Interactive Software Refactoring Bot

Vahid Alizadeh, Mohamed Amine Ouali, Marouane Kessentini and Meriem Chater
Software Engineering Intelligence Lab, CIS Department, University of Michigan, USA
alizadeh,mouali,marouane,meriemchater@umich.edu

***Abstract*—The adoption of refactoring techniques for continuous integration received much less attention from the research community comparing to root-canal refactoring to fix the quality issues in the whole system. Several recent empirical studies show that developers, in practice, are applying refactoring incrementally when they are fixing bugs or adding new features. There is an urgent need for refactoring tools that can support continuous integration and some recent development processes such as DevOps that are based on rapid releases. Furthermore, several studies show that manual refactoring is expensive and existing automated refactoring tools are challenging to configure and integrate into the development pipelines with significant disruption cost.**

In this paper, we propose, for the first time, an intelligent software refactoring bot, called RefBot. Integrated into the version control system (e.g. GitHub), our bot continuously monitors the software repository, and it is triggered by any "open" or "merge" action on pull requests. The bot analyzes the files changed during that pull request to identify refactoring opportunities using a set of quality attributes then it will find the best sequence of refactorings to fix the quality issues if any. The bot recommends all these refactorings through an automatically generated pull-request. The developer can review the recommendations and their impacts in a detailed report and select the code changes that he wants to keep or ignore. After this review, the developer can close and approve the merge of the bot's pull request. We quantitatively and qualitatively evaluated the performance and effectiveness of RefBot by a survey conducted with experienced developers who used the bot on both open source and industry projects.

***Index Terms*—Software bot, refactoring, quality**

I. Introduction

Refactoring, defined as a set of program transformations intended to improve the system design while preserving the desired behaviour, is becoming a critical software maintenance activity, especially with the growing complexity of software systems [1]. A recent study by the US Air Force Software Technology Support Center (STSC) shows that restructuring the code of a large project reduced developers' time by over 60% when introducing new features. However, refactoring is expensive. Developers take an average of 6 weeks to refactor the design of medium-size projects (around 30K LOC) [2]. There has been much work done on various techniques and tools for software refactoring [3]–[7] and these approaches can be classified into three main categories: *manual*, *semi-automated* and *fully-automated* approaches.

In manual refactoring, the developers refactor with no tool support except the execution part, identifying the parts of the program that require attention and performing all aspects of the code transformation by hand. It may seem surprising that a developer would eschew the use of tools in this way, but Murphy-Hill et al. [8] found in their empirical study of the developers' usage of the Eclipse refactoring tooling that in almost 90% of cases the developers performed refactorings manually and did not use automated refactoring tools. Kim et al. [9] confirmed this observation, finding that the interviewed developers from Microsoft preferred to perform refactoring manually in 86% of cases. Despite its apparent popularity, manual refactoring is very limited. However, several studies have shown that manual refactoring is error-prone, time-consuming, not scalable and not practical for extensive application of refactorings to fix major quality issues [4], [10], [11]. Although developers are doing refactorings manually, the surveys confirmed that they are not frequently refactoring their code because of the above limitations.

In fully-automated refactoring, developers provide their code as input, and the tool will provide refactoring recommendations automatically [12]. The majority of existing automated refactoring tools assume that developers want to fix code smells [13]–[15]. This approach is appealing, in that it is a complete solution and requires little developer effort, but it suffers from several serious drawbacks as well. First, the recommended refactoring sequence may change the program design radically, and this is likely to cause the developer to struggle to understand the refactored program, and they lose any control of the introduced code changes. Second, it lacks flexibility since the developer has to either accept or reject the entire refactoring solution. In fact, developers intentions may not be, most of the time, fixing code smells or the majority of them. Third, it fails to consider the developer perspective, as the developer has no opportunity to provide feedback on the refactoring solution as it is being created. Furthermore, as development must halt while the refactoring process executes, fully-automated refactoring methods are not useful for floss refactoring where the goal is to maintain good design quality while modifying existing functionality. The developers have to accept the entire refactoring solution even though they prefer, in general, step-wise approaches where the process is interactive and they have control of the refactorings being applied [16]. Finally, one of the significant limitations of existing automated refactoring tools is the high configuration effort required to integrate them into the current development pipeline of the team/company. In fact, several companies are now using continuous integration and DevOps, which make the adoption of current automated refactoring tools very challenging.

Recently, few interactive refactoring techniques were proposed [17]–[20]. They provide to the developers the flexibility to approve or reject the recommended refactoring that can improve the quality. However, this interaction process is time-consuming, and developers get frustrated from providing feedback on files that are out of their interests/ownership or navigating through many refactoring recommendations/strategies to improve several quality metrics.

To address all the above challenges, we propose the first attempt to design and build an intelligent refactoring bot as a GitHub app that can be easily integrated into any project repository on GitHub. The bot can be customized to monitor the quality in the repository after some pull-requests repeatedly or automatically executed when the quality analysis shows a significant decrease. The bot analyzes the files changed during that pull request(s) to identify refactoring opportunities using a set of quality attributes then it will find the best sequence of refactorings to fix the quality issues if any. The bot recommends all these refactorings through an automatically generated pull-request. The developer, whenever available without interrupting the development pipeline, can review the recommendations, and their impacts in a detailed report and select the code changes that he wants to keep or ignore. After this review, the developer can close and approve the merge of the bot's pull request. We quantitatively and qualitatively evaluated the performance and effectiveness of RefBot by a survey conducted with experienced developers who used the bot on both open source and industry projects.

The primary contributions of this paper can be summarized as follows:

1) The paper introduces a novel way to refactor software systems using autonomous intelligent software bots but still considering developers interaction to review the generated pull-request.
2) We propose an implementation of the refactoring bot as a Git app that can be quickly adopted in a continuous integration environment or DevOps process.
3) The paper reports the results of an empirical study on an implementation of our approach. The obtained results provide evidence to support the claim that, on average, our bot is more efficient than existing automated refactoring techniques based on a benchmark of six open source systems and one industrial project. The paper also evaluates the relevance and usefulness of the suggested refactorings for software developers in improving the quality of the modified files in several pull-request.

The remainder of this paper is structured as follows. Section 2 presents the relevant related work. Section 3 describes our intelligent refactoring bot, while the results obtained from our experiments are presented and discussed in Section 4. Threats to validity are discussed in Section 5. Finally, in Section 6, we summarize our conclusions and present some ideas for future work.

II. Related work

Our work is mainly related to [21]–[31] 1) refactoring recommendations; 2) empirical studies on refactoring, mostly the ones investigating its relationship with fault-proneness; and 3) software bots.

A. Refactoring Recommendation

Much effort has been devoted to the definition of approaches supporting refactoring. One representative example is JDeodorant, the tool proposed by Tsantalis and Chatzigeorgiou [16].Our paper is mostly related to approaches exploiting search-based techniques to identify refactoring opportunities, and our discussion focuses on them since the bot is based on multi-objective refactoring. We point the interested reader to the survey by Bavota [32] for an overview of approaches supporting code refactoring.

