Reconciling the Past and the Present: An Empirical Study on the Application of Source Code Transformations to Automatically Rejuvenate Java Programs

Reno Dantas, Antônio Carvalho Júnior, Diego Marcílio, Luísa Fantin,
Uriel Silva, Walter Lucas, and Rodrigo Bonifácio
Computer Science Department
University of Brasília
Brasília, Brazil

***Abstract*—Software systems change frequently over time, either due to new business requirements or technology pressures. Programming languages evolve in a similar constant fashion, though when a language release introduces new programming constructs, older constructs and idioms might become obsolete. The coexistence between newer and older constructs leads to several problems, such as increased maintenance efforts and steeper learning curve for developers. In this paper we present a RASCAL Java transformation library that evolves legacy systems to use more recent programming language constructs (such as multi-catch and lambda expressions). In order to understand how relevant automatic software rejuvenation is, we submitted 2462 transformations to 40 open source projects via the GitHub pull request mechanism. Initial results show that simple transformations, for instance the introduction of the diamond operator, are more likely to be accepted than transformations that change the code substantially, such as refactoring enhanced for loops to the newer functional style.**

I. INTRODUCTION

Mainstream programming languages present a certain similarity to software: in order to keep them up to date, while attending to the *user needs* and technology pressures, both have to evolve through time. To cite a few examples, the C++ programming language has evolved substantially in the current decade, with new releases of the standard being published in 2011, 2014, and 2017 (still in a draft version). Since 2008, Python is available in two major releases (2.x and 3.x) that are backward incompatible, and developers still struggle to chose the right version to use when starting a new project. Java programming language has evolved as well, with some substantial improvements present in the Tiger (J2SE 5.0, 2004), Dolphin (Java SE 7, 2011), and Spider (Java SE 8, 2014) releases.

However, when a language evolves, new idioms and tools to solve recurrent problems also emerge. In situations where a legacy system is able to use new features of a language, it is common to find new idioms coexisting with older ones, which might hinder developers to understand and maintain a software [1]. To mitigate this problem, Overbey and Johnson suggest the use of refactoring tools to allow programming languages to evolve [1], and thus help to introduce new language constructs and idioms in legacy systems—a process named software *rejuvenation* [2]. Actually, this has become a research trend and several tools have been recently proposed [3, 4, 5, 6].

Research problem. To the best of our knowledge, the existing literature does not present any report about a large scale experiment involving the refactoring of real systems towards programming language evolution, which leads to the central research question we investigate here: Is it worth it to refactor a Java legacy system to use new programming language constructs and idioms? Answering this research question might help researchers and tool developers to better understand what kinds of transformations are welcome and what features refactoring tools should provide to support global rejuvenation efforts.

Paper contributions. The main contributions of this research work are two fold. First we present RJTL (Section II), a RASCAL library that implements a set of transformations to migrate Java legacy systems to use new programming language constructs (including source code transformations to introduce the Java *multi-catch* construct, the *diamond operator*, and *lambda expressions*). Second, we report the results of an empirical study (Sections III and IV) that applied 2462 source-code transformations in different open source projects using our RASCAL library. We discuss our findings in Section V and relate our work to the existing research literature in Section VI. Finally, we present some final remarks and future work in Section VII.

II. RJTL: A JAVA TRANSFORMATIONS LIBRARY

In this research we are implementing a set of RASCAL Java transformations (RJTL) to support the evolution of legacy systems,[1] towards the usage of more recent constructions of the language. Ideally, developers should keep the code updated with new programming language constructs [1], but many

[1]RJTL is available at https://goo.gl/jmosUT

factors prevent this from happening in a timely fashion. Our transformations automate this update task and target constructs which may not have been used due to different reasons, such as the code having been written in a prior version of Java or because the development team was not yet familiar with features recently introduced in the language.

