Can developers tell the difference between automated and manual refactorings?
PQ2: Is there any difference between the perceived quality of automated and manual refactoring?

3.2 Hypotheses

For PQ1, we test the following null hypothesis when subjects review refactored code changes.

$H_{0identify}$: Developers correctly differentiate the refactoring changes generated by an automated tool from the ones by another developer.

For PQ2, to examine differences in the perceived quality of the refactoring changes, we test the following null hypothesis.

H_{0rank}: Developers prefer manual refactoring over automated refactoring.

3.3 Objects

The selection criteria of the anti-pattern instances refactored in this study is based on the detection results of our own tool for detecting and refactoring anti-patterns RePOR [20] applied to the SF110 corpus [9]. SF110 is a representative sample of 110 Java projects from *SourceForge* Website[2], which is a popular open source software repository, with more than 500,000 projects and with more than 33 million registered users. The projects in SF110 corpus are packed into a common build infrastructure, including developers tests suites and automated-generated test suites previously validated by the authors of the corpus. We explicitly asked freelancers that performed the manual refactoring to run developer's unit tests, and in case the refactoring(s) affected the execution of the unit tests, due to change in the code structure, we asked them to update the corresponding unit tests, as developers will normally do in such cases. We followed the same procedure for automated refactorings. This help us to validate, to some extent, that regression is not introduced in the code after refactoring. Once we collected the results of the anti-patterns detection, the first author of this work and a Master's student in our research lab selected two instances, for each anti-pattern type studied, that were deemed representative examples of anti-patterns based on the works of Brown et al. and Fowler et al. [5, 8]. In case of disagreement, we came to the second author of this work for solving differences, and followed a conservative approach of discarding instances where we could not reach consensus. We are also interested to know if different refactoring types affect the perception of developers or not. However, study only one single instance of each type would be insufficient as one instance could be intrinsically easier or more difficult to refactor. Hence, we opted for selecting one easy and one hard instance of each anti-pattern type. In total we study 10 Java systems (two for each anti-pattern type) from different domains and sizes. We select the anti-pattern examples from different systems for both treatments (automated

[2] https://sourceforge.net/

Table 1: List of studied Anti-patterns types and the refactorings used to correct them.

Name	Description	Refactoring(s) strategy
Blob (BL) [5]	A large class that absorbs most of the functionality of the system with very low cohesion between its constituents. In addition, Blob Classes are surrounded by classes who serve mainly as data holders, and that does not implement any functionality (*a.k.a.,* data classes).	*Move Method (MM).* Move the methods that does not seem to fit in the Blob class abstraction to more appropriate classes [33]. Another strategy, when there are not suitable classes to move methods, is the creation of new classes with methods and attributes that have high cohesion, and that are semantically related (*a.k.a.,* extract class refactoring).
Lazy Class (LC) [8]	Small classes with low complexity that do not justify their existence in the system.	*Inline Class (IC).* Move the attributes and methods of the LC to another class in the system.
Long Parameter List (LP) [8]	A class with one or more methods having a long list of parameters, specially when two or more methods are sharing a long list of parameters that are semantically connected.	*Introduce Parameter Object (IPO).* Extract a new class with the long list of parameters and replace the method signature by a reference to the new object created. Then access to this parameters through the parameter object.
Spaghetti Code (SC) [5]	A class without structure that declares long methods without parameters.	*Replace Method With method Object (RMWO).* Extract long methods into new classes so that all local variables become attributes on that object.
Speculative Generality (SG) [8]	There is an abstract class created to anticipate further features, but it is only extended by one class adding extra complexity to the design.	*Collapse Hierarchy (CH).* Move the attributes and methods of the child class to the parent and remove the *abstract* modifier.

and manual), to control for possible learning effect on the respondents, who might get familiar with a system's code design.

The anti-patterns types studied are briefly introduced in Table 1. We provide the name, a brief description and the refactoring strategies used to remove them according to the literature of anti-patterns [8, 5].

In Table 2 we present information about the systems studied. The ID column contains the ID assigned by SF110 (from now on, we use it to reference the system) to a system, anti-pattern type (Ap. Type), package where the source class is located, and the source class containing the anti-pattern instance.

In Table 3, we present the metrics used to assess the difficulty of anti-pattern instances studied. The last tree columns measured the normalized entropy of refactoring changes applied to those anti-patterns performed by tool and developer respectively. We decide on the level of difficulty of the instances based on (1) expert judgment and (2) code metrics. (1) We assess how difficult it would be to refactor the anti-pattern instance in question if we had to do it ourselves. (2) We use code metrics to validate our expert judgment. For example, in the case of Blob class, we evaluate the effort of manually refactoring the class by moving methods to related classes, or extracting new ones. Then, we took into account lines of code, and number of data classes associated to Blob to determine which instance required more effort. With respect to Collapse Hierarchy and Lazy Class, we measure the *number of incoming invocations (NII)*, as we suggest that removing these classes requires to update all method calls to the inlined classes, which in turn requires extra effort when values of NII are large. For Spaghetti Code, we measured *McCabe Complexity (CC).* For Long-parameter list classes, we consider

Table 2: List of systems from where we extract the anti-pattern's instances studied

ID	System	Description	Ap. Type	Package	Source class
110	FireBird	A relational database manager	Blob	org.firebirdsql.gds .impl.jni	isc_stmt_handle _impl
83	Xbus	It is a central Enterprise Application Integration (EAI)	Blob	net.sf.xbus. technical.mail	POP3XML Receiver
52	Lagoon	A XML-based framework for web site maintenance	Collapse hierarchy	nu.staldal.laggon .core	FileRead
86	Advanced T-Robots	Fighting robots Arena	Collapse hierarchy	net.virtualinfinity .atrobots.arena	LinearDamage Function
47	dvd-homevideo	Application to burn DVDs	Lazy class	default package	SaveStackTrace
101	SAP NetWeaver	Server Adapter for Eclipse	Lazy class	com.sap.netweaver .porta.mon	CommandFactory
103	Sweet Home 3D	An interior design application	Long-parameter List	com.eteks. sweethome3d.j3d	HomePieceOf Furniture3D
104	Vuze	BitTorrent Client	Long-parameter List	org.gudy.azureus2. core3.tracker. client.impl.bt	TRTrackerBT AnnouncerImpl
81	JavAthena	Online role playing game	Spaghetti Code	org.javathena.core .data	ROCharacter
70	EchoDep	Digital preservation application	Spaghetti Code	edu.uiuc.ndiipp. hubandspoke. workflow	WorkflowManager

the number of parameters of methods in a class. We did not use fixed thresholds for discriminating easy from hard instances as these values are relative to the systems in question. Instead, we leverage the information obtained by our appreciation of the required effort, and code metrics, and based on that we defined the difficulty. To ensure the refactorings performed by developers and the tool were comparable in terms of effort, we measured the normalized static entropy (N.E.) [11] to quantify the effort required to refactor an anti-pattern based on the number of changes made by a developer/tool respectively (see columns N.E.D and N.E.T.). We observe that the refactoring effort, as measured with N.E., is very similar between machine and human treatments for most cases.

Finally, we report the use of synthetic identifiers for new code entities introduced after refactoring (S.N.). This column can take any of these values: A (all refactoring examples), H (just human example, in case the tool did not create any new entity, like in move method refactoring), and N (none of the examples).

Our benchmark of refactoring changes is comprised of two sets: 10 *high-level* refactorings applied automatically by our tool (M); and 10 *high-level* refactorings applied by real developers (D). A high-level refactoring is a refactoring composed of more than one low-level refactoring [28]. For example: Collapse Hierarchy, which is a high-level refactoring, is composed of the following low-level refactorings: one or more pull-up method/attributes; delete class; remove abstract modifier; update class references.

For set M, we refactored the selected anti-patterns' instances, using the Eclipse plug-in implementation of RePOR[3]. For set D, we hired five experienced Java developer freelancers from two well-known marketplace websites (Guru.com and Freelancers.com).

[3] `https://github.com/moar82/RefGen`

Table 3: List of anti-pattern instances studied, where LOC: lines of code, NII: number of incoming invocations, CC: McCabe complexity, N.E.T: Normalized entropy for automated refactorings, N.E.D: Normalized entropy for human refactorings, and S.N.: Use of synthetic names.

ID	Ap. Type	LOC	NII	CC	DataClass	NoParam	Difficulty	N.E.T.	N.E.D.	S.N.
110	Blob	27			1		Easy	0.94	0.94	H
83	Blob	595			2		Hard	0.89	0.93	H
52	Collapse hierarchy		0				Easy	0.97	0.97	N
86	Collapse hierarchy		3				Hard	0.92	0.92	N
47	Lazy class		14				Hard	0.95	0.94	N
101	Lazy class		1				Easy	0.88	0.89	N
103	Long-parameter List					8	Hard	0.86	0.89	A
104	Long-parameter List					5	Easy	0.85	0.88	A
81	Spaghetti Code			80			Easy	0.44	0.99	A
70	Spaghetti Code			93			Hard	0.59	0.62	A

We provide them with the definition of the studied anti-patterns and the Fowler's catalog of refactorings [8], to allow them to select the most adequate refactorings, according to their experience. One can argue that it would be more natural to collect refactorings from the original developers of the studied systems, as they are familiar with the code. However, it would be hard to first locate, and then convince the original developers from open-source systems, as most of contributors of open-source projects are volunteers, without any contractual relationship with the project. Instead, we assume an hypothetical scenario where a new developer is integrating to a project, and is assigned the task of refactoring the existing code.