O'Keeffe and Cinnéide [33] presented the idea of formulating the refactoring task as a search problem in the space of alternative designs, generated by applying a set of refactoring operations. Such a search is guided by a quality evaluation function based on eleven object-oriented design metrics that reflect refactoring goals. Harman and Tratt [34] were the first to introduce the concept of Pareto optimality to search-based refactoring. They used it to combine two metrics, namely CBO (Coupling Between Objects) and SDMPC (Standard Deviation of Methods Per Class), into a fitness function and showed its superior performance as compared to a mono-objective technique [34].

The two aforementioned works [33], [34] paved the way to several search-based approaches aimed at recommending refactoring operations [17], [18], [35]–[38]. Several other studies proposed refactorings at the model level as well [21], [22], [26], [27], [29], [39]–[41]. A representative example of these techniques is the recent work by Alizadeh et al. [20], who proposed an interactive multi-criteria code refactoring approach to improve the QMOOD quality metrics while minimizing the number of refactorings. In our approach, we decided to rely on a simpler optimization algorithm by only considering the refactoring of recently changed files in other pull requests rather than the root-canal refactoring approach of Alizadeh et al. [20].

B. Empirical Studies on Refactoring

Empirical studies on software refactoring mainly aim at investigating the refactoring habits of software developers and the relationship between refactoring and code quality. We only discuss studies reporting findings relevant to our work. Murphy-Hill [8] investigated how developers perform refactorings. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment and information extracted from versioning systems. Among their several findings, they show that developers often perform *floss refactoring*; namely, they interleave refactoring with other programming activities, confirming that refactoring is rarely performed in isolation. Kim [9] present a survey of software refactoring with 328 Microsoft engineers. They show that the major obstacle of adopting many existing refactoring tools is their configuration and painful integration within their pipelines without disturbing developers with their current focus in terms of meeting deadlines and making regular code changes. Those findings stress out the need for refactoring bots that can be adopted for continuous integration without considerable configuration effort.

C. Software Bots

The design and implementation of software bots are still in its infancy with a significant focus on chatbots. For instance, Lebeuf et al. [42], [43] discussed the potential of using chat bots in software engineering and how they can be helpful to increase collaborations between programmers. The authors also proposed a possible classification of potential benefits of using software bots in various domains, especially to improve the productivity of developers.

An extensive empirical study of over 90 software bots was performed by Wessel et al. [44] to provide a classification and taxonomy for them. They found that around 21 bots were actually tried on GitHub repositories and the dominant majority are around testing but without providing any code actions or recommendations to developers. The authors found that none of these bots provides explanations of their analysis which reduced the adoption by developers.

Some examples of regression testing bots include Travis CI and the bot designed by Urli et al. [45] to repair bugs. These tools did not open a new pull-request, but they are executed manually by the developers where they can check the recommended patches. Another bot related to quality assessment but not refactoring is Fix-it [46]. It is mainly limited to a few types of code changes, mainly targeting dynamic analysis metrics.

Finally, Wyrich et al. [47] proposed a vision paper to emphasize the importance of refactoring bots and motivates their potential use in practice. They proposed a prototype, not a complete bot, by running SonarQube to detect code smells. However, the work is still in its initial stage where refactorings are not recommended yet.

III. Approach

We developed the "Refactoring Bot" (RefBot) as a GitHub App using which the workflow can be automated, and the developers can integrate the bot easily to any repository of their interest. The overview of the Refactoring bot is shown in Figure 1.

A. RefBot Installation

The first step of utilizing the Refactoring bot is to install its GitHub application on organizations or user accounts and to set up the appropriate permissions. As the installation page in Figure 2 shows, the user can select the repositories. Therefore, RefBot is granted access to the specific repositories via the GitHub API. RefBot has read and write permissions to "Pull Requests" and "WebHook", and also is subscribed to "Pull Requests" and its related "reviews and comments" events.

After this step, RefBot automatically sets up a web-hook for the developer's profile which means the permitted activities on the selected repositories will be posted as JSON-formatted payloads to the designated external server.

B. Processing a Pull Request

RefBot continuously monitors the actions performed on the repository by checking the subscribed payloads delivered to its server. In our current configuration, opening a new pull request action triggers the RefBot's workflow.

First, the commits in the pull request are compared to the commit at the point where the branch is created to extract the list of all files changed by the pull request. Then, two versions of the files, before and after the pull request, are downloaded to the external server for further processing and modifications.

By processing only the changed files by the pull request, we ensure that the developers are provided with the reports and refactorings limited to the codes they recently modified. This feature facilitates the evaluation of recommended refactorings and is aligned with the idea of maintaining/improving the code quality in the continuous development process.

*1) **Calculating Quality Changes**:* The RefBot analyses the code quality of the extracted files. For this purpose, we adopted QMOOD quality assessment methodology, which is a hierarchical model for object-oriented designs [48].

QMOOD model comprises of four levels from which we utilized the first level, Design Quality Attributes, to measure code quality changes of the pull request. This quality attributes set is defined based on ISO 9126 and consists of "Reusability", "Flexibility", "Understandability", "Functionality", "Extendibility", and "Effectiveness". Table I describes the QMOOD metrics definitions.

It is shown that QMOOD metrics model is highly effective in predicting software defects in both traditional and iterative (like agile) software development processes [49].

Since the QMOOD metrics are not limited to a specific range, it is difficult for the user to interpret their values. Therefore, we built a software quality benchmark dataset consisting of the quality metrics calculated for over 100 open-source and industrial software projects. Then, to summarize all six quality attributes, we defined a super metric called Total Quality Index (TQI) as the linear summation of the metrics.

Finally, we compared the quality metrics and TQI of a new project/file with the range of the benchmark and assigned a quality label (A, B, C, and D) based on the quartile of a value.

This method facilitates the analysis of quality reports and gives meaning to the metrics in terms of the quality level (low/high) of software compared to other standard projects.

*2) **Optimization using Refactoring**:* Finding a refactoring solution can be a challenging task since a huge search space requires to be explored. This search space is the outcome of the number of refactoring operations and the importance of their order and combination. To search this space, we employed an adaptation of the non-dominated sorting genetic

