Automated Refactoring: Can They Pass The Turing Test?

Rodrigo Morales Member, IEEE, Foutse Khomh Member, IEEE, and Giuliano Antoniol Senior Member, IEEE

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. We performed an empirical study to investigate whether the use of automated refactoring approaches 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 machine, and to rate the refactoring 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. 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. Results show that for 3 out of the 5 types of studied anti-patterns, developers cannot recognize the origin of the refactoring (i.e., whether it was performed by a human or an automatic tool). We also observe 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 refactoring 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 of human refactorings is slightly higher.

I. INTRODUCTION

In 1950, Alan Turing developed a test to assess a machine's ability to display behavior equivalent to that of a human being [1]. 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 want to gather quantitative and qualitative evidences that automated software refactorings can be applied to real-world software systems and with developer’s approval, by performing a Turing test, and a series of comprehension tasks on code transformations generated by human, and an automated software tool, respectively.

Context: Software systems age as the result of deviations of the original design due to the implementation of new features [2], changes in the business logic, etc. To combat software system deterioration, developers perform software maintenance tasks to combat design deterioration, and the correction of poor design choices (a.k.a., anti-patterns) [3]. Refactoring is a software maintenance activity that aims to reorganize code structure without altering system’s behavior [4]. 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 [5], [6], [7]. In the last decade, a plethora of tools and frameworks to perform refactorings automatically and/or semi-automatically have been proposed [8], [9], [10], [11]. 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 incorporating machine-generated code into their code bases [12]. 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, in a Turing test to assess the quality, pertinence, and understandability of automatically generated code. Note that we perform a Turing-test experiment to challenge the idea that automated refactorings can compete with humans ones in terms of quality, according to human judges, and not to prove/deny that automated approaches are intelligent. Hence, if automated refactorings pass the Turing test, that will mean that automatic refactoring is feasible in practice.

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 I, 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...
We measured the subjects’ performance using: (1) the 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.

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.

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

Deligiannis et al. [16] 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 [17] 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. [18] 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. [19] 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 plugin), 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. [20] 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. [21] added a new dimension with respect to previous studies on the effect of refactoring, that is productivity. Fowler [4] popularized the term refactoring, by defining some heuristics to improve the design of existing code while preserving its functionality. Brown [3] introduce the notion of anti-patterns as poor-design choices that hinders code evolution. Opdyke [14] is the first to formulate a set of pre- and post-conditions to automatize the refactoring of object-oriented systems. Mens et al. [15] 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.

1 Freelancer.com and Guru.com

II. RELATED WORK

Fowler [4] popularized the term refactoring, by defining some heuristics to improve the design of existing code while preserving its functionality. Brown [3] introduce the notion of anti-patterns as poor-design choices that hinders code evolution. Opdyke [14] is the first to formulate a set of pre- and post-conditions to automatize the refactoring of object-oriented systems. Mens et al. [15] 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.

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

Deligiannis et al. [16] 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 [17] 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. [18] 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. [19] 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 plugin), 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. [20] 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. [21] added a new dimension with respect to previous studies on the effect of refactoring, that is productivity.
They analyzed the impact 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. [23] 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.

B. Manual compared to automatic refactoring studies

Negara et al. [23] performed an empirical study to compare semiautomatic (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. [24] performed a case study with a R&D company to study the effects of automatic refactoring on code maintainability. They corroborate the statement by Murphy et al. [19] 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 cmdmended 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.

C. Evaluation of Refactoring process

Kataoka et al. [25] proposed an approach to measure the effect of refactorings on the maintainability of the code, using coupling metrics.

In a recent preliminary work by Arima et al. [26], 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 match 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.

III. Experimental Design

We perform two experiments 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 well-known. In fact, Bavota et al. [27] 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. Additionally, previous works have proposed approaches to detect these anti-patterns [28], [29]. Experiment 1 is about (1) 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 the study (2) to rank the refactoring code changes according to their perceived quality. In Experiment 2, we present participants with code refactored by both developers and an automated tool, and ask them to complete some comprehension tasks on these pieces of code. The aim is to assess the comprehensibility of code refactored by developers and automated tools. In each experiment, we work with two instances of each type of anti-pattern studied: one easy and one difficult. The level of difficulty is decided based on the number of lines of code added, modified, or deleted by a refactoring.