Each freelancer performed two refactorings, and they were excluded from experiment 1 and 2 to avoid introducing bias in our results.

3.4 Subjects

We invited participants to participate in preliminary study E_1 through Java developers' mailing lists, and through the social network of the authors of this paper. In total we received 87 answers from which we filtered out responses that were incomplete, and those where participants who did not provide justifications for their choices. We ended up with 80 responses for this study. All participants were volunteers and could withdraw at any time from the study, for any reason. From the 80 responses, 57 participants declared software development as their main occupation, 7 participants declared their main occupation to be research, 6 work as software architects, 4 scrum masters, 4 students and 2 fall in the *Other* category. Among these 80 participants, 37 declared to work on both open-source and proprietary systems, 34 declared to work on proprietary systems and only 9 on open-source. Fifty percent (40 out of 80) of participants declared to have more than 5 years of experience as developers. Eighteen participants have more than 2 and up to 5 years of experience, 14 participants have between 1 to 2 years of experience, and 8 participants have less than one year of experience. We ran the

experiment from October 27th 2017 to January 29th 2018. Since, we invited participants through mailing lists and social networks, it is hard to estimate the response rate.

3.5 Independent variable

The independent variable for the E_1 is related to who refactored an anti-pattern instance, i.e., its origin, and it is a binary variable stating whether the refactoring was performed by an automatic tool or by a developer.

3.6 Dependent Variables

The first dependent variable is a binary variable stating whether or not the participant correctly identified the origin of the refactoring change. The second dependent variable is a categorical variable capturing the perceived quality of the refactoring solution proposed, on a scale of 1 to 5, where 1 is for "Poor" quality, and 5 is for "Outstanding" quality.

3.7 Questions

For E_1, we use an online survey system (Jotform[4]) where we present the refactoring change, the online link to repository of the system that contained the anti-pattern's instance, and the type of anti-pattern. To summarize the refactoring changes applied to the studied systems, we generate for each anti-pattern's instance refactored, a patch file using the diff command from the control version system (Git). A patch file contains the description of the changes made using diff notation, which is a unified format that developers and control version systems can understand. The link to the repository of the systems studied provide respondents with a complete reference of the refactoring changes applied. We also provide a link to the repository containing the original source code; for example, if they want to study deeper the impact of the refactorings applied, they could clone the repository and apply the patch. In Listing 1 we show a fragment of a refactoring change from our online survey.

Listing 1: Fragment of a refactoring change from the online survey.

```
1  diff --git a/src/main/java/org/character/data/TXTCharacter.java
       b/src/main/java/org/character/data/TXTCharacter.java
2  index df86a66..964b164 100644
3  --- a/src/main/java/org/character/data/TXTCharacter.java
4  +++ b/src/main/java/org/character/data/TXTCharacter.java
5  @@ -6,6 +6,7 @@ import java.io.FileReader;
6   import java.io.IOException;
7
8   import org.character.data.config.CharConfig;
9  +import org.javathena.core.data.Clazz007382383094620344;
10  import org.javathena.core.data.Friend;
11  import org.javathena.core.data.Hotkey;
12  import org.javathena.core.data.IndexedFastMap;
13 @@ -14,7 +15,6 @@ import org.javathena.core.data.PersistenteData;
14  import org.javathena.core.data.Point;
15  import org.javathena.core.data.ROCharacter;
```

[4] http:www.jotform.com

```
16   import org.javathena.core.data.Skill;
17  -import org.javathena.core.data.ROCharacter.JOB;
18   import org.javathena.core.utiles.Functions;
19
20   public class TXTCharacter implements PersistenteData<IndexedFastMap<Integer,
           ROCharacter>>
21  @@ -139,7 +139,7 @@ public class TXTCharacter implements
         PersistenteData<IndexedFastMap<Integer, ROC
22               currChar.setName (mainCharSL[2]);
23
24               tmpSplit = mainCharSL[3].split(",");
25  -            currChar.setClass_(JOB.parseFromValue (Short.parseShort (tmpSplit[0])));
26  +            currChar.setClass_(Clazz007382383094620344.parseFromValue (Short.parseShort
         (tmpSplit[0])));
27               currChar.setBase_level (Integer.parseInt (tmpSplit[0]));
28               currChar.setJob_level (Integer.parseInt (tmpSplit[0]));
```

In diff notation, a patch does not show the complete file, but only shows the code fragments that were modified. Those code fragments are called chunk. The lines starting with "+" indicate new lines added, and "-" lines removed. "@@" is the chunk header, where Git indicates which lines were affected. For example, line 5 in Listing 1 indicates that from file A (original source code represented by a "-"), 6 lines are extracted starting from line 6. From file B (refactored source code represented by a "+"), 7 lines are displayed, starting also from line 6 . The text after "@@" serves to clarify the context. Git tries to display a method name or other contextual information of where this chunk was taken from in the file.

We asked developers to answer whether the refactoring change was generated by a developer or a software tool. We also had an option “Unknown” that participants could select if they were unable to tell whether the refactoring change was performed by a developer or generated by a tool. To control for possible randomness in the answers, we asked participants to provide their level of confidence in their answers (on a scale of 1 to 5), and a brief explanation to support each answer. Since the quality of developers' solutions may be different from that of our automated approach, we asked participants to rate the quality of the refactoring changes (according to their perception of quality) and mark their preferred solutions. We also asked them to provide any additional comment about the refactoring change's quality, if they considered it appropriate.

3.8 Anonymization of new code lexicon

Identifier names and comments (code lexicon), when thoughtfully assigned, can support developers to better understand a software system. However, as already stated, current existing automatic refactoring tools and frameworks are not equipped with mechanisms allowing them to generate a human-like name for a new class, or a new method introduced when refactoring. On the other hand, developers generate names that reflect the roles of the entities and–or follow projects guidelines. To ensure a fair experiment using the refactoring changes generated by humans and by RePOR, we decided to post-process the changes by removing any code comment, and renaming any new class or method added to the code base with an artificial name (for both origins), to avoid providing any hint that can lead the human evaluators to discover the origin of the changes. Note that renaming new classes and methods was only necessary for the following refactoring types: Introduce parameter-object, Replace method with object, and Extract class, while Inline Class, Collapse Hierarchy, and Move method did not require it.

Table 4: E_1 Experimental Design

Question	Group 1	Group 2	Group 3	Group 4
1	SC-83-M	BL-83-H	BL-110_M	LP-103-M
2	CH-86-H	SC-70-H	SC-70-M	SC-81-H
3	LC-101-M	CH-52-M	LP-104-H	LC-101-H
4	BL-83-M	LP-104-M	LC-47-0	BL-110-H
5	LP-103-H	LC-47-H	CH-52-H	CH-86-M

3.9 Design

For E_1, we divided the 20 refactoring patches (10 changes for each origin) into 4 groups. So each respondent will answer a survey containing 5 refactoring changes (each change corresponds to a different system) for removing 5 different types of anti-patterns. We also interleave the origin of the changes (machine, human) and the level of difficulty (easy and hard) in a way that any group has more than 3 patches from the same origin or level of difficulty. As previously stated, the level of difficulty is assessed using well known object oriented metrics. To assess feasibility, that time given to respond the survey was enough, and anticipate adverse events, we performed a pilot study with two Post-doctoral fellows and one Ph.D. student from our lab, with more than 5 years of experience developing with Java. The pilot study also helped us to refine the questions, and improve the visual design of our survey in terms of readability. Note that none of the people that participated in the pilot study took part to the final study.

In addition to asking respondents to identify the origin of the patch, we also asked them to justify their response in a free-text box, and to indicate how confident they felt when answering the questions, using a scale from 1 to 5, to control for possible randomness in the answers.

We present our design in Table 4. First column is the number of question and the rest of the columns corresponds to the different groups. Each cell contains the type of anti-pattern, using the abbreviations of Table 1, the ID of the system, and a letter indicating the origin of the patch (H:human, M: Machine).

3.10 Procedure

All the data collected is anonymous. The subjects could drop the experiment at any time, for any reason and without penalty of any kind. For the freelancers hired for refactoring the systems, and the second group hired for answering the comprehension tasks, we set milestones which clearly specify work deliverables, so we pay for each task completed. All the freelancers hired for refactoring the anti-patterns studied passed an interview, where they stated their experience as Java developers, and refactoring, and correcting anti-patterns. In addition to the anti-patterns definitions and refactoring strategies, we provided them with references from the literature and web sites related to refactoring, but we did not persuade them to blindly follow any of these materials, but encouraged them to base their actions on their work experience and own reasoning.

We first briefly introduced the description of the anti-patterns using Table 1. Next, we asked them, to not base their judgment on code lexicon, and to accept that they will only focus on code structure and its quality, and not on indentations, naming conventions, space, etc.