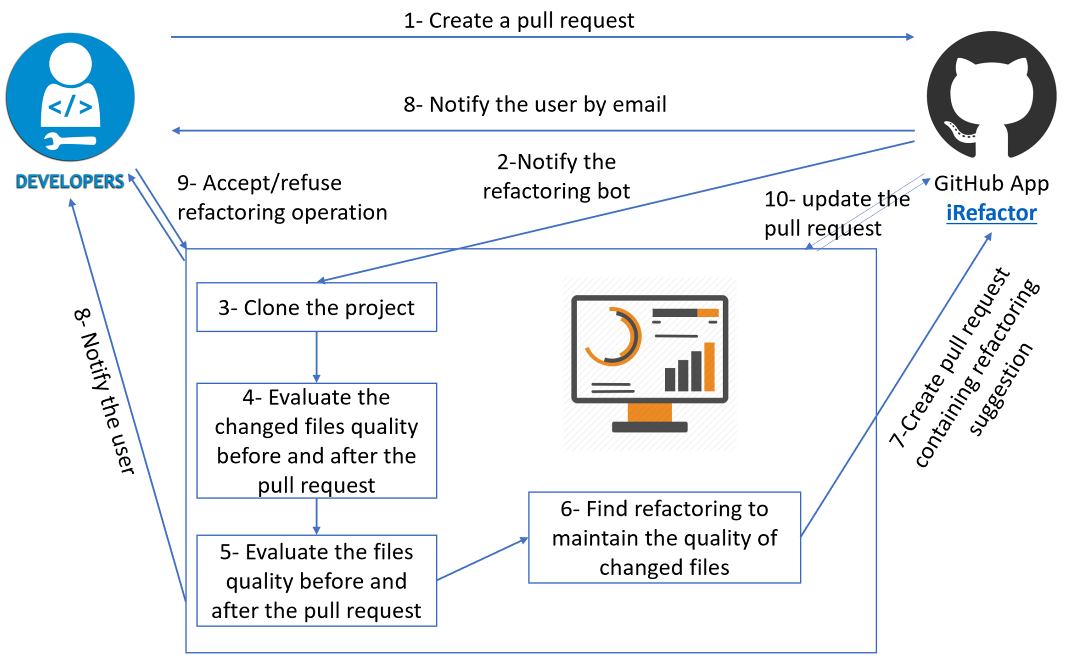
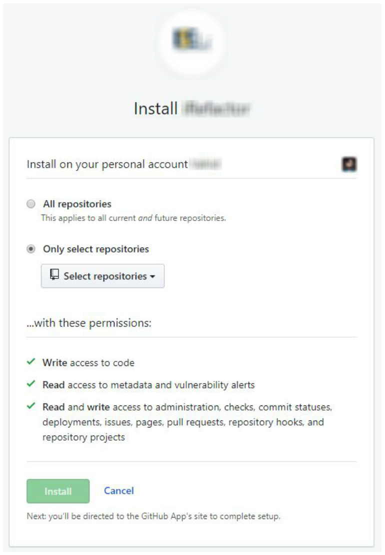

1- Create a pull request
8- Notify the user by email
DEVELOPERS
9- Accept/refuse refactoring operation
2-Notify the refactoring bot
10- update the pull request
GitHub App
iRefactor
8- Notify the user
3- Clone the project
4- Evaluate the changed files quality before and after the pull request
5- Evaluate the files quality before and after the pull request
6- Find refactoring to maintain the quality of changed files
7-Create pull request containing refactoring suggestion

Fig. 1. The overview of RefBot Pipeline


Install [redacted]

Install on your personal account [redacted]

All repositories
This applies to all current *and* future repositories.

[✓ Only select repositories]

Select repositories

...with these permissions:

✓ **Write** access to code
✓ **Read** access to metadata and vulnerability alerts
✓ **Read** and **write** access to administration, checks, commit statuses, deployments, issues, pages, pull requests, repository hooks, and repository projects

Install Cancel

Next: you'll be directed to the GitHub App's site to complete setup.

Fig. 2. Installing RefBot on a repository

algorithm (NSGA-II) [50] to discover a trade-off between multiple quality attributes.

NSGA-II is a multi-objective evolutionary algorithm operating on a population of candidate solutions that are evolved toward the Pareto-optimal solution set. NSGA-II uses an explicit diversity-preserving strategy together with an elite-preservation strategy. [50].

A refactoring solution is designed as a vector that consists of an ordered sequence of multiple refactoring operations. Each refactoring operation includes a refactoring action and its specific controlling parameters. The refactoring operations

TABLE I
QUALITY ATTRIBUTES AND THEIR COMPUTATION EQUATIONS.

Quality attributes	**Definition** / **Computation**
Reusability	A design with low coupling and high cohesion is easily reused by other designs.
	$0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	The degree of allowance of changes in the design.
	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	The degree of understanding and the easiness of learning the design implementation details.
	$0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	Classes with given functions that are publicly stated in interfaces to be used by others.
	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	Measurement of design's allowance to incorporate new functional requirements.
	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	Design efficiency in fulfilling the required functionality.
	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

TABLE II
LIST OF REFACTORING OPERATIONS INCLUDED IN REFBOT.

Refactoring	**Controlling Parameter**
Moving Features Between Objects	
Move Method	Source, Target, Method
Move Field	Source, Target, Attribute
Extract Class	Source, Target, Attributes, Methods
Organizing Data	
Encapsulate Field	Source, Attribute
Simplifying Method Calls	
Decrease Field Security	Source, Attribute
Decrease Method Security	Source, Method
Increase Field Security	Source, Attribute
Increase Method Security	Source, Method
Dealing with Generalization	
Pull Up Field	Source, Target, Attribute
Pull Up Method	Source, Target, Method
Push Down Field	Source, Target, Attribute
Push Down Method	Source, Target, Method
Extract SubClass	Source, Target, Attributes, Methods
Extract SuperClass	Source, Target, Attributes, Methods

considered in RefBot cover the most used operations selected from different categories: ”Moving features”, ”Data organizers”, ”Method calls simplifiers”, and ”Generalization modifiers”. These refactorings are listed in Table II. Refactoring operations are created or modified randomly during the population initialization or mutation. Also, the size of a solution vector which is the number of included refactoring operation is randomly selected between lower and upper bound values. Therefore, it is crucial to examine the feasibility of a solution using related pre-conditions and post-conditions [51]. These conditions ensure that the program will not break while the behaviour is preserved by the refactoring.

To evaluate a candidate refactoring solution, a fitness function is defined to estimate its goodness. In order to measure the impact of a refactoring solution on the software project, we utilized six QMOOD quality attributes. The relative change of each quality attribute after applying the refactoring solution to the software system is considered as the fitness function and is expressed as:

$$FitnessFunction_i = \frac{AQM_i^{after}(CC) - AQM_i^{before}(CC)}{AQM_i^{before}(CC)} \tag{1}$$

where AQM_i^{before} and AQM_i^{after} are the averages of the quality metric i before and after applying a refactoring solution over all changed classes CC, respectively.

By defining the fitness function in this way, we aim to find the solutions capable of improving the quality attributes of the pull request.

Additionally, we constraint the search process to the solutions in which at least a ”class” controlling parameter is in the set of changed files in the pull request. For this purpose, we modified a variation operator of the search algorithm called ”Selection Operator”. Variation operators help to navigate through the search space and to maintain a good diversity in the population. Parent selection is a crucial step that directly affects the convergence rate. We added the controlling parameter constraint to the selection process.

After the execution of the refactoring search algorithm is finished, the instruction of applying each refactoring operation is added to the related files as a distinctive marker format similar to the Git conflict marker. Finally, RefBot creates a new pull request to introduce the changes to the repository.

C. Developer’s Interaction

One of the main advantages of RefBot is to include the developer in the refactoring process loop. When the internal workflow of RefBot on a pull request is completed, the developer is notified by email and also via GitHub checks API in the same page of the pull request. These notifications contain a link to the report page of the pull request where the users can analyze the results and give feedback to the recommended refactorings.

There are three levels of reports generated for each pull request and provided for the user:

- *Solution Report:* contains the quality history of the pull request and the impact of the recommended solution on the changed files.
- *File Report:* includes the list of refactorings applied to the selected file and the detailed quality history and impact of refactoring.
- *Refactoring Report:* represents the instruction of a single refactoring and the high-level code abstraction of source and target classes which are transformed by the operation.