A. 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: Can developers tell the difference between automated and manual refactorings?

RQ2: Is there any difference between the perceived quality of automated and manual refactorings?

RQ3: Do automated refactorings affect code understandability?

B. Hypotheses

For RQ1, we test the following null hypothesis when subjects review refactored code changes.

H0_identify: Developers correctly differentiate the refactoring changes generated by an automated tool from the ones generated by another developer.
For RQ2, to examine differences in the perceived quality of the refactoring changes, we test the following null hypothesis. 

\[ H_{\text{rank}}: \text{Developers prefer manual refactoring over automated refactoring.} \]

RQ3 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 TLX, and percentage of correct answers. We test \( H_{\text{performance}}: \text{There is no difference between the performance of developers performing comprehension tasks on code refactored by developers compared to automatically refactored code.}\)

### C. Objects

We choose two instances of each type of studied anti-patterns. The rationale for using two instances is that one system could be intrinsically easier or more difficult to refactor. We decide on the level of difficulty of the systems based on different metrics that reflect effort required to perform the refactoring. In the case of the Blob, we measure the size of the Blob class (lines of code) and the number of data classes associated to the Blob class; in this work, a data class is a class composed mainly by attributes, and the only methods declared are accessor \( \text{(getters and setters)} \). 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 inline classes, which in turn require extra effort when values of NII are large. For Spaghetti Code, we measured Mccabe Complexity (McCC). Finally, for Long-parameter list classes, we consider the number of parameters.

We select 10 Java systems from the SF110 corpus \[^{30}\]. The SF110 corpus is a representative sample of 110 Java projects from SourceForge web site, 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. This allows us to validate, to some extent, that regression is not introduced in the code after refactoring either by our approach or by developers, when generating the objects of our experiment, and to have a common framework to import and build the systems (SF110 uses ant scripts to automate the building process).

The anti-patterns types studied are briefly introduced in Table 1. We provide the name, a brief description and the refactoring strategies used to fix them according to the literature of anti-patterns \[^{3},^{4}\].

To find the 10 anti-pattern’s instances to refactor we use the following procedure. First, we use our automated refactoring approach RePOR \[^{32}\] on simulation mode to detect anti-patterns and generate a list of refactoring candidates to remove the anti-patterns without applying it to the code. To avoid the negative effect of learning that could happen when a developer get familiar with a system after performing several tasks on it, we opt for selecting each anti-pattern’s instance from a different system. We manually validate each anti-pattern’s instance to be sure that the anti-pattern detected was indeed an anti-pattern based on the guidelines depicted by Brown and Fowler \[^{3},^{4}\].

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

| Name       | Description                                                                 | Refactoring(s) strategy                                                                 |
|------------|----------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Blob (BL)  | 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 (e.g., other classes). | Clean Up Class (CL) \[4\]. Move the attributes and methods of the LC to another class in the system. |
| Lazy Class (LC) | Small classes with low complexity that do not justify their existence in the system. | Inline Class (IC) \[6\]. Move the attributes and methods of the LC to another class in the system. |
| Long Parameter List (LP) | A class with one or more methods having a long list of parameters, especially when two or more methods are sharing a long list of parameters that are semantically connected. | Introduce Parameter Object (IPO) \[4\]. 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 parameter through the parameter object. |
| Spaghetti Code (SC) | A class without structure that declares long methods without parameters. | Replace Method With Method Object (RMWO) \[4\]. Extract long methods into new classes so that all local variables become fields on that object. |
| Speculative Generality (SG) | 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) \[6\]. Move the attributes and methods of the child class to the parent and remove the abstract modifier. |

\[^{2}\]https://sourceforge.net/

\[^{3}\]https://github.com/moar82/RefGen
we imposed to them for this task was that the developer’s unit tests provided in the benchmark should pass, and if necessary they should perform pertinent updates on the unit tests to ensure that no regression is introduced in the code during the execution of the task. Each developer performed two refactorings, and these developers were excluded from the experiment to avoid introducing any type of bias in the experiment (even though they only worked on two anti-pattern’s instances each).