3.11 Analysis method

In PQ1, to attempt rejecting $H_{0identify:}$, we test whether the proportion of refactoring changes correctly identified (or not) by participants, significantly varies between changes generated by human or by machine. We use Fisher's exact test [35], which checks whether a proportion vary between two samples. We also compute the odds ratio (OR) [35] that indicates the likelihood for an event to occur. The odds ratio is defined as the ratio of the odds p of an event occurring in one sample, *i.e.,* the odds that changes generated by machine were correctly identified, to the odds q of the same event occurring in the other sample, *i.e.,* the odds that changes generated by humans were correctly identified: $OR = \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 indicates that the event is equally likely in both samples. An OR greater than 1 indicates that the event is more likely in the first sample (machine), while an OR less than 1 indicates that it is more likely in the second sample (human). In PQ2, we use a (non-parametric) Mann-Whitney test to compare the perceived quality (*i.e.,* the rates assigned by participants) of refactoring changes generated by machine with the rates of refactoring changes generated by humans. Non-parametric tests do not require any assumption on the underlying distributions. Other than testing the hypothesis, it is of practical interest to estimate the magnitude of the difference between the rates assigned to changes generated by machine and humans. Therefore, we compute the non-parametric effect size measure Cliff's δ (ES) [6], which indicates the magnitude of the effect of the treatment on the dependent variable. The effect size is considered negligible if < 0.147, small if between 0.147 and 0.33, medium if between 0.33 and 0.474, and large if > 0.474 [32].

3.12 Preliminary Results

3.12.1 PQ1: Can developers tell the difference between automated and manual refactorings?

In Table 5 we summarize the results of E_1. In the first column we use the following abbreviations: *all* for all the anti-pattern types studied, or the abbreviation of each type; columns 2 to 4 are the percentage of correct, wrong, or unknown answers; columns 5 to 7 are the corresponding percentages for machine changes (m), while columns 8 to 10 are the corresponding percentages for human changes (h).

When considering all anti-pattern types, we observe that the proportion of correct and wrong answers are the same (45%). In the remaining 10% of cases, respondents were not able to discern the origin of the changes, *i.e., I do not know* answer was selected, first row, columns 2-3. With respect to the anti-pattern's type fixed, Long-parameter and Spaghetti code have the highest percentage of machine changes correctly identified (more than 70%), and therefore we consider that they could not pass by manual changes. Conversely, Speculative generality, Blob and lazy class were correctly identified in less than 50% of cases, indicating that to some extent they mimic human behavior on refactoring tasks.

If we study the percentage of correct answers by refactoring origin, we observe that respondents found it more difficult to identify human patches (38%), whereas machine patches were little easier to identify (52%). This trend holds for all anti-patterns studied, except for Speculative Generality (SG), where the percentage of correctly identified machine changes (23%) is lower than the correctly identified human ones (55%). This

Table 5: E_1 RQ1 overall results

ap-type	correct	wrong	unknown	correct-m	wrong-m	unknown-m	correct-h	wrong-h	unknown-h
all	45.00%	45.00%	10.00%	52.00%	38.50%	9.50%	38.00%	45.50%	16.50%
bl	30.00%	61.25%	8.75%	37.50%	57.50%	5.00%	22.50%	65.00%	12.50%
lc	47.50%	45.00%	7.50%	47.50%	47.50%	5.00%	35.00%	42.50%	22.50%
lp	57.50%	33.75%	8.75%	80.00%	12.50%	7.50%	35.00%	55.00%	10.00%
sc	50.00%	45.00%	5.00%	72.50%	22.50%	5.00%	42.50%	37.50%	20.00%
sg	45.00%	45.00%	10.00%	22.50%	52.50%	25.00%	55.00%	27.50%	17.50%

Table 6: E_1 RQ1 results by respondent's expertise

Expertise (years)	Total	Correct	Correct %	Correct-h	Correct-m
>5	200	96	48.00%	40	56
>2 to 5	90	32	36.00%	12	20
1 to 2	70	36	51.00%	20	16
<1	40	16	40.00%	4	12

result is surprising since the refactoring type applied for removing the two different SG instances, which is Collapse Hierarchy (CH), was applied for both treatments. To find a plausible explanation for this result, we manually examined the cases where respondents failed to distinguish refactoring changes corresponding to SG anti-pattern type, and observed that they belong to the easy instance, which corresponds to 14 different respondents who failed to identify the origin of the changes generated by the automatic tool to remove SG anti-pattern type. In general, human judges doubted the ability of software tools to perform this refactoring type. For example, one responded commented: *"The abstract class is useless (the change is better). I don't think a tool can detect it."*.

To control for some confounding factors, like development experience, or confidence when answering the questionnaire, we cluster the results based on developer's experience (Table 6) and developer's confidence for each patch reviewed (Table 7).

We observed that the largest number of respondents declared to have more than five years of experience as Java developers, while the smallest group declared to have less than one year. Contrary to what one would expect, the group with more correct answers (51%) was the group with one to two years of experience, followed closely by the group with more than five years of experience (48%), indicating that the experience factor was not decisive to correctly identify the origin of the refactoring changes. The same situation occurs with the other groups, as the group with less than one year of experience correctly identified more refactoring changes than the group with more than two, up to five years of experience. We suggest that perhaps developers surveyed with less experience, are more skilled in Java and more versed in object-oriented design, and beside their little experience, they have worked on more challenging projects than their more experienced peers, or they simply have more experience on code refactoring.

Concerning the number of correctly identified refactoring changes by origin (columns 5-6, Table 6), we corroborate what we observed in the overall results, respondents had more difficulty to identify refactoring changes generated by human than by machine. With one remarkable exception: developers with one up to two years of experience, who correctly identified more refactoring changes applied by human than by machine.

With respect to confidence level declared per question, we observed that respondents felt confident enough, as in 60% of the questions they selected a confidence level between

Table 7: E_1 RQ1 results by respondent's confidence per question

Confidence	Total	Correct	Correct %	Correct-h	Correct-m
5	109	48	44.04%	24	24
4	142	69	48.59%	29	40
3	105	46	43.81%	18	28
2	30	12	40.00%	3	9
1	14	5	35.71%	2	3

4 and 5. We also observed that the largest proportion of correct answers (48.59%) corresponds to the confidence level of 4, followed by the ones identified with a confidence level of 5 (44.04%); the next group are respondents with a confidence level of 3 (43.81%); refactoring changes identified with a confidence level of 2, reached 40% of correctly identified patches; the last group, the one with the lowest level of confidence, achieved the smallest proportion of correct answers (35.71%), which makes sense.

If we analyze results based on the origin of the refactoring changes (columns 5-6), we observe that the machine's ones are more easily identified than the human ones, with one exception: those changes identified with a confidence level of 5 have the same proportion of correct answers. Contrary to the analysis based on respondents' expertise, the analysis based on respondents' level of confidence is more linear, as higher level of confidence led to higher number of correctly identified refactoring changes, in almost sequential order (the only exception being between levels 4 and 5, where respondents with confidence level of 4 achieved better results than those with confidence level of 5). Respondents that selected confidence level of 4 may have been more modest than those who selected 5. Still both intervals are on the top of the confidence scale.

In Table 8 we present the contingency table for the number of correctly/incorrectly identified refactoring changes with respect to their origin: machine (m), human (h) and the results of the Fisher's exact test and odds ratio OR between changes generated by human or machine. The contingency table shows the frequency distribution of the refactoring changes identified by the respondents with respect to the refactoring change's origin. Fisher's exact test indicates whether a significant difference of proportions between automatically- and manually-generated refactoring changes exists. Odds ratio indicates the probability of a respondent to correctly identify a change according to the origin of the refactoring change analyzed.

The first column of Table 8 corresponds to the anti-pattern's type; columns 3 to 4 correspond to the number of correctly/incorrectly identified changes by origin; and columns 5 to 6 reports the result of Fisher's exact test and ORs when testing $H_{0identify}$. In the first row, when aggregating all refactoring types together, the $p - value$ of the Fisher's exact test is statistically significant, with an OR greater than one, indicating that changes generated by machine were easier to identify than the ones generated by humans. Aggregating all anti-pattern types might not be very descriptive, specially considering that the refactoring types studied cover a wide range of design problems (coupling, lack of cohesion, design size, readability, etc).