Analyzing these simple yet effective reports give the ability of swift detection of required improvements based on individual preferences.

The developer can interact with the refactoring results of RefBot with three actions. Each refactoring can be ”rejected”, ”applied with a code marker”, or ”applied automatically”.

By rejecting a refactoring, it is not considered in the pull request. Applying with a code marker adds the refactoring instruction as a marker inside the related files. Therefore, the developer can manually implement the required changes. Last, applying automatically, gives permission to RefBot to change and apply the refactorings to the source code itself.

The reason we have both manual and automated refactoring is that sometimes the developers prefer to take control of the refactoring process and the changes in the structure of their code either for the whole software or a specific set of important classes/files.

When the developer is satisfied with the feedback, he/she can update the previously created RefBot’s pull request.

RefBot can be combined with continuous integration tools like TravisCI, Jenkins, or CircleCI to identify the problems that may occur during the automated refactoring by running integration tests.

D. Configuration and Customization

RefBot is highly customizable in terms of setting its internal workflow parameters and execution management.

Sometimes a developer is not willing to be disturbed for every new pull request. Therefore, RefBot can be configured to monitor the repository at a specific time interval or even can be triggered manually for a specific pull request.

Furthermore, users can enable/disable different refactoring types and quality attributes. In this way, they can control the optimization process and limit the search to the refactoring operations they are willing to apply and to the quality attributes they prefer to improve.

Additional materials such as the default parameter settings for NSGA-II and video demo of RefBot can be found at this publication's web page [1].

E. Running Example

In this section, to illustrate the process of RefBot and its performance in refactoring a pull request, we provide a running example on a real open-source software system.

We considered a pull request from "atomix" software repository and manually triggered RefBot to process it. Figure 3 represents part of the file quality table in the solution report page, which is generated for the selected pull request. It shows the TQI grade for the changed files before and after creating the pull request alongside with the impact of the recommended refactoring solution on the quality. As an example, the quality of the second file is degraded from 4.05 (B) to 1.18 (C). The solution which RefBot found for the pull request contains seven refactoring operations applied to this file. These refactorings could improve the file quality to 5.72 (B).

The user can view the detailed report page for each file. The bar charts in the file report page are provided in Figure 4. It shows the quality changes after the pull request and the refactoring solution impact for each of the six quality attributes, individually. We can observe that the recommended refactoring solution improves 5 out of 6 quality attributes for the file compared to the pull request quality.

Another section in the file report page is shown in Figure 5. It lists the refactoring operations from the recommended solution which have a controlling parameter applied to the selected file. The developer can interact with this list and reject or apply (code mark/auto options are as a popup window) each of the refactorings.

Additionally, the developer can further investigate each of the refactorings by viewing the refactoring report page. Figure 6 represents the abstract code changes after applying the selected refactoring on the source and target classes. This report can facilitate the decision making of users and help them to understand the changes in the structure introduced by a specific refactoring.

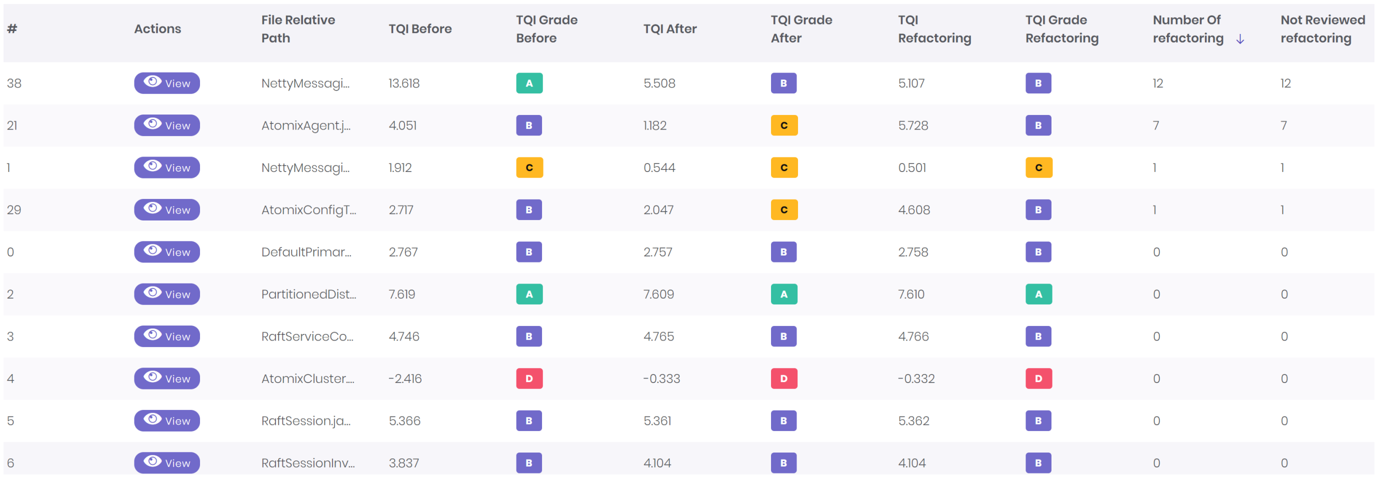
[1]https://sites.google.com/view/ase2019refbot

When a developer completes the interaction and analysis, the pull request is updated in the software repository, including the feedbacks on the refactorings. For any refactoring that applied as a code marker, the instructions are added to the top of the related files. Figure 7 depicts an example of the format of these markers.

IV. Validation

We define three categories of research questions to evaluate RefBot and compare it to state-of-the-art techniques for automated refactoring:

- **RQ1: Quality improvement.** *To what extent can our refactoring bot improve the quality of software systems as compared to existing automated refactoring techniques?* In RQ1, we use the internal quality attributes [48] and code smells as proxies to assess the quality improvement brought by the refactoring operations generated by the RefBot for a set of selected pull-requests on different systems. We compare the performance of our approach (MO-MFO) with two, state-of-the-art, refactoring techniques: Ouni [38] and JDeodorant [52]. Ouni [38] proposed an automated multi-objective refactoring formulation based on NSGA-II using an aggregation of quality metrics while reducing the number of refactorings. JDeodorant [52] is an Eclipse plugin able to detect code smells and automatically recommend refactorings to fix them. JDeodorant is not based on the use of heuristics search. As JDeodorant supports a lower number of refactoring types with respect to the ones we considered, we restrict our comparison with it to these refactorings. We have also limited the comparison to the changed files in the pull-requests.
- **RQ2: Refactoring meaningfulness.** *Are the refactoring recommendations produced by the RefBot meaningful from a developer's point of view? How do they compare with those generated by existing automated refactoring techniques?* Using antipatterns or internal quality indicators as proxies for code quality (as we do in RQ1) has substantial limitations. For this reason, in RQ1, we survey 25 developers asking for their opinion about the meaningfulness of the refactorings recommended by our technique and by the automated refactoring competitive technique [38]. In RQ2, we do not compare with JDeodorant since we preferred to focus on the most similar competitive technique in the literature to better study the advantages brought by the refactoring bot. The main substantial difference between RefBot and the approach by Ouni [38] is indeed the interactive and incremental approach of the refactoring bot to focus on pull-requests.
- **RQ3: Industrial validation.** *To what extent can RefBot support of refactoring in a real-world continuous integration setting?* We integrated a beta version of Refbot into a previously licensed refactoring tool and asked one of our industrial partners to use it for a limited period of 3 business days (with six developers involved) on their regular pull-request after installing the bot on their