RJTL includes as central piece a RASCAL module called `Driver`, which is responsible for acquiring the parsed Java files (as abstract syntax trees – ASTs) and applying transformations on a selection of them. It reads from a CSV file some project metadata, such as the project's name, the transformation to be applied, and in which percentage of the cases it should be performed. Note that only one transformation is chosen at a time. In addition, the transformation is not necessarily performed in each and every possible case, only in some fraction of them, stated by the percentage value. The reason for allowing only one transformation at a time, and a limited number of executions, is to enable us to apply the library and generate outputs (log files) with varying number of changed lines of code. This is an important facility, which allows us to measure how the size of the transformation may affect the perceived benefit from the developer's point of view.

The decision to use RASCAL as the tool for implementing our transformations was based on its primitives for performing static code analysis and manipulation using high-level programming constructs, such as traversing through ASTs using the Visitor pattern, or being able to pattern match concrete Java's syntax elements to a specific node in an AST. In addition, RASCAL has a strong similarity with Java, regarding its syntax as well as its runtime environment (JVM). It is also possible to execute Java code from a RASCAL program, which we envision to be useful for us in a near future.

At the time we started this research, there was only a RASCAL Java syntax definition for the *Tiger* version (Java 5). Therefore, we had to implement our own syntax definition for Java 8 from scratch. Although our syntax definition does not use all disambiguation features of RASCAL, currently it recognizes 99.7% of the Java code from our benchmarks. We have already implemented several transformations, including (a) Introduction of the *diamond operator*, (b) Introduction of the *switch-string* constructor, (c) Introduction of *variadic arguments*, (d) Introduction of the *multi-catch* constructor, and (e) Introduction of *lambda expressions*, which converts *anonymous inner classes* and *enriched for loops* to the Java functional style.

These transformations required 4229 lines of RASCAL code. In truth, some of the transformations (such as the introduction of the *diamond operator* and the *switch-string* constructor) were elegantly implemented using only AST transformations. In these situations, where we do not have to query the type system, RASCAL features are outstanding. However, to implement more complex transformations, such as converting anonymous inner classes into lambda expressions, we have to extract some simple facts from the type system—in order to check a number of preconditions [3]. In our experience, other tools (such as Eclipse JDT) provide more facilities to query the declared types of a Java project than RASCAL, particularly because RASCAL is a general purpose meta-programming language environment that is able to analyze programs in different languages. For this reason, we are working on a specific RASCAL infrastructure to help us extract facts from Java declared types in a more declarative, simplified, manner.

In the present paper we share our early findings with the application of RJTL and these transformations. To check their effectiveness, we run our library in a number of other open source projects and submit the generated output as pull requests. We experiment with large and small patches, ranging from multiple files changed to submissions containing only a few transformations, as discussed later in Section IV.

III. Study Settings

To investigate the general research question of this work (*Is it worth it to refactor a Java legacy system to use new programming language constructs and idioms?*) we conducted a first empirical study that consisted of: (a) applying RJTL transformations to open source projects, (b) submitting pull requests with the results of the transformations, and (c) evaluating the pull requests assessments from the projects' contributors. This way, we can answer the following research questions: *(RQ1) Do open source developers accept contributions for rejuvenating legacy systems?*, *(RQ2) What are the reasons that motivate open source developers to reject transformations for rejuvenating legacy systems?* and *(RQ3) Considering the current set of* RJTL *transformations, which ones are more likely to be accepted?*

The selected projects and the transformations we used to rejuvenate the legacy systems are the controlled variables of this study. In addition to our analysis, we measure the total number of transformations, the total number of lines of changed code, and the number of accepted, rejected, and ignored pull requests. To build our population of interest, in May 2017 we selected the 100 most popular Java open source systems from GitHub. We used the GitHub number of stars as measurement of popularity. From this initial population, we randomly selected 40 projects as the *targets* of our transformations and then we also randomly selected the types of transformations we would apply for each one of these target projects. Most of these steps were supported by Python scripts.