### D. Subjects

We recruited participants to experiment $E_1$ by posting on 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 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 developer as their main occupation, 7 participants declared their main occupation to be researchers, 6 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 only and 9 on open-source. Fifty percent (40 out of 80) of participants declared to have more than 5 years of experience as developer. 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.

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-based software for the industry. To control for possible confounding factors, we asked freelancers to provide their number of years of experience as professional developers, and Java’s level of confidence (using a Likert scale from 1 to 5).

### E. Independent variable

The independent variable for the two experiments 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.

### F. Dependent Variables

In $E_1$, 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.

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 \[33\]. 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, for a question on the code references to a given object, if there are ten references but the subject finds only four, the percentage of correct answers is forty for that question.

### G. Questions

For $E_1$, we use an online survey system (Jotform\[4\]) where we present the refactoring change, the repository URL 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 picture of the

---

### Table II: List of systems from where we extract the anti-pattern’s instances studied

| ID | System                          | Description                                      | Anti-pattern type | Package                                                | Difficulty |
|----|---------------------------------|--------------------------------------------------|-------------------|--------------------------------------------------------|------------|
| 86 | Advanced T-Robots               | Fighting robots Arena                            | Collapse hierarchy | net.virtualinfinity.atrobots.arena                      | Hard       |
| 47 | dvdl-homevideo                  | Application to burn DVDs                         | Lazy class        | default package                                        | Hard       |
| 70 | EchoDep                         | Digital preservation application                  | Spaghetti Code    | edu.uuci.ndipp.hubandspoke.workflow                    | Hard       |
| 110| FireBird                        | A relational database manager                    | Blob              | org.firebirdsql.gds.impl.jni                          | Easy       |
| 81 | JavAthena                       | Online role playing game                          | Blob              | org.javathena.core.data                                | Easy       |
| 52 | Lagoon                          | A XML-based framework for web site maintenance   | Collapse hierarchy | nu.staldal.laggon.core                                 | Easy       |
| 101| SAP NetWeaver                   | Server Adapter for Eclipse                       | Lazy class        | com.sap.netweaver.porta.mon                            | Easy       |
| 103| Sweet Home 3D                   | An interior design application                    | Long-parameter List| com.ecks.sweethomeML.3D                                | Hard       |
| 104| Vuze                            | BitTorrent Client                                | Long-parameter List| org.gudy.azureus2.core3.tracker.client.impl.fb         | Easy       |
| 83 | Xbus                            | It is a central Enterprise Application Integration (EAI)| Blob              | net.sf.xbus.technical.mail                             | Hard       |

---

[^4]: http://www.jotform.com
refactoring changes applied, we also provide a link to the repository containing the original source; for example, if they want to study deeper the impact of the refactorings applied, they could clone the repository and apply the patch. In Figure 1, we show an example of a refactoring change from our online survey.

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.

For $E_2$, we used comprehension questions to elicit comprehension tasks and collect data on the subject’s performance. As in [20], we consider questions in three of the four categories of questions regularly asked and answered by developers [13]: (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 questions of the third category.

- 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?
- Question 1: Where is this method called or this type referenced?
- Question 2: What data can we access from this object?
- Question 1: How are these types or objects related?

For example, with system 47 dvd-homevideo (cf., Table I), we replace “this behavior” in Question 1, Category 1, by “creating a new form”. For category 2, we acknowledge that the questions might be answered by developers using 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.

H. Anonymization of new code lexicon

Identifier names and comments (code lexicon), when thoughtfully assigned, can support developers to better understand a software system. However, current exiting automatic refactoring tools 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 make a fair Turing test between the refactoring changes generated by humans and by automated tools, 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.

I. 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. The level of difficulty is assessed based on the first two authors’ experience, and the feedback obtained from a pilot study with two Post-docs 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 ask respondents to identify the origin of the patch, we also asked them to justify their response in 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 III. First column is the number of question and the rest of the columns corresponds...
Table III: E1 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        | SC-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-H| BL-110-H|
| 5        | LP-103-H| LC-47-H| CH-52-H| CH-86-M|

Figure 1: Example of a refactoring change from the online survey.