#	Actions	File Relative Path	TQI Before	TQI Grade Before	TQI After	TQI Grade After	TQI Refactoring	TQI Grade Refactoring	Number Of refactoring ↓	Not Reviewed refactoring
38	View	NettyMessagi...	13.618	A	5.508	B	5.107	B	12	12
21	View	AtomixAgent.j...	4.051	B	1.182	C	5.728	B	7	7
1	View	NettyMessagi...	1.912	C	0.544	C	0.501	C	1	1
29	View	AtomixConfigT...	2.717	B	2.047	C	4.608	B	1	1
0	View	DefaultPrimar...	2.767	B	2.757	B	2.758	B	0	0
2	View	PartitionedDist...	7.619	A	7.609	A	7.610	A	0	0
3	View	RaftServiceCo...	4.746	B	4.765	B	4.766	B	0	0
4	View	AtomixCluster...	-2.416	D	-0.333	D	-0.332	D	0	0
5	View	RaftSession.ja...	5.366	B	5.361	B	5.362	B	0	0
6	View	RaftSessionInv...	3.837	B	4.104	B	4.104	B	0	0

Fig. 3. The quality table in solution report page

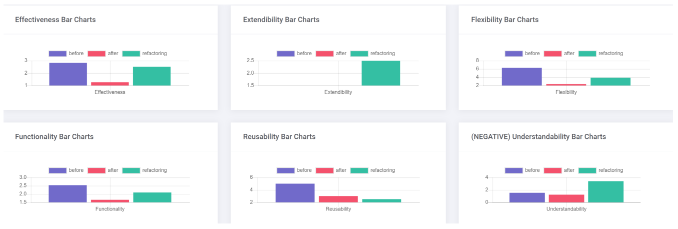
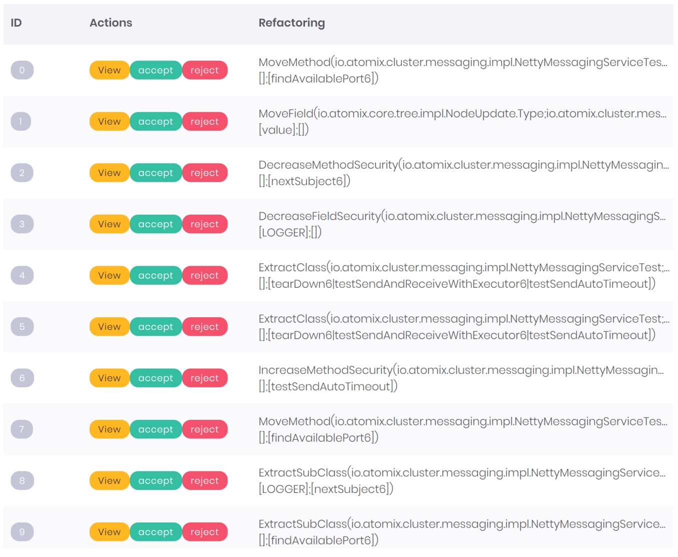


Fig. 4. The quality bar charts in file report page for all six quality attributes.

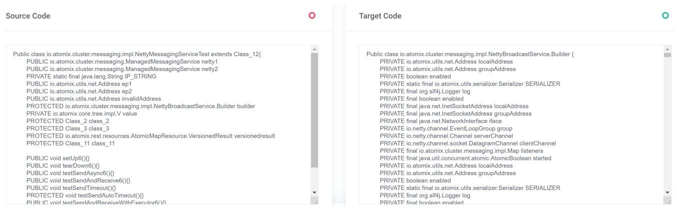


ID	Actions	Refactoring
0	View accept reject	MoveMethod(io.atomix.cluster.messaging.impl.NettyMessagingServiceTes... [];[findAvailablePort6])
1	View accept reject	MoveField(io.atomix.core.tree.impl.NodeUpdate.Type;io.atomix.cluster.mes... [value];[])
2	View accept reject	DecreaseMethodSecurity(io.atomix.cluster.messaging.impl.NettyMessagin... [];[nextSubject6])
3	View accept reject	DecreaseFieldSecurity(io.atomix.cluster.messaging.impl.NettyMessagingS... [LOGGER];[])
4	View accept reject	ExtractClass(io.atomix.cluster.messaging.impl.NettyMessagingServiceTest;... [];[tearDown6|testSendAndReceiveWithExecutor6|testSendAutoTimeout])
5	View accept reject	ExtractClass(io.atomix.cluster.messaging.impl.NettyMessagingServiceTest;... [];[tearDown6|testSendAndReceiveWithExecutor6|testSendAutoTimeout])
6	View accept reject	IncreaseMethodSecurity(io.atomix.cluster.messaging.impl.NettyMessagin... [];[testSendAutoTimeout])
7	View accept reject	MoveMethod(io.atomix.cluster.messaging.impl.NettyMessagingServiceTes... [];[findAvailablePort6])
8	View accept reject	ExtractSubClass(io.atomix.cluster.messaging.impl.NettyMessagingService... [LOGGER];[nextSubject6])
9	View accept reject	ExtractSubClass(io.atomix.cluster.messaging.impl.NettyMessagingService... [];[findAvailablePort6])

Fig. 5. The list of refactoring operations recommended for a single file.

repository. During this period, we checked the ability of RefBot to select relevant refactorings for the recent pull-requests introduced by the programmers during their daily activities.

The *context* of our study is represented by the seven systems in Table III. We selected these seven systems for our validation because they range from medium to large-size projects and



Source Code

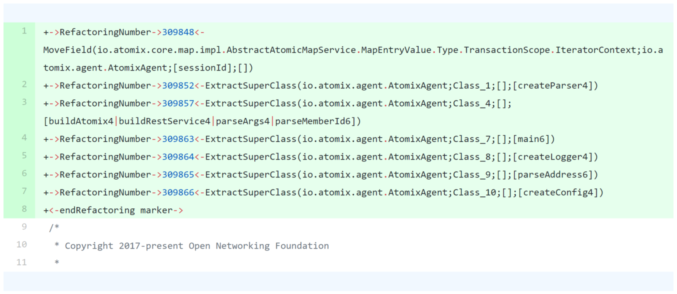
```
Public class io.atomix.cluster.messaging.impl.NettyMessagingServiceTest extends Class_12{
        PUBLIC io.atomix.cluster.messaging.ManagedMessagingService netty1
        PUBLIC io.atomix.cluster.messaging.ManagedMessagingService netty2
        PRIVATE static final java.lang.String IP_STRING
        PUBLIC io.atomix.utils.net.Address ep1
        PUBLIC io.atomix.utils.net.Address ep2
        PUBLIC io.atomix.utils.net.Address invalidAddress
        PROTECTED io.atomix.cluster.messaging.impl.NettyBroadcastService.Builder builder
        PRIVATE io.atomix.core.tree.impl.V value
        PROTECTED Class_2 class_2
        PROTECTED Class_3 class_3
        PROTECTED io.atomix.rest.resources.AtomicMapResource.VersionedResult versionedresult
        PROTECTED Class_11 class_11

        PUBLIC void setUp6(){}
        PUBLIC void tearDown6(){}
        PUBLIC void testSendAsync6(){}
        PUBLIC void testSendAndReceive6(){}
        PUBLIC void testSendTimeout(){}
        PROTECTED void testSendAutoTimeout(){}
        PUBLIC void testSendAndReceiveWithExecutor6(){}
```