Before applying the transformations for a given project, we first clone its repository and *check-out* the most active branch of development. We then execute the specific build of the project (either using *Maven* or *Gradle* build systems), running all available tests. We considered this step necessary because we decided to only apply a transformations to a system after we had been able to successfully build it. This decision allows us to identify transformations that might actually break a system in specific scenarios. After that, we configure the input file for the RJTL `Driver` and run a set of RJTL transformations that has been randomly selected for a particular project. In the final step, we rebuild the system running all test cases. In the cases where we accidentally introduced a compilation error or a bug identified by a failed test case, we create a RJTL issue and checkout the related file, restoring it to the version prior to the transformation. In some cases, it was also necessary to fix some source code formatting issues. At first, this activity was performed manually. But, by the end of the research, we started to use the *google-java-format* tool to fix formatting issues only in the parts of a file that had been changed by a RJTL transformation.

We submitted the results of the code changes via the GitHub pull request mechanism, in order to understand the acceptance rate of our transformations. We followed two different strategies here. In the first one, we submitted all changes (often involving different types of transformations) as a single pull request; in the second, we submitted several pull requests to the same project. In summary, we collected data from 45 pull requests submitted to 40 open source projects. They correspond to 2462 transformations that changed 6399 lines of code in 1207 files. We collect most of the relevant data using the log files produced by RJTL. In addition, we use information present in the GitHub pull request to compute the number of lines that actually changed (that is, the total number of lines that have been added and removed in a pull request).

We discarded from our analysis wrong pull requests that we submitted. For instance, in a specific case we submitted a pull request after a merge operation, which in the end combined contributions from other developers in the pull request. We also discarded from our analysis pull requests with more than 500 changed files.

IV. Results

The results achieved from the pull requests were classified in three forms: Accepted, when the pull request has been merged; Ignored, when we did not receive any response yet; and Rejected, when the pull request was closed without merging. We further classified the Rejected status into rejections motivated because the target system should be compatible with previous versions of Java (Rej. Version) and rejections because the maintainers considered the results of the pull request inadequate (Rej. Negative Effect). Figure 1 summarizes the acceptance rate of the pull requests, according to the categories just introduced. Accepted pull requests corresponded to 44.4%; rejected pull requests corresponded to 53.3% (31% due to incompatible versions and 22.2% due to inadequate results); and ignored pull requests corresponded to 4.44% of our observations.

Regarding the accepted pull requests, the introduction of the *Multi Catch* construct and the *Diamond Operator* are the transformations with higher acceptance rate (80% and 55%, respectively). The other transformations present a significantly lower acceptance rate. For instance, only 23% of the pull requests that transform anonymous inner classes and enhanced for loops into lambda expressions have been accepted. In this study, we concentrate our analysis in these three transformations only, because we sent a few pull requests involving the other transformations supported by RJTL.

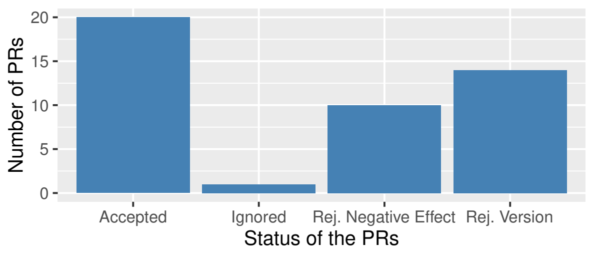

Fig. 1. Status of the pull requests.

Although we only submitted 5 pull requests involving the introduction of the *Multi-Catch* construct, four of them have been merged. They comprehend 56 transformations applied to 35 files, and leading to the removal of 130 lines of code. The introduction of the *Multi-Catch* construct brings the benefit of eliminating source code clones and reducing the number of lines of code. It is also a transformation that is relatively easy to understand the results. Listing 1 shows an example of a merged *Multi-Catch* transformation. In this example, we merged two `catch` blocks using the *multi-catch* construct. The higher acceptance rate of this transformation might have been motivated to both the benefits (clone and lines of code removal) and easy of understanding.

```
- } catch (IOException e) {
-     base64DecodeField.setText(e.getMessage());
-     base64DecodeField.setEnabled(false);
- } catch (IllegalArgumentException e) {
+ } catch(IOException | IllegalArgumentException e) {
      base64DecodeField.setText(e.getMessage());
      base64DecodeField.setEnabled(false);
  }
```

Listing 1. Example of a multi-catch transformation.