To make a more precise analysis of the results, we expand our analysis to consider the refactorings of each anti-pattern type separately (rows 2 to 6, Table 8). We observe that only in two anti-pattern types, the results are statistically significant. Refactoring changes to remove long-parameter list have higher probability to be correctly identified by respondents when they are automatically generated (7 times more according to OR). Conversely, refactoring changes to correct Speculative Generality classes are more likely

Table 8: Contingency table and Fisher's exact test results for developers' survey on refactoring changes

ap_type	correct-m	wrong-m	correct-h	wrong-h	$p-value$	OR
all	104	96	76	124	<**0.01**	1.77
bl	15	25	9	31	0.22	2.05
lc	19	21	14	26	0.36	1.67
lp	32	8	14	26	<**0.01**	7.22
sc	29	11	17	23	0.01	3.51
sg	9	31	22	18	<**0.01**	0.24

to be correctly identified when they are refactored manually ($OR < 1$). Based on some respondent's comments, we suggest that the results obtained for Long-parameter list reflect respondent's view that the refactorings performed did not improve the quality of the systems. Because of this perception of poor quality, the respondents may have considered that such refactorings could have only been produced by an automatic tool. This results suggests that the human judges considered the introduce parameter-object applied by both, developers and machine, artificial. For example, Table 5 shows that the largest proportion of correct answers for an anti-pattern type correspond to long-parameter list with 80% of correct answers; conversely, 55% of respondents incorrectly chose machine origin when the origin was human; the remaining 10% of respondents declared themselves unable to decide (they did not know). For Speculative Generality anti-pattern, respondents tend to attribute the refactoring changes proposed for removing this type to humans, and some of them even mentioned that the refactoring proposed was too complex to be generated by an automatic tool. Note that existing popular Java IDEs like Eclipse and IntelliJ IDEA provide refactoring support for Long-parameter list (introduce-parameter object refactoring) but not for Speculative Generality (*e.g.,* Collapse hierarchy refactoring). Hence, they could have been misled by the wide availability of automatic refactoring tool support for refactoring long-parameter list anti-pattern when making their choices.

For the remaining anti-patterns types, the $p-values$ are > 0.01. Hence, we cannot reject $H_{0identify}$ for those types of anti-patterns. We therefore conclude that developers cannot differentiate between refactorings changes made by human and machine for those types of anti-patterns.

In general, automatically generated refactorings and refactoring changes made by humans were equally difficult to identify. In 10% of cases, developers couldn't even make a decision and opted for the "I do not know" option. Based on anti-pattern types, the results vary depending on the type. Refactorings generated for removing Long-parameter list and Speculative Generality anti-patterns failed the Turing test, whereas Blob, Lazy class and Spaghetti Code anti-patterns passed it.

3.12.2 PQ2: Is there any difference between the perceived quality of automated and manual refactoring?

In this research question, we examine developers' perception of automated refactorings. We want to know whether they have the same appreciation for automatically generated

Table 9: E_1, RQ2 Median rates by anti-patterns' type

Antipattern Type	rate_machine	rate_human	$p-value$	ES
all	3/5	3/5	0.02	negligible
bl	3/5	4/5	$<$**0.01**	medium
lc	3/5	4/5	0.13	small
lp	3/5	3/5	0.65	negligible
sc	3/5	3.5/5	0.10	small
sg	4/5	3/5	0.38	negligible

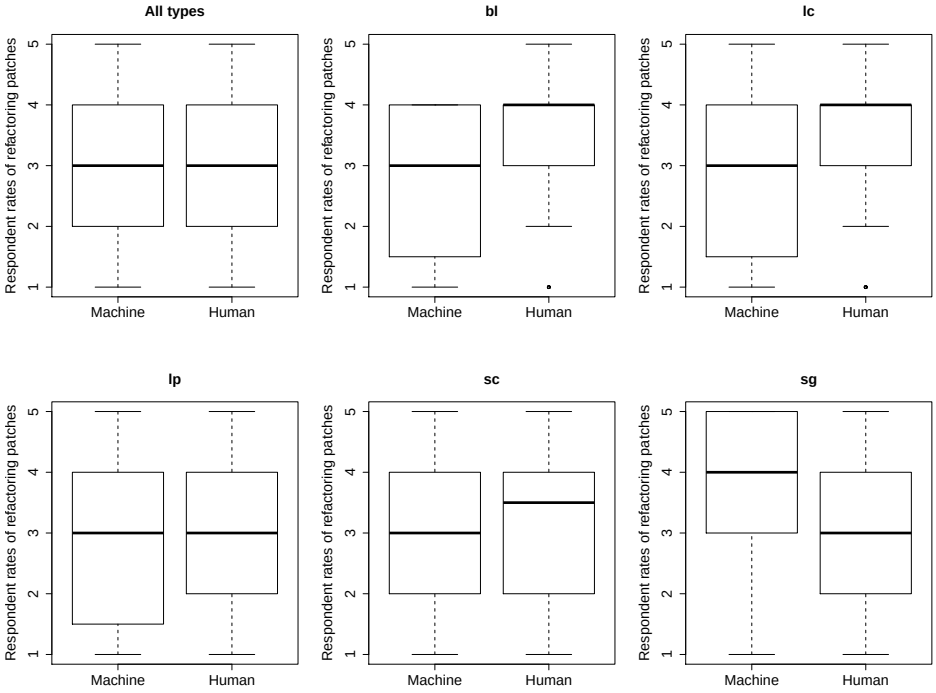


Fig. 1: User's rates distribution.

refactorings and manual refactorings. The absence of statistically significant difference between the two treatments could serve as evidence that automated refactoring can be interleaved with manual changes without affecting the quality of a system.

In Table 9 we present the respondent's median rates for the refactoring changes presented in E_1, and in Figure 1 the boxplots of user's rates distribution.

Respondents input their rate for each refactoring change using a Likert-scale from 1 to 5. When considering all anti-patterns types, both sources of change (human, machine) attained the same rate. With respect to anti-pattern's type, the median rates for human changes are higher (0.5 to 1 point) than the median rates of machine changes (Blob, Lazy class and Spaghetti code types). Conversely, the machine refactoring changes to remove Speculative Generality are rated higher than the changes generated by humans to remove the same anti-pattern type. Automatic and manual refactoring changes

for removing Long-parameter list anti-patterns achieved the same rate. We apply the Mann-Whitney test to determine if the results are statistically significant, and we find that only refactoring changes that remove Blob instances achieve statistical significance, with a $p - value$ less than 0.01, and a medium Cliff's δ ES. Hence, we reject $H_0 rank$ for all anti-patterns types except the Blob.

We conclude that for the anti-patterns types studied, developers do not find manual refactoring changes to be of better quality than automated refactoring changes. The exception being the Blob type.

3.13 Recommendations to improve automated refactoring operations based on participants feedback

In the next paragraphs we outline some recommendations obtained from the participants of the preliminary study with respect to the quality of the automated refactorings. This information is summarized from the comments left to refactorings performed by RePOR in the online survey.

In general, developers highlight the importance of adopting naming and code conventions in refactoring tools. For example, the use of uppercase for class names, correct indentation, camel case, etc.

In some cases, the refactoring changed the visibility of some attributes to 'public', which was severely criticized by developers, despite the fact that the refactoring is inline with refactoring guidelines. In other situations, developers feel that the refactoring was incomplete, and suggested that human developers would go further. For example, save the objects used frequently in a local variable, or merging `Catch` statements using logic operators.

In the case of the Blob type, RePOR refactored the code using indirection, and `@deprecated` tags, which was not well seen by developers.

On the other hand, some developers confessed that they would live with the problem of anti-patterns like lazy class, due to the extra effort required to fix it. They also admitted that having a tool to remove that burden from their shoulders is positive.

4 Experimental Design

We perform an experiment to assess the comprehension of source code by developers in the presence of five types of object-oriented anti-patterns: Blob (BL), Lazy Class (LC), Long Parameter List (LP), Spaghetti Code (SC) and Speculative Generality (SG). We chose these anti-patterns because they are representative of poor design choices . In fact, Palomba et al. [30] found in a case study with developers from both industry and academia, that Blob class and Spaghetti Code are highly recognized and considered high severity design problems; Speculative Generality was perceived as problem of medium severity; Long-parameter list and Lazy class were considered low severity problem. The rational to study Long-parameter list is that this anti-pattern type is deemed to affect code readability, and Lazy class to bloat code design unnecessarily. Additionally, previous works have proposed approaches to detect the anti-patterns studied in this work [19, 16].

Using a different pool of participants from the preliminary study, we present them with code refactored by either developers or an automated tool, and ask them to complete some comprehension tasks on the refactored code entities. The aim is to assess the comprehensibility of code after refactoring. In each experiment, we selected two instances of each anti-pattern type studied: one *easy* and one *difficult*. The level of difficulty is decided based on several object-oriented metrics, based on each anti-pattern type.

4.1 Research Questions

Our research questions stem from our goal of understanding the impact of automated refactoring on developer's comprehension. We state them as follows.
RQ1: Do automated refactorings affect code understandability?

4.2 Hypotheses

RQ1 is related to subjects performing code comprehension tasks on refactored code. We study the performance of the subjects along three dimensions: time to execute the comprehension task, effort measured using NASA task load index (TLX) [34], and percentage of correct answers. We test $H_{0performance:}$ There is no difference between the performance of developers performing comprehension tasks on code refactored by developers compared to automatically refactored code.

4.3 Objects

The objects of this study are the same used in the Preliminary study Section 3.3

4.4 Subjects

For E_2, we hired 30 more additional Java developers from aforementioned two marketplace websites. To select a candidate, we perform an informal interview with the candidates, and we set a minimum of experience of one year developing Java software for the industry. To control for possible confounding factors, we registered the number of years of experience of each participant as professional developers, and asked them to provide their Java's level of confidence (using a Likert scale from 1 to 5).