Target Code

```
Public class io.atomix.cluster.messaging.impl.NettyBroadcastService.Builder {
        PRIVATE io.atomix.utils.net.Address localAddress
        PRIVATE io.atomix.utils.net.Address groupAddress
        PRIVATE boolean enabled
        PRIVATE static final io.atomix.utils.serializer.Serializer SERIALIZER
        PRIVATE final org.slf4j.Logger log
        PRIVATE final boolean enabled
        PRIVATE final java.net.InetSocketAddress localAddress
        PRIVATE final java.net.InetSocketAddress groupAddress
        PRIVATE final java.net.NetworkInterface iface
        PRIVATE io.netty.channel.EventLoopGroup group
        PRIVATE io.netty.channel.Channel serverChannel
        PRIVATE io.netty.channel.socket.DatagramChannel clientChannel
        PRIVATE final io.atomix.cluster.messaging.impl.Map listeners
        PRIVATE final java.util.concurrent.atomic.AtomicBoolean started
        PRIVATE io.atomix.utils.net.Address localAddress
        PRIVATE io.atomix.utils.net.Address groupAddress
        PRIVATE boolean enabled
        PRIVATE static final io.atomix.utils.serializer.Serializer SERIALIZER
        PRIVATE final org.slf4j.Logger log
        PRIVATE final boolean enabled
```

Fig. 6. The code abstraction of source and target classes after applying a specific refactoring.



```
+->RefactoringNumber->309848<-
MoveField(io.atomix.core.map.impl.AbstractAtomicMapService.MapEntryValue.Type.TransactionScope.IteratorContext;io.a
tomix.agent.AtomixAgent;[sessionId];[])
+->RefactoringNumber->309852<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_1;[];[createParser4])
+->RefactoringNumber->309857<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_4;[];
[buildAtomix4|buildRestService4|parseArgs4|parseMemberId6])
+->RefactoringNumber->309863<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_7;[];[main6])
+->RefactoringNumber->309864<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_8;[];[createLogger4])
+->RefactoringNumber->309865<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_9;[];[parseAddress6])
+->RefactoringNumber->309866<-ExtractSuperClass(io.atomix.agent.AtomixAgent;Class_10;[];[createConfig4])
+<-endRefactoring marker->
 /*
  * Copyright 2017-present Open Networking Foundation
  *
```

Fig. 7. The refactoring instructions related to a single file are added to the source code as a marker style.

TABLE III
STATISTICS OF THE STUDIED SYSTEMS.

System	Release	#classes	#smells	KLOC
Xerces-J	v2.7.0	991	91	240
JHotDraw	v7.5.1	585	25	21
JFreeChart	v1.0.18	521	72	170
GanttProject	v1.11.1	245	49	41
JDI	v5.8	638	88	247
Apache Ant	v1.8.2	1191	112	255
Rhino	v1.7.5	305	69	42

have been actively developed over the past 10 years. JDI[2] is an industrial project for which 6 of the developers involved in the JDI maintenance agreed to take part in our experiments.

Table III provides information about the size of the subject

[2]Company anonymized for double-blind.

systems (in terms of the number of classes and KLOC), and the number of code smells affecting them as detected with the rules defined in [53].

A. Data Collection

We present the data collection and analysis process grouped by research question category.

To address **RQ1**, we calculated *NF* as the percentage of code smells fixed by the refactoring solutions generated by the three considered approaches, over the total number of code smells which are affecting recent pull-requests of the subject systems. We selected the latest ten pull-requests for each of the open-source systems while a total of 8 pull-requests were opened during the three business days of the RefBot trial by our industrial partner. The detection of code smells before/after applying a refactoring solution was performed with the rules defined in [53]. The considered code smells are *Blob, Feature Envy (FE), Data Class (DC), Spaghetti Code (SC), Functional Decomposition (FD), and Shotgun Surgery (SS).*

Since the concept of code smell is very subjective (different developers may have different opinions on whether a code component is smelly or not) [54], we also use more objective metrics to assess the quality of the refactorings generated by the experimental approaches. We adopted the *G* metric based on *QMOOD* [48] that estimates the quality improvement of the system by comparing the quality before and after refactoring independently from the number of fixed design defects. Six quality factors are considered by *QMOOD*: reusability, flexibility, extendibility, functionality, understandability and effectiveness. All of them are formalized using a set of quality metrics. Hence, the total gain in quality *G* for each of the considered *QMOOD* quality attributes q_i before and after refactoring can be estimated as:

$$G = \frac{\sum_{i=1}^{6} G_{q_i}}{6} \quad where \quad G_{q_i} = q'_i - q_i \tag{2}$$

where q'_i and q_i represent the value of the quality attribute i respectively after and before refactoring.

To answer **RQ2** we asked 25 developers to evaluate the meaningfulness of the refactorings recommended by RefBot and by the approach of Ouni [38] for pull-requests on the seven subject systems. Before explaining the study design for RQ2, it is important to remember that both the experimental techniques generate output sequences of refactoring operations that make sense when considered together rather than when looking at them in isolation. However, it is not an option to ask a developer to assess the meaningfulness of all the refactoring operations generated for a given system. For this reason, we started by filtering for each system the sequences of refactoring operations impacting the files of a set of pull-requests to make a fair comparison between both tools. Then, the developers manually evaluated the outcomes of both tools for each pull-request.

Each participant was then asked to assess the meaningfulness of the sequences of refactoring operations. Since on six of the seven systems (all but JDI) we involved external developers (professional developers who did not take part in the development of the subject system), we made sure that each participant only evaluated refactoring sequences recommended by the two competitive techniques on one specific system (JHotDraw). The rationale for such a choice is that an external developer would need time to acquire a system's knowledge by inspecting its code, and we did not want participants to comprehend the code from four different systems since this would introduce a strong tiring effect in our study.

To answer **RQ3**, the six developers of the JDI project evaluated the refactoring sequences generated for that system, since here we wanted to exploit their experience as original developers of the system. They used RefBot, as a beta version tool, during a period of 3 days instead of a refactoring tool that we licensed to their company in the past. Our industrial partner was motivated to try out RefBot since they are interested in upgrading their current quality assessment tool to another one that can support DevOps like our RefBot. They also expressed a concern about the lack of customization and high configuration effort/training required by existing automated refactoring tools.