We created 20 pull requests of the *diamond operator* transformation (11 have been accepted), corresponding to 593 transformations applied to 256 files, and that changed 1188 lines of code. This is a simple transformation whose results are easy to understand, although it does not improve structural properties of the source code. In spite of that, more than 50% of the pull requests with this kind of transformation have been accepted. We believe that this acceptance rate is mostly because the results of this transformation are easy to understand (Listing 2 shows an example) and they lead to a slight simplification of the source code.

We submitted 16 pull requests of the *lambda expression* transformations. These transformations convert *anonymous inner classes* and *enhanced for loops* into lambda expressions, considering a set of preconditions [3]. Only three pull requests have been accepted, eight pull requests have been rejected because the target system (or library) should still support Java 6, and five pull requests have been rejected because the systems' maintainers considered them inadequate. The criticism related to the results of the transformations are mostly related to the maintainers not being convinced that the

```
- Map<String,Map<String,Integer>> _map = new HashMap<String,Map<String,Integer>>();
+ Map<String,Map<String,Integer>> _map = new HashMap<>();
```

Listing 2. Example of the transformation that introduces the *diamond operator*.

introduction of lambda expressions improve the source code. In addition, some maintainers actually believe that, in some situations, the use of lambda expressions might actually lead to a poor performance, when compared to a straight *enhanced for loop*.

After analyzing some of the lambda transformations that we submitted as pull requests, we agree that only part of the results lead to a code improvement (in terms of legibility), though some transformations actually lead to a code that is even harder to understand. It is important to note that other tools (such as LAMBDAFICATOR) would lead to the same result. That is, not all *enhanced for loops* are good candidates for applying the transformation, even in the situations they satisfy the constraints. For instance, Listing 3 shows an example we believe the transformation improves source code readability (using a *filter pattern*)—though this can be seen as a matter of opinion, affected by how familiar with the functional programming style the developer is. In contrast, Listing 4 shows an example of transformation that does not improve the source code. Both examples were submitted in a single, later rejected, pull request.

V. Discussion and Threats to Validity

We found some evidences that open source developers accept contributions for rejuvenating legacy systems, particularly when the benefits are clear to the developers. That is, a pull request that significantly modifies the source code might be hard to understand and the rejuvenation effort might not be integrated into the source code. Nevertheless, the maintainers of `google/binavi` accepted a pull request involving 135 transformations (87 files) of the *diamond operator*. Therefore, it seems to us that a global transformation involving simple transformations might be applied using a unique *pull request*; whereas complex transformations should be submitted using a single pull request per code change—since the consequences might not be easily understood. The size of our pull requests is a threat that might compromize the generalization of our results.

We found two main reasons that lead maintainers to reject external rejuvenation efforts. First, to our surprise, many systems and libraries considered in our study still use Java 6–although Java 8 was introduced in 2014. Moreover, many applications in production still run on top of older JVMs, and some Java libraries are also used to develop Android applications, whose platform does not full supports Java 8. This finding slightly contrast with existing works, which suggest the adoption of programming language features even before they are officially released [7].

Another threat to the validity of our work is the existence of *false-positives*, which was the second main reason that motivated developers to reject our pull-requests. Here a false-positive occurs when we transform a piece of code to introduce a new language feature and the result does not lead to a perceived source code improvement. This problem particularly occurred on the *lambda transformations*, which means that some research might be necessary to fully understand the situations in which it is worth it to transform *anonymous inner classes* and *enhanced for loops* into lambda expressions. Since we designed our transformations taking into account other tools [8], we believe that those tools might also introduce false-positives. Finally, we found that simple transformations (*multi-catch* and *diamond operator*) with clear benefits are more likely to be integrated into the code bases.