4.5 Independent variable

The independent variable for E_2 is the same used in E_2. It is a binary variable stating whether the refactoring was performed by an automatic tool or by a developer.

4.6 Dependent Variables

In E_2, the dependent variables measure the subjects' performance, in terms of effort, time spent, and percentage of correct answers. We measure a subject's performance using the NASA Task Load Index (TLX). TLX evaluates the subjective workload of subjects. It is a multidimensional measure that provides an overall workload index based on a weighted average of ratings on six sub-scales: mental demands, physical demands, temporal demands, own performance, effort and frustration. We combine weights and ratings provided by the subjects into an overall weighted workload index by multiplying ratings and weights; the sum of the weighted ratings divided by fifteen (sum of the weights) represents the effort [34]. To measure the time that participants spent on each task, we recorded the participants' remote session on the virtual machine where participants performed their assignment. The time reported only considers the time spent on the comprehension task, from the moment they open the project in the IDE, until they close it. We compute the percentage of correct answers for each question by dividing the number of correct elements found by a participant by the total number of correct elements (s)he has found. For example, if the question requires to find the total number of code references for a given object, and there are ten references but the subject finds only four, the percentage of correct answers is forty for that question.

4.7 Questions

For E_2, we used comprehension questions to elicit comprehension tasks and collect data on the subject's performance. As in [1], we consider questions in three of the four categories of questions regularly asked and answered by developers [36]: (1) finding a focus point in some subset of the classes and interfaces of some source code, relevant to a comprehension task; (2) focusing on a particular class believed to be related to some task and on directly-related classes; (3) understanding a number of classes and their relations in some subset of the source code; and, (4) understanding the relations between different subsets of the source code. Each category contains several questions of the same type.

We only selected questions in the first three categories, since the last category concerns different subsets of the source code, while in our experiments, we focus exclusively on one or two packages at most, that are affected by a particular anti-pattern. For each category, we choose the two most relevant questions through discussions between the first author and a Master's student intern who collaborated on this work. The decisions were validated by the second author. Selecting two questions for each category of question, which provided us with two data points from each participant. The six selected questions are the followings. The text in bold is a placeholder that we replace with the appropriate behaviors, concepts, elements, methods and types depending on the system on which the subjects performed their tasks.

- Category 1: Finding focus points:
  - Question 1: Where is the code involved in the implementation of **this behavior**?
  - Question 2: Which type represents **this domain concept** or **this UI element or action**?
- Category 2: Expanding focus points:
  - Question 1: Where is **this method** called or **this type** referenced?
  - Question 2: What data can we access from **this object**?

- Category 3: Understanding a subset of classes:
  - Question 1: How are **these types or objects** related?
  - Question 2: What is the behavior that **these types** provide together and how is it distributed over **these types**

For example, with system *83 Xbus* (*cf.,* Table 2), we replace "**this behavior**" in question 1, category 1, by "manipulating and store the email received by class `POP3XMLReceiver`". For category 2, we acknowledge that the questions might be answered by developers using the IDE search functionality. However, developers still must identify and understand the classes or methods that they consider relevant to the task. Additionally, discovering classes and relationships that capture incoming connections prepare the developers for the questions of the third category. Below we present the comprehension task for system *83 Xbus*.

Answer the following comprehension questions related to the system Xbus. We focus on the package **net.sf.xbus.technical.mail**.

1. Where is the code involved in manipulating and storing the email received by class `POP3XMLReceiver`?
2. Which type (class) defines a method for reading an email after registering its receiver in the Transaction manager?
3. Where is the type `POP3XMLReceiver` referenced?
4. What data can we access from the object `mEmailMessage`?
5. How are `Email` and `POP3XMLReceiver` related?
6. What is the behavior that `POP3XMLReceiver` and `Email` provide together and how is it distributed over these types?

4.8 Design

For E_2, we use the same collection of refactoring examples from E_1, 5 anti-pattern types, and two instances for each type; each instance with two possibilities: being generated by human or by machine. That totals 10 refactoring changes for each origin. We hired 30 freelancers, who completed two tasks each of them, leading to 60 comprehension tasks.

4.9 Procedure

For E_2, the subjects knew that they would perform comprehension tasks, but did not know the goal of the experiment nor whether the system was refactored by a developer or by an automated approach. We informed them of the goal of the study after they finished the experiment.

4.10 Analysis method

For RQ1 we use Mann-Whitney test to compare two sets of dependent variables and assesses whether their difference is statistically significant. The two sets are the subject's data collected when they answer the comprehension questions on the systems refactored by either machine or humans. For example, we compute the Mann-Whitney tests to compare the set of times measured for each subject on the systems refactored by either machine or humans. We also compute the Cliff's δ (ES).

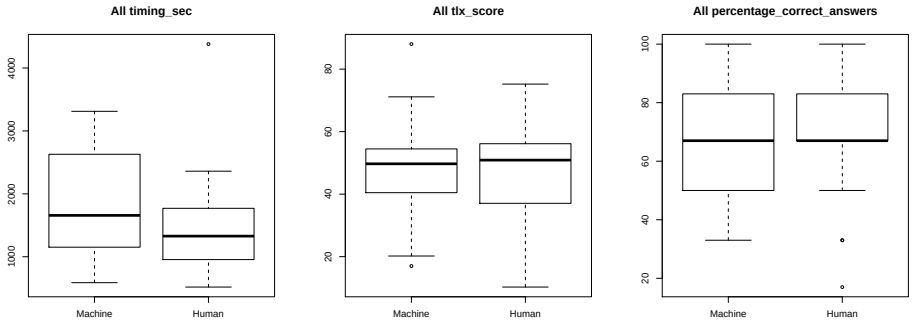


Fig. 2: Comprehension tasks' performance results distribution.

5 Study Results

We now describe the collected data and present the results of our study, answering the RQ formulated in Section 4.

5.1 RQ1: Do automated refactorings affect code understandability?

In this research question, we are interested to know the impact on understandability of refactoring changes generated by human and machine. We now present the results of E_2. Table 10 summarizes the median of collected data, and Figure 2 presents the overall performance of participants on the comprehension tasks, in terms of: time, effort, and percentage of correct answers. In addition, we divide the refactoring changes in two groups (see rows 4 to 9 of Table 10), i.e., those who deceived human judges in E_1 (T. passed), and those who failed the test (T. failed). In columns 5, and 6, we report the $p - value$ of the Mann-Whitney tests and the Cliff's δ ES obtained for each studied dimensions.

We observe that when considering all refactoring changes, the median time for completing the comprehension task on code automatically refactored is slightly higher than the median time spent on code manually refactored (approximately 5 minutes more). However, the difference is not statistically significant and the effect size is small.

In the case of effort, the effort perceived by developers is almost the same, with a negligible effect size.

In the case of percentage of correct answers, the median values are the same for both human and machine.

We repeated the analysis along the same three dimensions while dividing the refactoring changes in two groups (passed and failed the Turing tests). With respect to time, the difference of the medians for anti-patterns that passed the Turing test (*i.e.,* BL, SC, LC) is higher (medium ES) for automatically generated changes, though the difference is not statistically significant. In the case of effort, the effect size is negligible and not statistically significant, same for percentage of correct answers.

For anti-patterns that failed the Turing test, the results are not statistically significant either.

Since we could not find statistically significant differences in the performance of developers when performing code comprehension tasks on systems that were manually

Table 10: RQ3 Comprehension task results E_2: median values for the three dimensions studied

Grouping	Dimension	Human	Machine	$p-value$	ES
ALL	Time (seconds)	1 328.00	1 657.50	0.07	small
ALL	Effort (TLX index)	50.90	49.71	0.82	negligible
ALL	% of correct answers	67.00	67.00	0.76	negligible
T. passed	Time (seconds)	1 100.00	1 606.00	0.09	medium
T. passed	Effort (TLX index)	51.37	49.71	0.72	negligible
T. passed	% of correct answers	67.00	75.00	0.83	negligible
T. failed	Times (seconds)	1 176.00	1 606.00	0.16	small
T. failed	Effort (TLX index)	51.37	48.00	0.36	small
T. failed	% of correct answers	83.00	67.00	0.01	large

and automatically refactored, we reject $H_{0performance}$. Hence, we conclude that for the anti-patterns studied in this work there is no difference between the performance of developers performing comprehension tasks on code refactored by developers compared to automatically refactored code. This is good news for toolsmiths working on automatic tools for removing anti-patterns. For developers, this result will likely increase their trust in automatic refactoring tools, and for researchers it extends the existing body of knowledge on the benefits of automatic refactoring.

We conclude that for the set of refactoring strategies studied, developers can safely use automated tools since the impact of the automated changes on the comprehension of the code is not significantly different from the impact of changes performed by human developers.

6 Discussion

In this section, we provide further information about the obtained results and discuss their implications. The results in E_1 show that in general, automatically generated refactorings and refactoring changes made by humans were equally difficult to identify. In 10% of cases, developers couldn't even make a decision and opted for the *"I do not know"* option.

6.1 Refactoring change differences by anti-pattern type

When analysing results based on anti-patterns types, the picture is a bit different.