For E1 we first briefly introduced the description of the anti-patterns using Table I. Next, we asked them, not to base their judgment on code lexicon, and to accept that they will only focus on code structure and the quality of it, and not in indentation, naming conventions, space, etc.

For E2 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 collecting their data, after they finished the experiment.

J. 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 provide 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 encourage them to base their actions on their work experience and reasoning.

K. Analysis method

In RQ1, to attempt rejecting $H_{identify}$, 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 [34], which checks whether a proportion vary between two samples. We also compute the odds ratio ($OR$) [34] 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:

$$OR = \frac{p}{1-p} \div \frac{q}{1-q}.$$  

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. In RQ2, 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
The abstract class is useless (the change is better). I don’t think a soft can detect it.”; (2) other respondents refer to the size–or complexity of the refactoring performed to justify their wrong answer: “Small changes”.

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 IV) and developer’s confidence for each patch reviewed (Table V).

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.

Concerning the number of correctly identified refactoring changes by origin (columns 5-6, Table VI), we corroborate what we observed in the overall results: that respondents have more difficulty to identify refactoring changes generated by human than by machine (rows 1, 2, 4, Table VI). This is not the case for developers with one up to two years of experience, who correctly identified more refactoring changes written by human than those written by machine.

With respect to confidence declared when identifying the refactoring changes, we observed that respondents felt confident enough, as in 60% of the questions they selected a confidence level between 4 and 5. We also observed, according to respondent’s confidence level, 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, reach 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%).

If we analyze 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 seems to fit better the results, 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).

In Table VII 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

\[ \delta (ES) \] which indicates the magnitude of the effect of the treatment on the dependent variable. The effect size is consider negligible if \( < 0.147 \), small if between 0.147 and 0.33, medium if between 0.33 and 0.474, and large if \( > 0.474 \). For RQ3 and RQ4 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 \( \delta \) (ES).

IV. STUDY RESULTS

We now describe the collected data and present the results of our study, answering the four RQs formulated in Section III.

A. RQ1: Can developers tell the difference between automated and manual refactorings?

In Table IV we summarize the results of our Turing-test-like-survey. The first column indicates all for all the anti-pattern types studied, or the abbreviation of each types; columns 2 to 4 are the percentage of correct, wrong, or unknown answers; columns 5 to 7 are the corresponding percentages with respect to the origin of the changes i.e., machine (m); and columns 8 to 10 the corresponding percentage 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 answers), first row, columns 1-3. 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 failed the Turing test. Conversely, Speculative generality, Blob and lazy class were correctly identified in less than 50% of cases, and therefore, we can claim that they passed the Turing test.

If we study the percentage of correct answers by refactoring origin, we observe that respondents found it more difficult to identify human patches (38%), while machine patches were more easily identified (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 result is surprising since, the refactoring type applied for removing the two different SG instances, which is Collapse Hierarchy (CH), is the same. We manually examined the justification statements of the respondents that failed to distinguish CH changes and observed that all belong to the SG anti-pattern instance that we classified as easy one. A total of 14 respondents failed to identify the origin of the changes generated by the automatic tool to remove SG anti-pattern. We summarize the type of justifications given when selecting a wrong origin. (1) Some respondents doubted the ability of a software tool to perform this refactoring type. For example: “The abstract class is useless (the change is better). I don’t
origin. Fisher’s exact test indicates whether a significant difference of proportions between correctly identified changes among those automatically generated and those generated by humans exists. Odds ratio indicates the probability of a respondent to correctly identify a change according to the origin of the change analyzed.

The first column of Table VII 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 \(H_{\text{identify}}\). In the first row, the result of the Fisher’s exact test when considering all anti-pattern’s types studied is statistically significant, with an OR greater than one, confirming that changes generated by machine are easier to identify than the ones generated by humans.