To support such a complex experimental design, we built a Java Web-app that automatically assigns the refactored pull-requests to be evaluated to the developers. The Web-app showed each participant one sequence of refactoring operations on a single page, providing the developer with (i) the list of refactorings (move method m_i to class C_j, then push down field f_k to subclass C_j,), (ii) the code of the classes impacted by the sequence of refactorings, and (iii) the complete code of the system subject of the refactoring with the description of the opened pull-request and the generated refactoring pull-request by the refactoring bot. The web page showing the refactoring sequence asked participants the question *Would you apply the proposed refactorings?* with a choice between *no* (the refactoring sequence is not meaningful), *maybe* (the refactoring sequence is meaningful, but the quality improvement it brings does not justify changing the code), or *yes* (the refactoring sequence is meaningful and should be implemented). Moreover, participants were allowed to leave a comment justifying their assessment (this was optional). The Web-app was also in charge of:

Balancing the evaluations per system. We made sure that each system received roughly the same number of participants evaluating the different refactored pull-requests (files associated/modified by these pull-requests) by the two approaches.

Keeping track of the time spent by participants in the evaluation of each refactoring sequence/refactoring pull-request. The time spent by participants was counted in seconds since the moment the Web-app showed the refactoring on the screen to the moment in which the participant submitted their assessment. This feature was done to remove participants from our data set who did not spend a reasonable amount of time in evaluating the refactorings. We consider less than 60

TABLE IV
PARTICIPANTS INVOLVED IN RQ2.

System	#Partic.	Avg. Prog. Experience	Avg. Java Experience	Avg. Refact. Exp.(1-5)
Xerces-J	4	11	9	4.0 (high)
JHotDraw	4	10	7	3.0 (medium)
JFreeChart	4	10	7	3.3 (medium)
GanttProject	4	9	8	3.5 (high)
JDI	6	14	12	4.5 (very high)
Apache Ant	3	9	7	3.7 (high)

seconds a reasonable threshold to remove noise (we removed all evaluation sessions in which the participant spent less than 60 seconds in analyzing a single refactoring sequence).

Collecting demographic information about the participants. We asked their programming experience (in years) overall and in Java, and a self-assessment of their refactoring experience (from very low to very high).

Table IV shows the participants involved in our study and how they were distributed in the evaluation of the refactoring sequences generated on the seven systems.

For the three days industrial validation, we integrated a routine in our RefBot to record all the actions of the 6 developers including the number of applied and rejected refactorings, number of selected test cases, the introduced code changes and commit messages.

B. Experimental Setting and Data Analysis

For each algorithm and each system, we performed a set of experiments using several population sizes: 50, 100, 200, and 300. Then, we specified the maximum chromosome length (maximum number of operations/test cases per solution). The resulting vector length is proportional to the number of refactorings that are considered, and the size of the program to refactor. Based on those considerations, the upper and lower bounds on the chromosome length were set to 10 and 350, respectively. The stopping criterion was set to 10,000 fitness evaluations for all algorithms to ensure fairness. In order to have significant results, for each couple (algorithm, system), we use the trial and error method [55] for parameter configuration.

Concerning RQ2, we report the percentage of refactoring sequences assessed with a *no*, *maybe*, or *yes* by developers for each treatment (RefBot and Ouni system [38]). Then, we discuss interesting comments left by developers when justifying their assessment.

C. Results

RQ1: Quality improvement. Figures 8 and 9 provide the percentage of fixed code smells (NF) and the quality gain (G) based on the *QMOOD* model, respectively. The average *NF* on the seven systems is 91% with peaks of ~96% for JHotDraw and GanttProject.

The recommended refactorings also improved the G metric values (Figure 9) of the seven systems. The average quality gain for the Rhino system was the highest among the seven systems with 0.43. The improvement in the quality gain shows that the recommended refactorings help to optimize different

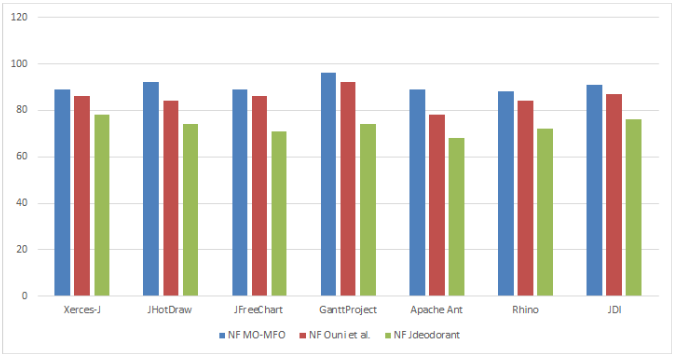


Fig. 8. Median percentage of fixed code smells (NF) on the different pull-requests of the seven systems.

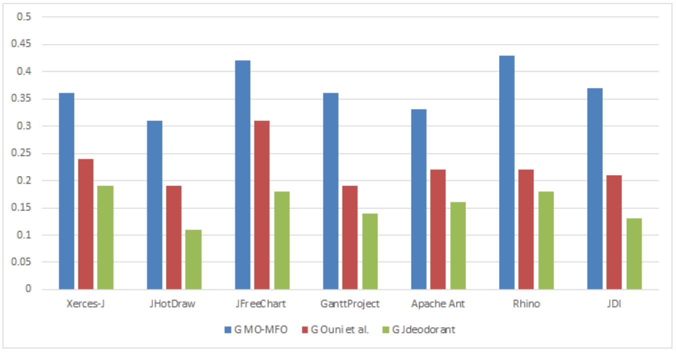


Fig. 9. Median quality gain (G) on the different pull-requests of the seven systems.

TABLE V
RQ2: WOULD YOU APPLY THE PROPOSED REFACTORINGS OF THE GENERATED REFACTORING PULL-REQUEST?

Approach	no	maybe	yes
RefBot	4/68 (5%)	11/68 (16%)	53/68 (77%)
Ouni [38]	29/83 (34%)	41/83 (49%)	13/83 (15%)

quality metrics. Besides, the performance of RefBot is superior as compared to the competitive refactoring techniques [38], [52], even though the difference in terms of fixed code smells is not that marked (Figure 8). This latter result is also due to the fact that RefBot does not only recommend refactoring operations aimed at removing code smells it also focuses on refactoring classes not affected by code smells but were changed during recent pull-requests. For example, in a manual investigation of the refactorings recommended by RefBot for JFreeChart, we found that 17 of the impacted classes do not exhibit any criticality as indicated by code smells and they were still improved in terms of quality attributes.

RQ2: Refactoring meaningfulness. Table V summarizes the manual refactoring evaluation results obtained from the 25 participants. Note that there is a slight deviation between the total number of refactorings evaluated by the two approaches (68 *vs* 83) since, as explained in Section IV, we did not consider for the data analysis the evaluations in which participants spent less than 60 seconds to assess the meaningfulness of the refactoring sequence under analysis and also the approach of Ouni et al. tends to generate much more refactorings on the analyzed files from the pull-requests.

The analysis of the quality by the Refactoring Bot improved the relevance of the recommended refactorings compared to the fully automated multi-objective approach. Indeed, the percentage of meaningful recommendations (the sum of the *maybe* and *yes* answers) is much better for RefBot comparing to Ouni et al. (94% for RefBot and 66% for Ouni). The percentage of refactorings that participants believe must be applied (*yes* answers) is significantly higher for Refbot as well (77% *vs* 15%).