6.1.1 Long-parameter-list anti-patterns automatically refactored were easier to identify than their manual counterpart

For the two instances of Long-Parameter list studied, the refactoring strategy selected by both human and machine was to introduce a parameter-object. In fact, the solution provided for the *easy* instance is conceptually the same for both the automatic tool and the developer (the only difference is the name selected for the new class). However, since we replaced the name in both solutions by an artificial name (*e.g.,* Clazz09465) to avoid introducing bias in the experiment (as explained in Section 3.8), the two refactoring

changes are semantically equivalent. In the *hard* instance, there was a small difference between the two solutions; the freelancer declared the new class in the same file, while our automated approach created a new file; besides that minor change, we can consider both solutions to be semantically equivalent too.

As we mentioned in Section 5, we suggest that the quality of the refactoring changes applied to correct Long-parameter list did not influence the choice of the participants in the Turing test, but their expectations about refactoring. Some respondents mentioned that there is no reason to extract a new parameter object, given that the parameter object only stores data. Hence, we suggest that the developers surveyed did not give importance to the understandability aspect of having a long-list of parameters over having an extra class that clusters the common parameters together, and that is why they attributed the change to a machine. Note that the number of parameters extracted for the Long-parameter list instance identified by the authors as *hard* is considerably long (8 parameters) compared to the rest of the classes (4 parameters) in the system where the instances of Long-parameter list anti-pattern resides.

6.1.2 Refactoring changes of speculative generality and lazy class are semantically equivalent

With respect to Speculative Generality and Lazy class, the refactoring strategies employed by both the automated tool and the developers are the same for the first anti-pattern's type (*i.e., Collapse hierarchy*), while for the second type (*i.e.,* Lazy class) the refactoring applied (*i.e., inline class*) exhibits a small variation on the selected target class. The Lazy classes instances are composed of a unique static method, so they could be placed in any class that makes use of this method, so we consider the refactoring solutions for these two anti-pattern types to be equivalent, despite their origin. Given that in the solutions for both anti-pattern's types no classes or methods are added to the system, these refactorings can be completely automated, as they do not require to provide names to entities, according to some code conventions set for the systems.

6.1.3 The hard instance of Spaghetti code that was manually refactored was easier to identify than the automatically refactored one

The machine and human refactoring changes applied to remove spaghetti code, for the easy and hard instances, used the same strategy: *replace long method with method object*. This strategy consists of extracting a long method inside the spaghetti code class into a new class. For the easy instance, both solutions differ just in the name selected for the new class, and since we normalized the names, we consider them semantically equivalent. On the other hand, the refactorings applied for the hard instance of spaghetti code class exhibit some design differences that may have help some survey respondents to identify their origin.

The long method extracted from the spaghetti code class makes use of a private attribute and a static method inside the spaghetti code class, and it does not require access to other external attributes or methods. In the automated refactoring change, two private attributes where moved to the new extracted class, while in the manual solution the freelancer added public getter to access the required private attribute outside the spaghetti code class. Hence, the use of the private attributes, in the automatically refactored code, are delegated to the extracted class inside the spaghetti code class. Since

both machine and human set the access modifier of the static method to public, there was no need to pass a reference to the spaghetti code class in the automated refactoring solution. But in the manual solution the extracted long method requires a reference to the spaghetti code class to use the required private attribute. These differences were noticed by one of the survey respondent, who provided the following explanation to justify why he thought that the refactoring was automatically generated: *"I don't think that a developer would extract the long method from the spaghetti code class to the new extracted class, and add as a parameter to the long method a reference to the spaghetti code class, because that reference is not used"*. With respect to the manual solution, one respondent commented that the addition of a getter, to the private attribute, and the way the extracted class was instantiated to call the long method does not seem to be automatically generated. To provide a complete picture of what we discussed, we show in Listing 2 and Listing 3, the actual changes performed to instantiate the new class, and execute the extracted long method for both automatic and manual solutions. We use the acronyms ADD, DEL, and CHG to indicate added, deleted, and changed line.

Listing 2: Extract from refactoring change for *hard* instance of spaghetti code by machine.

```
public class WorkflowManager {
DEL  private String logconfig =
       System.getenv ("HS_HOME") +
       File.separator + "config" +
       File.separator +
       "log4j.properties";

ADD private Clazz008353
       Clazz008353 = new Clazz008353
       (System.getenv("HS_HOME") +
       File.separator + "config" +
       File.separator +
       "log4j.properties");

   public WorkflowManager() {
DEL  PropertyConfigurator.configure
       (logconfig);
ADD PropertyConfigurator.configure
       (Clazz008353.logconfig);
   }
   private void
          process_submission(String[]
          args) {
CHG Clazz008353.process_submission
       (this, args);
   }
 }
```

Listing 3: Extract from refactoring change for *hard* instance of spaghetti code by human.

```
public class WorkflowManager {
      private void
             process_submission(String[]
             args) {
CHG new Clazz008353
       (this).submit(args);
      }
 }
```

The differences between the automatic solution for the *hard* instance of Spaghetti code with respect to the manual one could be easily matched by adding a validation to prevent passing a reference of the refactored spaghetti class as parameter to the long method call, when none of its attributes and methods are used. Toolsmiths should pay attention to corner cases like this, which can only be revealed by thoroughly testing their tools with several scenarios and systems.

6.2 Perceived quality of refactoring changes

Concerning quality perceived by developers, we observed that it is only for the Blob anti-pattern type (which passed our Turing test) that the difference of rates achieved between manual and automatic refactoring changes is statistically significant, favoring manual refactoring.

We identify the following differences in the refactoring strategies used by human and the automatic approach that are worth to discuss.

First, the automated approach selected *move method* as a mean to decompose Blob classes and to redistribute functionality to other classes in the system, while the human approach relied on extracting new classes to reduce the size of Blob classes.

With respect to the refactoring strategy applied for the *easy* instance of Blob, the automated solution consists of delegating 12 methods to a class where an association relationship exists. One respondent complained that the coupling between these two classes increased dramatically, which was not the case. In contrast, the manual solution for the *hard* Blob's instance took a different path. It consists of extracting two attributes and their corresponding getters and setters to a new class. This solution, though semantically correct, does not reduce the size of the Blob class significantly, but introduce coupling to a new data class.

For the *hard* instance, one respondent commented about the manual solution as follows: *"it seems good and needs to be corrected with minor changes"*. We studied the manual solution and found that it performs a clear separation of concerns by extracting methods and attributes to new classes according to their names. That would be hard for a machine to achieve, unless they gather some knowledge about the code lexicon. The refactoring strategy followed by the automated approach consists of delegating three methods to a related class that is member attribute of the Blob class. Although, the strategy taken by the tool prevents adding extra coupling to a new extracted class while reducing Blob's class size, respondents of E_1, did not like it. They highlighted that when moving methods, the automated approach changed the visibility of the required attributes to public, which implied several changes with no clear benefit. Respondents rated the automatically-generated refactorings for Blob type with average scores of 2.6 and 2.7, for the easy and hard instances respectively. While the human ones obtained 4.4 and 2.5 respectively. It is interesting to note that although the automatically-generated refactorings of Blob did not achieved the best rate, the perceived quality showed little variation between the two instances despite their different levels of complexity of each system. On the other hand, the rate achieved by the two manual refactoring changes for Blob type varied considerably based on the human effort of each freelancer, and the complexity of the code that enable (or not) to abstract functionality to new classes. From a practical point of view, developers will not only reduce maintenance costs by using automated approaches, but can ensure a standardized quality gain after the refactoring process.

6.3 Improving automated refactoring of Blob classes

To improve the performance of automated refactoring on Blob anti-patterns, it would be necessary to control for code semantics targeting two aspects: (1) the generation of refactoring candidates should be guided by code lexicon; (2) an automated refactoring approach should incorporate a mechanism for naming new code entities based on code

Table 11: $p-values$ of the impact of the mitigating variables on the performance of participants for E_2

Variable	origin	effort: $p-values$	time: $p-values$	% of correct answers: $p-values$
Expertise	human	0.3658	0.9516	0.857
	machine	0.2172	0.6145	0.9109
Java confidence	human	0.4522	0.01657	0.346
	machine	0.8893	0.8166	0.1339

lexicon. The first aspect has been already discussed by Ouni et al. [29], while the second point remains unexplored. We suggest that by addressing these two aspects we could improve the performance of automated refactoring of Blob classes, making it as good as the human refactorings, or even better.

6.4 Refactoring changes impact on code comprehension

Concerning the impact on code comprehension, we could not make an analysis by anti-pattern type as we only have 6 points for each anti-pattern type. However, E_2 reveals no significant difference between subject's efforts, times, and percentages of correct answers on systems refactored by human and machine. We investigated whether the two mitigating variables : expertise and Java's level of confidence (not to be confused with respondent's level of confidence per question in E_1) declared by participants impacted our results. We set 4 levels for the years of experience and 5 levels for the degree of Java's confidence, using a Likert scale, from *no confidence* to *highly confident*.
Table 11 presents some descriptive statistics of the data collected for these two mitigating variables. Since for each mitigating variable we have multiple levels (more than two) corresponding to multiple categories, we used the Kruskal-Wallis Test, which is a non-parametric test for comparing multiple medians, to assess the impact of the mitigating variables on the three performance dimensions (time, effort, and % of correct answers). We observed that the mitigating variables do not impact our results, as shown by the high $p-values$ in Table 11; *i.e.,* participants mostly had the same performance on refactorings from the same origin, no matter their level of expertise/Java confidence, which reinforces our assessment that refactoring changes are what did matter.