With respect to anti-pattern’s types (rows 2 to 6 Table VII), 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 to be correctly identified when they are refactored manually (OR < 0). 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. Hence, they should have been automatically generated. To support this claim, consider Table IV and observe that the largest proportion of correct answers for refactoring long-parameter list corresponds to respondents choosing machine origin (80%), while another considerable proportion of respondents, for the same anti-pattern type, failed selecting machine origin when the patch’s origin was human (55%); the remaining 10% of respondents declare themselves unable to decided (they did not know). That indicates that most of respondents consider the refactoring of long-parameter list proposed in this study the output of an automatic approach.

Conversely, for Speculative Generality anti-pattern type, 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 proposed by a 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 (Collapse hierarchy). Hence, they could have been misled by the wide availability of automatic refactoring tool support for refactoring long-parameter list anti-patterns when making their choices, in addition to the complexity of the refactorings proposed.

For the remaining anti-patterns types their \(p\) – values are > 0.01, hence we cannot reject \(H_{\text{identify}}\) for those types. Therefore, we conclude that developers cannot differentiate between refactorings changes made by human and machine for those types.

### Table IV: E₁ RQ1 overall results

| ap-type | correct | wrong | unknown | correct-m | wrong-m | unknown-m | correct-h | wrong-h | unknown-h |
|---------|---------|-------|---------|-----------|---------|-----------|-----------|---------|-----------|
| all     | 45.00%  | 45%   | 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%    |
| tc      | 47.50%  | 45%   | 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%     | 50.00%    | 55.00%  | 10.00%    |
| sc      | 50.00%  | 45%   | 5.00%   | 72.50%    | 22.50%  | 5.00%     | 42.50%    | 37.50%  | 20.00%    |
| sg      | 45%     | 45.00%| 10.00%  | 22.50%    | 52.50%  | 25.00%    | 35.00%    | 27.50%  | 17.50%    |

### Table V: E₁ 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        |

### Table VI: E₁ 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         |

### Table VII: 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 | F
|---------|-----------|---------|-----------|---------|---------------|----|---
| all     | 104       | 96      | 76        | 124     | <0.01         | 1.77| 4.68
| bl      | 15        | 25      | 9         | 31      | 0.22          | 2.05| 4.68
| tc      | 19        | 21      | 14        | 26      | 0.36          | 1.67| 4.68
| lp      | 32        | 8       | 14        | 26      | <0.01         | 7.22| 4.68
| sc      | 29        | 11      | 17        | 23      | 0.01          | 3.51| 4.68
| sg      | 9         | 31      | 22        | 18      | <0.01         | 0.24| 4.68

B. RQ2: 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
Table VIII: $E_1$, 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 |

the same appreciation for automatically generated 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 VIII we present the respondent’s median rates for the refactoring changes presented in $E_1$. Respondents input their rate for each refactoring change using a Likert-scale from 1 to 5. When considering all anti-patterns types (first row) both source of changes (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 medium Cliff’s $\delta ES$. Hence, we reject $H_{0\text{rank}}$ 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.

C. RQ3: 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 IX summarizes the median of collected data. It presents, for all the systems, the median of each dimension: time, effort, and percentage of correct answers (Rows 1-3). In addition, we divide the refactoring changes in two groups: those who passed the Turing test in $E_1$ (T. passed), and those who failed the test (T. failed), respectively. In columns 5, and 6, we report the $p-value$ of the Mann-Whitney tests and the Cliff’s $\delta ES$ 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.

Table IX: 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    | 1.575.50    | 0.08      | small |
| ALL      | Effort (TLX index) | 40.00    | 40.71       | 0.82      | negligible |
| ALL      | % of correct answers | 67.00    | 67.00       | 0.76      | negligible |
| T. passed| Time (seconds) | 1.100.00    | 1.060.00    | 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| Time (seconds) | 1.176.00    | 1.060.00    | 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 |

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 small and not statistically significant. For the percentage of correct answers, the effect size is large, though 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 and automatically refactored, we reject $H_{\text{performance}}$. Hence, we conclude that for Blob, Lazy class, and Speculative Generality, 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 consequences 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.