VI. Related Work

Although the work of Overbey and Johnson was the first to **advocate** the use of refactoring tools to help with software rejuvenation [1], Khatchadourian et al. present an automated approach for converting legacy Java systems to use enumerations [9]. More recently, several works have been proposed to deal with this problem, evolving Java systems to use new programming language constructs and idioms. For instance, LAMBDAFICATOR implements a set of refactorings to transform Java *anonymous inner classes* and *enriched for loops* into *lambda expressions* [3]. LAMBDAFICATOR considers a number of preconditions before applying a transformation, and we are actually taking them into account when implementing some of our transformations.

Tsantalis et al. detail an approach that automatically eliminates particular types of clone using *lambda expressions* [6]. According to the authors, these particular types of clone (that present behavioral differences) are hard to eliminate without *lambda expressions*. Khatchadourian and Masuhara present an approach that automatically converts the *skeletal implementation design pattern* to *default methods* in Java interfaces [4, 5]. Similar to our investigation, the authors of this mentioned work empirically evaluated the acceptance of the proposed approach by sending pull requests with the elimination of the skeletal pattern to 19 open source projects (four pull requests had been accepted).

VII. Final Remarks

In this paper we presented a RASCAL library (RJTL) for rejuvenating Java systems and the results of an empirical study we conducted to understand the challenges of applying transformations to evolve legacy systems to use new programming language constructs. We submitted 45 pull requests related to our transformations to several open source systems, with an acceptance rate around 44%. Our results reveal that simple transformations (such as introducing the *multi-catch* construct

```
- for (CoreLabel cl : m.originalSpan) {
-   if (interrogatives.contains(cl.word())) {
-      return true;
-   }
- }
- return false;
+ return m.originalSpan.stream().anyMatch(cl -> interrogatives.contains(cl.word()));
```

Listing 3. First example of the *lambda transformation*.

```
- for (int m : c) {
-  List<Integer> goldCluster = mentionToGold.get(m);
-  if (goldCluster != null) {
-     goldCounts.incrementCount(goldCluster);
-  }
- }
+ c.forEach(m -> {
+  List<Integer> goldCluster = mentionToGold.get(m);
+  if (goldCluster != null) {
+     goldCounts.incrementCount(goldCluster);
+  }
+ });
```

Listing 4. Second example of the *lambda transformation*.

and the *diamond operator*) are more likely to be accepted than transformations whose consequences are harder to understand. Currently we are investigating the situations in which refactoring a legacy Java system to introduce *lambda expressions* lead to source code improvements. Other than the nature and contents of the transformation itself, another relevant aspect is the moment that a transformation is introduced into the system. As our research noted, some pull requests were delayed or rejected because the code they changed had not been modified recently. That is, due to that piece of the software being considered stable, language-updating refactorings were not useful to the teams. Based on these findings, as future work, we intend to investigate if integrating the transformations as part of the regular development workflow would improve their perceived benefit and increase their acceptance. Our goal is to set up a tool that will watch a project repository for changes and automatically create pull requests for the modified code that contains obsolete constructs, moments after that code is committed.

References

[1] Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language: Refactoring tools allow programming languages to evolve. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 493–502, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0.

[2] A. Kumar, A. Sutton, and B. Stroustrup. Rejuvenating C++ programs through demacrofication. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 98–107, Sept 2012.

[3] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 543–553, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9.

[4] Raffi Khatchadourian and Hidehiko Masuhara. Automated refactoring of legacy java software to default methods. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 82–93, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2.

[5] Raffi Khatchadourian and Hidehiko Masuhara. Defaultification refactoring: A tool for automatically converting java methods to default. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 984–989, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-2684-9.

[6] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 60–70, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2.

[7] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790. ACM, 2014.

[8] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. Lambdaficator: from imperative to functional programming through automated refactoring. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1287–1290. IEEE Press, 2013.

[9] Raffi Khatchadourian, Jason Sawin, and Atanas Rountev. Automated refactoring of legacy java software to enumerated types. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, pages 224–233. IEEE, 2007.