7 Threats to Validity

There are threats that limit the validity of this study. We discuss these threats and how we alleviate or accept them following common guidelines provided in [40].

7.1 Construct Validity

Construct validity threats concern the relation between theory and observations. This work relies on good developers practices (*i.e.,* extreme programming [3]) where developers are advised to perform refactoring to remove anti-patterns, in order to maintain the design quality at an acceptable levels, and hence ease software evolution. However, we cannot claim that removing anti-patterns is the prime reason for developers to perform

refactoring, specifically the ones that we studied. However, relying on the notion of refactoring anti-patterns allowed us to objectively evaluate two methods (manual and automated refactoring), and to control from a large space of possibilities. That is why we had to constraint our study of the refactoring practice to existing well-known refactorings [8, 5].

In E_2, we use time and percentage of correct answers to measure the subjects' performance. The measured time was extracted from the video recording of the comprehension tasks sessions, while the percentage of correct answers was evaluated by one author of the paper and one Master's student. We believe that these measurements are objective, even if they can be affected by external factors, such as fatigue. We also use NASA TLX score to measure the subjects' effort. The TLX is by its own nature subjective and, thus, it is possible that our subjects declared effort values that do not perfectly reflect their effort.

The degree of severity of the anti-patterns is also a threat to construct validity. The anti-patterns instances selected in each system were validated through a voting process for decisions. The first author and a Master's student voted for the anti-patterns, and the second author reviewed the decisions. We based our selections on the definitions and examples provided by Brown and Fowler [5, 8]. To validate the solutions proposed by the automatic tool and the developers (*i.e.,* the freelancers), we check that the solution preserves code's behavior based on the unit tests included in the SF110 corpus. We also control for the level of complexity of the refactoring changes proposed by freelancers and our tool by computing normalized change entropy metric, which showed that the changes judged by human evaluators are fair for both treatments *cf.,*2. Yet, it is possible that some of the refactoring changes proposed would have a different effect if applied to other systems in different contexts.

Construct validity threats could be the result of a mistaken relation between (automated) refactoring and program comprehension. We believe that this threat is mitigated by the fact that this relation seems rational. The results of our analysis suggest that certain anti-patterns' type can be automatically refactored with the same level of quality as the refactorings performed by human developers.

7.2 Internal Validity

We identify 4 threats to the internal validity of our study: learning, selection, instrumentation, and diffusion.

7.2.1 Learning

Learning threats do not affect our study for the two experiments because we used a between-subject design. A between-subject design uses different groups of subjects, to whom different treatments are assigned. Additionally, we took each anti-pattern instance from different systems. For E_1, we balanced the groups (alternating difficulty level, and origin); then we randomized the appearance order of the refactoring changes for each group. For E_2, the freelancers had to perform two comprehension tasks. To mitigate the learning effect, the systems were presented in the same order for both treatments (manual and automatic refactoring). For example, consider a comprehension task for systems 47, and 52. We anticipated that developers will spent more time for the first system (*i.e.,* 47), while getting familiar with the instructions, developer environment,

etc., than with the second system (52). Hence, the extra time spent as a consequence of the learning process is considered in the same system for both treatments.

7.2.2 Participant's selection

Participant's selection threats could impact our study due to the natural difference among the participants' skills. In E_1, we tried to mitigate this threat by inviting developers through technical Java developers mailing list, (*e.g.,* openJDK project), developers groups on social networks (*e.g.,* LinkedIn, Reddit, and Facebook). In E_2, we studied the possible impact of their expertise in Java through two mitigating variables and found no significant impact on the obtained results.

7.2.3 Instrumentation

Instrumentation threats were minimized by using objectives measures like time and percentage of correct answers. We observed some subjectivity in measuring refactoring changes quality, developer's confidence, and developer's experience in E_1; and developers' effort measured using the TLX score. For example, 5 years of experience of one developer, could be the equivalent of 3 years for another one. However, this subjectivity is inevitable in self-evaluations.

Another instrumentation threat to our study is the anonymization of new code lexicon. This only affects Blob, Long-parameter list and Introduce-parameter object anti-pattern types. Automated refactorings that introduce new elements are likely to be distinguished from their manual equivalents. But naming code lexicon is just one part of the semantic context. By anonymizing the names of newly created entities, we wanted to steer the focus of the respondents toward the structure of the code changes. Recent works [12] in automated code comment generation have shown promising results on generating human-like comments using deep learning. Hence, we believe that there is a reasonable possibility to overcome the current code lexicon limitations of automated refactoring tools in the future, and this study can serve as base for performing further studies when the technology is mature enough.

7.2.4 Diffusion

Diffusion threats do not impact our study because (1) we recruit participants through web platforms and mailing lists, and they do not have physical interactions, and (2) we asked them to not disclose any information about the content of the surveys and the systems.

7.3 Conclusion Validity

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate the assumptions of the statistical tests that were performed. Indeed, we used non-parametric tests that do not require to make assumptions about the distribution of the data set.

7.4 Reliability Validity

Reliability validity threats concern the possibility of replicating this study. We provide all the necessary details to replicate our study in our lab's Web page [21], including a sample of the questionnaire and the comprehension tasks, and raw data to compute the statistics. The systems analyzed from SF110 are open-source and can be downloaded from the author's web site. Our automated refactoring tool is also available on-line at `https://github.com/moar82/RefGen`.

7.5 External Validity

We performed our study on 10 different real-world systems belonging to different domains and with different sizes (see Table 2). Our experimental design, providing a few classes of each system to each participant, is reasonable because, in real maintenance projects, developers perform their tasks on small parts of whole systems and probably limit themselves as much as possible to avoid getting lost in a large code base. In E_1 we summarize the refactoring changes using the diff notation, and provide it along with the original source code; allowing developers to spot changes fast, in a readable and standard way, and in case of doubt, to clone the repository to explore the code and–or apply the changes (by applying the diff file as a patch). To mitigate the impact that a lack of familiarity to diff notations could have on the responses of our study, we explained the notation to participants prior to participating in the study and provided them multiple examples in a guidelines document. We cannot claim that our results can be generalized to other refactoring tools, and to other subjects. To generalize our results, we should implement other approaches different from RePOR. However, this was not the main objective of this work. Rather, we want to empirically evaluate if automatically refactored code can be interleaved with human code on development and maintenance activities of real world systems.

Our future work includes replicating this study in other contexts, with other subjects, tools, questions, anti-patterns, and software systems.

8 Conclusions

Refactoring tool support is conjectured in the literature to be underused due to lack of awareness, and developers reluctance to incorporate machine-generated-code into their code base. To debunk this myth, and foster awareness of automated refactoring, we performed two experiments to evaluate the perceived quality of automated refactorings and their impact on code comprehension. Our results show that developers could not distinguished between automatically generated refactorings and refactorings created by humans, for 3 out of the 5 anti-pattern types studied. Moreover, in general, developers did not prefer refactorings generated by humans over automatically generated refactorings, the only exception being for removing Blob classes. We found no significant difference between the performance of developers performing comprehension tasks on code refactored by developers or by an automatic tool, to remove Blob, Lazy class, and Speculative generality. Hence, we conclude that automated refactoring can be as effective as manual refactoring. However, for complex anti-patterns' types like Blob, we suggest that developer's expertise be included in the refactoring process as much as possible.

In the future work, we plan to enhance automated refactoring with code semantics by leveraging the code lexicon of systems when determining the best candidates to receive functionalities extracted from Blob classes, and for automatically naming the new classes, and–or methods introduced during a refactoring process. By doing this, we could generate more natural refactoring solutions, and close the gap between human and machine generated refactorings.