V. 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.

However, when considering the types of the anti-patterns removed by the refactorings, automatically generated changes that removed long-parameter list were easier to identify than
their manual counterpart. The situation was reversed for the Spaghetti Code, where manual changes were easier to identify than automatically generated changes.

Concerning 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 III-H), the two refactoring changes performed by the human and the tool were semantically equivalent. In the hard instance of Long-parameter list, there was a small difference between the two solutions; that is the human’s one declared the new class in the same file, while our automated approach created a new file; besides that small difference, we can consider both solutions to be semantically equivalent too. As we mentioned in Section IV, 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 LP 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 instance of LP anti-pattern resides.

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.

With respect to the two instances of Spaghetti Code, the solutions generated for the easy instances and for automatic and manually refactoring used the same strategy: replace long method with method object. Since they only differ in the name selected for the new class, and we normalized the names, we can consider them semantically equivalent. The solutions for the two hard instances of Spaghetti Code differed in: (1) the way they received the object of the source class. In the solution provided by the automatic refactoring tool, the source class is passed as parameter of the long method call, while in the human solution, the source class is passed in the constructor call of the new class. (2) The way the automatic and manual solutions handled a private field declared in the Spaghetti code class. In the machine solution, the field was moved to the new extracted class, while in the human solution, a new getter was created to access the private field from the Spaghetti code class. Note that the methods called from the Spaghetti Code class by the long method extracted are static. Hence, if we move the private field to the new class, there is no need to pass the Spaghetti code class as a reference. However, in the manual solution, the extracted method still needs to have a reference to the Spaghetti code class, to access the private field (through the getter). This small detail was noticed by one of the respondents, 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 method from c1 to class c2 and pass a reference of m1 source’s class, because that reference is not used”. Conversely, on the manual solution, one respondent argued that the addition of the getter, and the signature of the new method does not seem to be automatically generated: “A getter was introduced on the Spaghetti code class (where it apparently belongs best) simultaneously with moving the bulk of the long method to the new class”. That variation between the automatic solution for the hard instance of Spaghetti code with respect to the manual one could be easily solved by adding a validation to prevent passing the Spaghetti Class as parameter to the new extracted class, when the methods used are static, to provide a more adequate solution for developers. Hence, toolsmiths should pay attention to corner cases like these, which can only be revealed by thoroughly testing their tools with several systems.

Concerning quality perceived by developers, we observed that only for the Blob anti-pattern type, which passed the Turing test, the difference of rates achieved between manual and automatic refactoring changes is a statistically significant, favoring manual refactoring. We highlight differences in the refactoring strategies employed by human and the automatic approach that are worth to discuss. First of all, our automated approach selected move method as a mean to decompose the two instances of 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. For the Easy instance, one respondent commented about the manual solution: “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 fields to new classes according to their names. That would be hard for a machine to achieve, unless they implement some knowledge about the code lexicon. The refactoring strategy followed by the automated approach consists of delegating three methods to a related class declared as an instance variable of the Blob class. Hence a coupling between the Blob and the related class already exist, but still the solution proposed was not attractive for the respondents of $E_1$ as expressed by one respondent: “I don’t like it at $^5$We refer to m1 as the long method extracted for the SC class $^6$We refer to c1 as the SC class $^7$We refer to c2 as the new extracted class from the SC class
all. It increased coupling with no benefit. With respect to the refactoring strategy applied for the hard instance of Blob, the automated solution consists of delegating 12 methods to a class where an association relationship exists. Again, one respondent complained that the coupling between these two classes increased dramatically, which was not the case. On the other hand, the solution proposed by developers for the hard Blob’s instance is clearly inadequate. It consists of extracting two fields and their corresponding getters and setters to a new class. Clearly, this solution cannot reduce the size of the Blob class, but introduce coupling to a new data class. Respondents rated the refactoring solutions generated by our automatic refactoring tool to fix the Blob with average scores of 2.6 and 2.7 for the easy and the hard instances respectively, while the human ones obtained 4.4 and 2.5 respectively. Despite the fact that machine-generated refactorings for the Blob instances do not achieve the best rate, the perceived quality showed little variation between the two instances, despite their complexity. However, the rate achieved by the two manual refactoring changes of the two Blob instances varied considerably. We suggest that from a practical point of view, by using automated approaches, practitioners benefit from a standardized level of quality of the refactoring changes applied to their systems, and reduce costs on maintenance tasks. 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 lexicon. The first aspect has been already discussed by Ouni et al., while (2) 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.