By looking at the comments left by participants when justifying their assessment, four out of the six original developers of the JDI system highlighted in their comments for three refactoring sequences that they found the refactorings relevant because it is improving the modularity of a class that they frequently modify in all the most recent pull-requests. For example, one of the developers wrote in a comment: *"That is a very good recommendation, I spent days working on this class recently there, so I like this move method very much and extract sub-class. It will improve the reusability a lot as highlighted by the explanations of the bot"*. We found this comment as important qualitative evidence of the value of our refactoring bot in terms of analyzing the recently closed pull-requests to identify changed files and fix the identified quality issues in these files.

RQ3: Industry validation. Figures 8 and 9 summarize the results of deploying our RefBot during 3 business days to our industrial partner on the JDI repository. The six developers used the bot as part of their daily programming activities instead of a previously licensed refactoring tool. The tool was deployed as a Git app that connects automatically to a private GitHub repository whenever some code changes are introduced by the developers to check for refactorings and generate a new pull-request for the review of developers.

Overall, the achieved results confirm the effectiveness of our bot to generate efficient refactoring pull-requests. We found that the developers approved 9 out of 11 refactoring pull-requests generated by the bot during the three days. For the two remaining pull-requests, we found that a total of 7 out of 11 refactorings were approved. The achieved results confirm the basic intuition behind this work, showing that developers are more motivated to apply refactorings when the tool is easy to integrate within their development pipeline. The six developers also confirmed that they feel more comfortable in applying refactorings due to the high level of control proposed by the bot to review the generated pull-request which gives them more confidence and trust to the tool. This may explain the reason why a good number of recommended refactorings were applied.

V. Threats to Validity

Our refactoring bot mainly focuses on the recent pull-requests, but developers may have different priorities based on their current context. However, the developers can modify the configuration of our bot to focus on commits, branches, specific files or developers' contributions. Another internal threat is related to the used quality attributes since developers may want to express different preferences than QMOOD, or they want to tune them based on their needs or how critical is the code.

Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected a set of pull-requests when comparing with other techniques, but may perform better on other pull-requests where the quality of them are different.

External validity refers to the generalize-ability of our findings. We performed our experiments on open-source systems belonging to different domains, and one industrial project, by involving participants in the evaluations of the refactoring operations. However, we cannot assert that our results can be generalized to other applications, and other developers. Future replications of this study are necessary to confirm our findings.

VI. Conclusion

We presented a first attempt to propose an intelligent software refactoring bot, as GitHub app, that can submit a pull-request to refactor recent code changes. The salient feature of the proposed bot is that it incorporates interaction support, via our Web app, hence allowing developers to approve or modify or reject the applied code refactoring. The refactoring bot also provides support to explain why the refactorings are applied by quantifying the quality improvements. To evaluate the effectiveness of our technique, we applied it to four open-source and one industrial projects comparing it with state-of-the-art approaches. Our results show promising evidence on the usefulness of the proposed interactive refactoring bot. The participants highlighted the high usability of the bot in terms of easy integration with their development environments with the least configuration effort.

Future work will involve validating our technique with additional refactoring types, programming languages, quality issues and participation from practitioners to investigate the general applicability of the proposed methodology.

References

[1] S. A. Bohner and R. S. Arnold, *Software change impact analysis*. IEEE Computer Society Press Los Alamitos, 1996, vol. 6.

[2] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 535–546.

[3] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[4] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving usability of software refactoring tools," in *2007 Australian Software Engineering Conference (ASWEC'07)*. IEEE, 2007, pp. 307–318.

[5] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 1263–1270.

[6] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[7] J. Simmonds and T. Mens, "A comparison of software refactoring tools," *Programming Technology Lab*, 2002.

[8] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 5–18, 2011.

[9] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *Software Engineering, IEEE Transactions on*, vol. 40, no. 7, pp. 633–649, July 2014.

[10] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 552–576.

[11] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 211–221.

[12] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, pp. 126–139, 2004.

[13] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 233–243.

[14] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Faultbuster: An automatic code smell refactoring toolset," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 253–258.

[15] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, 2017.

[16] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

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

[18] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 17:1–17:45, 2015.

[19] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 464–474. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238217

[20] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, 2018.

[21] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, "Model transformation modularization as a many-objective optimization problem," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1009–1032, 2017.

[22] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[23] H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in *International Conference on Service-Oriented Computing*. Springer, Cham, 2016, pp. 352–368.

[24] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.

[25] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, no. 4, pp. 551–571, 2013.

[26] A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer, "Search-based detection of high-level model changes," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 212–221.

[27] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, "Search-based design defects detection by example," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, Berlin, Heidelberg, 2011, pp. 401–415.

[28] M. Kessentini, H. Sahraoui, and M. Boukadoum, "Example-based model-transformation testing," *Automated Software Engineering*, vol. 18, no. 2, pp. 199–224, 2011.

[29] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating transformation rules from examples for behavioral models," in *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*. ACM, 2010, p. 2.

[30] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," *Software Quality Journal*, vol. 25, no. 2, pp. 473–501, 2017.

[31] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, 2017.

[32] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer Berlin Heidelberg, 2014, pp. 387–419.

[33] M. O'Keeffe and M. Ó Cinnéide, "A stochastic approach to automated design improvement," in *International Conference on Principles and practice of programming in Java*. Computer Science Press, Inc., 2003, pp. 59–62.

[34] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1106–1113.

[35] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *International conference on Genetic and evolutionary computation*. ACM, 2006, pp. 1909–1916.

[36] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2011, pp. 81–90.

[37] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, pp. 221–230.

[38] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: an industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 23, 2016.

[39] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue, "Web service antipatterns detection using genetic programming," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 1351–1358.

[40] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*, vol. 23, no. 2, pp. 323–361, 2015.

[41] M. Kessentini, A. Bouchoucha, H. Sahraoui, and M. Boukadoum, "Example-based sequence diagrams to colored petri nets transformation using heuristic search," in *European Conference on Modelling Foundations and Applications*. Springer, Berlin, Heidelberg, 2010, pp. 156–172.

[42] C. Lebeuf, M.-A. Storey, and A. Zagalsky, "Software bots," *IEEE Software*, vol. 35, no. 1, pp. 18–23, 2018.

[43] ——, "How software developers mitigate collaboration friction with chatbots," *arXiv preprint arXiv:1702.07011*, 2017.

[44] M. WESSEL, B. M. DE SOUZA, I. STEINMACHER, I. S. WIESE, I. POLATO, A. P. CHAVES, and M. A. GEROSA, "The power of bots: Understanding bots in oss projects," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, pp. 1–19, 2018.

[45] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, "How to design a program repair bot?: insights from the repairnator project," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 95–104.

[46] V. Balachandran, "Fix-it: An extensible code auto-fix component in review bot," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013, pp. 167–172.

[47] M. Wyrich and J. Bogner, "Towards an autonomous bot for automatic source code refactoring."

[48] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[49] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," *IEEE Transactions on software Engineering*, vol. 33, no. 6, pp. 402–419, 2007.

[50] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[51] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[52] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1037–1039.

[53] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[54] 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 *30th IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 101–110.

[55] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1–10.