References

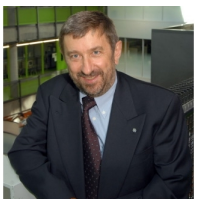
1. Abbes, M., Khomh, F., Gueheneuc, Y.G., Antoniol, G.: An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. on, pp. 181–190 (2011)
2. Arima, R., Higo, Y., Kusomoto, S.: Toward refactoring evaluation with code naturalness. In: Proceedings of the 26th International Conference on Program Comprehension, ICPC '18. IEEE Press (2018). DOI https://doi.org/10.1145/3196321.3196362
3. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change (2Nd Edition). Addison-Wesley Professional (2004)
4. Bois, B.D., Demeyer, S., Verelst, J., Mens, T., Temmerman, M.: Does god class decomposition affect comprehensibility? In: IASTED Conf. on Software Engineering (2006)
5. Brown, W.J., Malveau, R.C., Brown, W.H., McCormick III, H.W., Mowbray, T.J.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, first edn. John Wiley and Sons (1998)
6. Cliff, N.: Ordinal methods for behavioral data analysis. Psychology Press (2014)
7. Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M., Shepperd, M.: A controlled experiment investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software **72**(2), 129 – 143 (2004). DOI https://doi.org/10.1016/S0164-1212(03)00240-1
8. Fowler, M.: Refactoring: improving the design of existing code. Pearson Education India (1999)
9. Fraser, G., Arcuri, A.: A large scale evaluation of automated unit test generation using evosuite. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(2), 8 (2014)
10. Griswold, W.G., Notkin, D.: Automated assistance for program restructuring. ACM Transactions on Software Engineering and Methodology (TOSEM) **2**(3), 228–269 (1993)
11. Hassan, A.E.: Predicting faults using the complexity of code changes. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 78–88 (2009). DOI 10.1109/ICSE.2009.5070510
12. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, pp. 200–210. ACM (2018)
13. Kataoka, Y., Imai, T., Andou, H., Fukaya, T.: A quantitative evaluation of maintainability enhancement by refactoring. In: Software Maintenance, 2002. Proceedings. International Conference on, pp. 576–585. IEEE (2002)

14. Kim, M., Zimmermann, T., Nagappan, N.: An empirical study of refactoringchallenges and benefits at microsoft. IEEE Transactions on Software Engineering **40**(7), 633–649 (2014). DOI 10.1109/TSE.2014.2318734
15. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Software Engineering (ICSE), 2012 34th International Conference on, pp. 3–13. IEEE (2012)
16. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: IEEE Int'l Conference on Software Maintenance, ICSM, pp. 350–359. IEEE Computer Society (2004)
17. Mens, T., Tourwé, T.: A survey of software refactoring. Software Engineering, IEEE Transactions on **30**(2), 126–139 (2004)
18. Moghadam, I.H., Cinneide, M.O.: Automated refactoring using design differencing. In: Software Maintenance and Reengineering (CSMR), 16th European Conference on, Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, pp. 43 – 52. IEEE Computer Society (2012). DOI 10.1109/CSMR.2012.15. URL `http://dx.doi.org/10.1109/CSMR.2012.15`
19. Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.: Decor: A method for the specification and detection of code and design smells. Software Engineering, IEEE Transactions on **36**(1), 20–36 (2010)
20. Morales, R., Chicano, F., Khomh, F., Antoniol, G.: Efficient refactoring scheduling based on partial order reduction. Journal of Systems and Software (2018). DOI https://doi.org/10.1016/j.jss.2018.07.076
21. Morales, R., Khomh, F., Antoniol, G.: Repor: Mimiking humans on refactoring tasks. replication website. `https://moar82.github.io/emserefturing/` (2019)
22. Morales, R., Sabane, A., Musavi, P., Khomh, F., Chicano, F., Antoniol, G.: Finding the best compromise between design quality and testing effort during refactoring. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 24–35 (2016)
23. Morales, R., Soh, Z., Khomh, F., Antoniol, G., Chicano, F.: On the use of developers' context for automatic refactoring of software anti-patterns. Journal of Systems and Software **128**, 236 – 251 (2017)
24. Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., Succi, G.: A case study on the impact of refactoring on quality and productivity in an agile team. In: Balancing Agility and Formalism in Software Engineering, pp. 252–266. Springer (2008)
25. Murphy-Hill E.; Black, A.: Refactoring tools: Fitness for purpose. Software, IEEE **25**(5), 38–44 (2008)
26. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. Software Engineering, IEEE Transactions on **38**(1), 5–18 (2012). DOI 10.1109/TSE.2011.41
27. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A comparative study of manual and automated refactorings. In: G. Castagna (ed.) ECOOP 2013 – Object-Oriented Programming, pp. 552–576. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
28. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
29. Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Hamdi, M.S.: Improving multi-objective code-smells correction using development history. Journal of Systems and Software **105**(0), 18 – 39 (2015)

30. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A.D.: Do they really smell bad? a study on developers' perception of bad code smells. In: Software Maintenance and Evolution (ICSME), 2014 IEEE Int'l Conference on, pp. 101–110. IEEE (2014)
31. Parnas, D.L.: Software aging. In: ICSE '94: Proc. of the 16th Int'l conference on Software engineering, pp. 279–287. IEEE Computer Society Press (1994)
32. Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J., Devine, L.: Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. In: annual meeting of the Southern Association for Institutional Research (2006)
33. Seng, O., Stammel, J., Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems. GECCO 2006: Genetic and Evolutionary Computation Conference, Vol 1 and 2 pp. 1909–1916 (2006)
34. Sharek, D.: A useable, online nasa-tlx tool. In: Proceedings of the Human Factors and Ergonomics Society Annual Meeting, vol. 55, pp. 1375–1379. SAGE Publications Sage CA: Los Angeles, CA (2011)
35. Sheskin, D.J.: Handbook of parametric and nonparametric statistical procedures. crc Press (2003)
36. Sillito, J., Murphy, G.C., De Volder, K.: Asking and answering questions during a programming change task. IEEE Transactions on Software Engineering **34**(4), 434–451 (2008)
37. Szőke, G., Nagy, C., Hegedűs, P., Ferenc, R., Gyimőthy, T.: Do automatic refactorings improve maintainability? an industrial case study. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 429–438 (2015). DOI 10.1109/ICSM.2015.7332494
38. Turing, A.M.: Computing machinery and intelligence. In: Parsing the Turing Test, pp. 23–65. Springer (2009)
39. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, disuse, and misuse of automated refactorings. In: Software Engineering (ICSE), 2012 34th International Conference on, pp. 233–243. IEEE (2012)
40. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
41. Xing, Z., Stroulia, E.: Refactoring practice: How it is and how it should be supported-an eclipse case study. In: Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on, pp. 458–468. IEEE (2006)

Rodrigo Morales is a full-time lecturer at Concordia University in Montréal, Canada. He obtained his BS. degree in computer science in 2005 from Polytechnic of Mexico. In 2008, he obtained his MS. in computer technology from the same University, where he also worked as a Professor in the computer Science department for five years. He has also worked in the bank industry as a software developer for more than three years. He obtained his Ph.D. degree in computer engineering from Polytechnic of Montréal where he earned the best thesis award of 2017. He has published in top software engineering Journals and like IEEE TSE, ESEM, and JSS and top conferences including ICSE, and SANER. He is one of the main organizers of the International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT), co-located with ICSE, and actively participate as committee member of ICSME and ICPC conferences. His research interests include software design quality, energy efficiency, automated-refactoring, anti-patterns, mobile apps and Internet of Things. Web page: https://moar82.github.io/

Foutse Khomh Foutse Khomh is a Professor of Software Engineering at Polytechnique Montréal and FRQ-IVADO Research Chair on Software Quality Assurance for Machine Learning Applications. He received a Ph.D in Software Engineering from the University of Montreal in 2010, with the Award of Excellence. He also received a CS-Can/Info-Can Outstanding Young Computer Science Researcher Prize for 2019. His research interests include software maintenance and evolution, machine learning systems engineering, cloud engineering, empirical software engineering, and software analytic. His work has received three ten-year Most Influential Paper (MIP) Awards, and five Best/Distinguished paper Awards. He has served on the program committees of several international conferences including FSE, ICSM(E), SANER, MSR, ICPC, SCAM, ESEM and has reviewed for top international journals such as JSS, EMSE, TSC, TSE and TOSEM. He is program chair for Satellite Events at SANER 2015, program co-chair of SCAM 2015, ICSME 2018, PROMISE 2019, and ICPC 2019, general chair of ICPC 2018, SCAM 2020, and SANER 2020. He is on the steering committee of SANER, MSR, PROMISE, ICPC (chair), and ICSME(vice-chair). He initiated and co-organized the Software Engineering for Machine Learning Applications (SEMLA) symposium (https://semla.polymtl.ca/) and the RELENG (Release Engineering) workshop series (http://releng.polymtl.ca). He is an Associate Editor for IEEE Software. Web page: http://khomh.net/



Giuliano Antoniol (Giulio) worked in companies, research institutions and universities such as the Fondazione Bruno Kessler (FBK, forerly IRST), Trento (Italy) and the University of Sannio, Italy. In 2005 he joined the Polytechnique Montreal and was awarded the Canada Research Chair Tier I in Software Change and Evolution. He is a member of the editorial boards of the Journal of Software Maintenance and Evolution: Research and Practice and the Software Quality Journal. He served also as member of the editorial board of the Empirical software engineering journal, the information and software technology journal, the journal of software testing verification & Reliability, and IEEE software. He served as program chair, industrial chair, tutorial, and general chair of international conferences

and workshops. Dr Giuliano Antoniol served as Deputy Chair of the Steering Committee for the IEEE International Conference on Software Maintenance, International Symposium on Search-Based Software Engineering; he is presently member of the steering of the International Workshop on Search-Based Software Testing. Dr Giuliano Antoniol published more than 100 papers in journals and international conferences. He recently contributed to start and run the Software Engineering for Machine Learning Applications (SEMLA) and the International Workshop on Machine Learning Systems Engineering (iMLSE) initiatives. 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 and software testing.