Concerning the impact on code comprehension, we could not make an analysis by anti-pattern type as we only have 6 points for each 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 by 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 X 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 X i.e., participants mostly had the same performance on refactorings from the same origin no matter their level of expertise/Java confidence, which reinforce the claim that the refactoring changes are what did matter.

### VI. 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 [37].

#### A. Construct Validity

Construct validity threats concern the relation between theory and observations. 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 [3], [4]. 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. 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.

#### B. Internal Validity

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

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

| 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          | 0.1539                          |
balanced the groups (alternating difficulty level, and origin); then we randomize the appearance order of the refactoring changes for each group. For $E_2$, the freelancers have to perform two comprehension tasks. To mitigate the learning effect, the systems were presented in the same order for systems refactored by human and machine. For example, there is a task that includes the system pair 47, 52 for both human and machine. It is possible that developers spent more time in system 47, as they have to get familiar with the task, developer environment, etc., than when they work with the second system (52). Hence, the results of both origins (human, machine) will exhibit additional extra time as a result of participants getting familiar with the experiment environment, without biasing the results towards any side.

2) Selection: 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.

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.

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.

C. 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.

D. 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 [38], including a sample of the questionnaire and the comprehension tasks, and raw data to compute the statistics. The systems analyzed from the SP110 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](https://github.com/moar82/RefGen).

E. External Validity

We performed our study on 10 different real-world systems belonging to different domains and with different sizes (see Table [II]). 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 non familiarity to diff notations could have on the responses of our study, we explained the notation to participants prior to the study and provided them multiple examples in a guidelines document. We cannot claim that our results can be generalized to other Java systems, systems in other programming languages, and to other subjects. Our future work includes replicating this study in other contexts, with other subjects, other questions, other anti-patterns and other software systems.

VII. 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 refactoring 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] A. M. Turing, “Computing machinery and intelligence,” in Parsing the Turing Test. Springer, 2009, pp. 23–65.
[2] D. L. Parnas, “Software aging,” in ICSE ’94: Proc. of the 16th Int’l conference on Software engineering. IEEE Computer Society Press, 1994, pp. 279–287.
[3] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st ed. John Wiley and Sons, March 1998.
[4] M. Fowler, Refactoring: improving the design of existing code. Pearson Education India, 1999.
[5] W. G. Griswold and D. Notkin, “Automated assistance for program restructuring,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 2, no. 3, pp. 228–269, 1993.

[6] A. Murphy-Hill, E. Black, “Refactoring tools: Fitness for purpose,” Software, IEEE, vol. 25, no. 5, pp. 38–44, 2008.

[7] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, “Use, disuse, and misuse of automated refactorings,” in Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012, Conference Proceedings, pp. 233–243.

[8] I. H. Moghadam and M. O. Cinneide, “Automated refactoring using design differencing,” in Software Maintenance and Reengineering (CSMR), 16th European Conference on, ser. Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR. IEEE Computer Society, 2012, Conference Proceedings, pp. 43 – 52. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2012.15

[9] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, “Improving multi-objective code smells correction using development history,” Journal of Systems and Software, vol. 105, no. 0, pp. 18 – 39, 2015.

[10] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, and G. Antoniol, “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, 2016, Conference Proceedings, pp. 24–35.

[11] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, “On the use of developers’ context for automatic refactoring of software anti-patterns,” Journal of Systems and Software, vol. 128, pp. 236 – 251, 2017.

[12] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “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. IEEE, 2012, pp. 3–13.

[13] J. Sillito, G. C. Murphy, and K. De Volder, “Asking and answering questions during a programming change task,” IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 434–451, 2008.

[14] W. F. Odpke, “Refactoring object-oriented frameworks,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[15] T. Mens and T. Tourwé, “A survey of software refactoring,” Software Engineering, IEEE Transactions on, vol. 30, no. 2, pp. 126–139, 2004.

[16] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, “A controlled experiment investigation of an object-oriented design heuristic for maintainability,” Journal of Systems and Software, vol. 72, no. 2, pp. 129 – 143, 2004.

[17] Z. Xing and E. Stroulia, “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. IEEE, 2006, pp. 458–468.

[18] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, “Does god class decomposition affect comprehensibility?” in IASTED Conf. on Software Engineering, 2006.

[19] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” Software Engineering, IEEE Transactions on, vol. 38, no. 1, pp. 5–18, 2012.

[20] M. Abbas, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, “An empirical study of the impact of two anti-patterns, blob and spaghetti code, on program comprehension,” in Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. on, March 2011, pp. 181–190.

[21] R. Moser, P. Abrahamssoon, W. Pedrycz, A. Sillitti, and G. Succi, “A case study on the impact of refactoring on quality and productivity in an agile team,” in Balancing Agility and Formalism in Software Engineering. Springer, 2008, pp. 252–266.

[22] M. Kim, T. Zimmermann, and N. Nagappan, “An empirical study of refactoring challenges and benefits at microsoft,” IEEE Transactions on Software Engineering, vol. 40, no. 7, pp. 633–649, July 2014.

[23] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, “A comparative study of manual and automated refactorings,” in ECOOP 2013 – Object-Oriented Programming, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 552–576.

[24] G. Szöke, C. Nagy, P. Hegedüs, R. Ferenc, and T. Gyimóthy, “Do automatic refactorings improve maintainability? an industrial case study,” in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Sept 2015, pp. 429–438.

[25] Y. Katoaka, T. Inai, H. Andou, and T. Fukaya, “A quantitative evaluation of maintainability enhancement by refactoring,” in Software Maintenance, 2002. Proceedings. International Conference on. IEEE, 2002, pp. 576–585.

[26] R. Arima, Y. Higo, and S. Kusumoto, “Toward refactoring evaluation with code naturalness,” in Proceedings of the 26th International Conference on Program Comprehension, ser. ICPC ’18. IEEE Press, 2018.

[27] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, “Do they really smell bad? a study on developers’ perception of bad code smells,” in Software Maintenance and Evolution (ICSEME), 2014 IEEE Int’l Conference on. IEEE, 2014, pp. 101–110.

[28] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A. Le Meur, “Decor: A method for the specification and detection of code and design smells,” Software Engineering, IEEE Transactions on, vol. 36, no. 1, pp. 20–36, 2010.

[29] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in IEEE Int’l Conference on Software Maintenance, ICSM. IEEE Computer Society, 2004, Conference Proceedings, pp. 350–359.

[30] G. Fraser and A. Arcuri, “A large scale evaluation of automated unit test generation using evosuite,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 24, no. 2, p. 8, 2014.

[31] O. Seng, J. Stammel, and D. Burkhart, “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.

[32] R. Morales, F. Chicano, F. Khomh, and G. Antoniol, “Efficient refactoring scheduling based on partial order reduction,” Journal of Systems and Software, 2018.

[33] D. Sharek, “A useable, online nasa-tlx tool,” in Proceedings of the Human Factors and Ergonomics Society Annual Meeting, vol. 55, no. 1. SAGE Publications Sage CA: Los Angeles, CA, 2011, pp. 1375–1379.

[34] D. J. Sheskin, Handbook of parametric and nonparametric statistical procedures. crc Press, 2003.

[35] N. Cliff, Ordinal methods for behavioral data analysis. Psychology Press, 2014.

[36] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, “Ordinal methods for behavioral data analysis,” in Ordinal methods for behavioral data analysis. Psychology Press, 2014.

[37] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, “Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen’s indices the most appropriate choices,” in annual meeting of the Southern Association for Institutional Research, 2006.

[38] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in software engineering. Springer Science & Business Media, 2012.

[39] M. et al., “Automated refactoring additional material site,” http://www.swat.polymtl.ca/romales/sereref1/