The untold story of code refactoring customizations in practice

Daniel Oliveira*, Wesley K. G. Assunção*§, Alessandro Garcia*,
Ana Carla Bibiano*, Márcio Ribeiro†, Rohit Gheyi‡, Baldoino Fonseca†
*Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil
§Institute of Software Systems Engineering – Johannes Kepler University (JKU), Linz, Austria
†Computing Institute – Federal University of Alagoas (UFAL), Maceió, Brazil
‡Department of Computing and Systems – Federal University of Campina Grande (UFCG), Campina Grande, Brazil

***Abstract*—Refactoring is a common software maintenance practice. The literature defines standard code modifications for each refactoring type and popular IDEs provide refactoring tools aiming to support these standard modifications. However, previous studies indicated that developers either frequently avoid using these tools or end up modifying and even reversing the code automatically refactored by IDEs. Thus, developers are forced to manually apply refactorings, which is cumbersome and error-prone. This means that refactoring support may not be entirely aligned with practical needs. The improvement of tooling support for refactoring in practice requires understanding in what ways developers tailor refactoring modifications. To address this issue, we conduct an analysis of 1,162 refactorings composed of more than 100k program modifications from 13 software projects. The results reveal that developers recurrently apply patterns of additional modifications along with the standard ones, from here on called patterns of *customized refactorings*. For instance, we found customized refactorings in 80.77% of the *Move Method* instances observed in the software projects. We also investigated the features of refactoring tools in popular IDEs and observed that most of the customization patterns are not fully supported by them. Additionally, to understand the relevance of these customizations, we conducted a survey with 40 developers about the most frequent customization patterns we found. Developers confirm the relevance of customization patterns and agree that improvements in IDE's refactoring support are needed. These observations highlight that refactoring guidelines must be updated to reflect typical refactoring customizations. Also, IDE builders can use our results as a basis to enable a more flexible application of automated refactorings. For example, developers should be able to choose which method must handle exceptions when extracting an exception code into a new method.**

***Index Terms*—Refactoring, Custom Refactoring, Refactoring Tooling Support**

I. Introduction

Code refactoring is a widely used practice to promote program maintainability and other quality attributes [1]–[3]. Each code refactoring type is composed of a set of one or more modifications that aim at improving the program structure [1]. To support refactoring, the literature provides a set of standard modifications for each refactoring type [1], [4]. Despite the importance of refactorings as a strategy to keep internal software quality, developers remain reluctant on using IDE tools [5]–[7] to support these refactorings [8], [9]. In fact, developers believe these tools have limitations to practical use [9], [10]. These factors indicate that existing automated refactoring support may not be sufficient yet.

Previous studies observed that developers usually tailor the set of modifications associated with each refactoring type described in refactoring catalogs [10], [11]. These tailored modifications are named non-standard modifications and are part of refactoring customizations. A *customized refactoring* includes non-standard modifications that cohesively contribute to the realization of a refactoring type. Refactoring customization may be required to satisfy recurring developers' needs such as an adjustment to a local code structure, the removal of a certain poor structure, or even updating client methods [10].

We can observe some attempts to support developers in customizing refactorings. For instance, popular IDEs, such as Eclipse [5], NetBeans [6], and IntelliJ [7], allow developers to customize their refactorings through basic settings. However, previous studies [9]–[11] suggest that these settings are not aligned with the practice. Then, developers are induced to perform refactorings without the use of an IDE [12].

To the best of our knowledge, no study has analyzed in depth the typical customizations of refactoring types across multiple software projects. There are various open questions, including: (i) do developers indeed often customize their refactorings? (ii) what are the most common modifications related to each customized refactoring? (iii) how to improve IDEs to properly support the application of customized refactorings? The answers to such questions are necessary to guide tool builders in supporting the application of customized refactorings. Also, adequate guidelines and tooling support aligned with the practice may reduce developers' efforts.

Based on these limitations, we conducted a study by mining 13 open-source projects developed in Java. We focused our analysis on four common refactoring types, namely *Extract Method*, *Inline Method*, *Pull Up Method*, and *Move Method* [13], [14]. We identified, by using RefactoringMiner (RMiner) [4], 1,162 refactorings composed of more than 100k modifications. The analysis showed that standard modifications were often accompanied by recurring additional modifications, thereby showing that refactorings are indeed frequently customized by developers. We noticed in commits' comments that their authors mentioned the need for additional modifications to ensure the program's correctness [15], [16].

We found 42 patterns of customized refactoring that occurred in various refactorings of the same type. For instance, various patterns include a similar structure of exception handlers and related method calls, which go against certain IDE tooling mechanics. Developers would not be able to safely reuse these frequent customizations if they are not correctly predefined and supported by the IDE. Even worse, developers would have to: (i) find out by themselves the IDE's transformation is not adequate, (ii) ensure the program's correctness by avoiding unexpected behavior, and (iii) manually apply this non-trivial pattern in their code. Thus, understanding customized patterns is the basis for guiding in-depth investigations of customized refactorings and cataloging the scenarios in which these customized refactorings are applied.

Based on our findings, we evaluated the existing tooling support for applying frequent customized refactoring with widely used IDEs, namely Eclipse, IntelliJ, and NetBeans [12], [17]. We then listed and discussed 12 limitations that hamper the application of found patterns using such IDEs. For example, a key prevailing limitation is the lack of flexibility for developers to choose which method should handle exceptions when performing a *Extract Method*. IDEs make certain rigid choices on the behalf of developers, *e.g.*, inducing an exception handling location, which may lead to bugs in the refactored code. Therefore, our study findings shed light on how to improve refactoring guidelines and tool support. Our findings also provide insights on the design of recommenders for assisting developers in properly selecting code modifications of a refactoring that best match the contextual needs.

Finally, to corroborate the results of our previous analysis, we performed a survey with 40 developers. This survey was applied to investigate the relevance of the refactoring customization patterns and corresponding tool support. We found that 92.7% of the interviewed developers consider as important the addition of tooling support for customized refactoring in IDEs. Also, the interviewed developers provided additional arguments on the importance of these patterns.

II. Background and State of the Art

A. Refactoring Research and Practice

Code refactoring consists of applying modifications to code structures for enhancing program comprehensibility, maintainability, and other quality attributes [1], [10], [18]. The literature cataloged (*e.g.*, Fowlers' catalog [1]) various refactoring types and their *mechanics*. The mechanics for a refactoring type defines a set of standard code modifications, which guide developers in enhancing their code structure.

For our study, we focused on four popular refactoring types, chosen for two reasons. First, they have different scopes, *i.e.*, they cover both class-level modifications such as *Pull Up Method* and *Move Method*, and method-level ones such as *Extract Method* and *Inline Method*. Second, we focused on frequent, more complex, structural refactorings [18]–[20]. Simpler refactorings, *e.g.*, renaming, have less room for structural customization. Our selected refactoring types have wide scopes and allow a high number of customizations. Also, these four types of refactorings share structural similarities with other refactoring types, *e.g.*, *Move Method* moves a method from one class to another similarly to *Push Downs* and *Pull ups*.

TABLE I
Refactoring Details and Standard Refactoring Mechanics

Type	Description	Source	Target	Standard modification set
Extract Method	Create a method based on statements extracted from an existing method	Method where the extraction was performed	Extracted method	• Create the target method with code extracted from the source method • Update variables' references • Add in the source method's body a call to the target method
Inline Method	Incorporate the body of a method into an existing method	Method to be inlined	Method that inlined the source	• Replace each call to the source method with its method body • (Optional) Remove the source's method declaration
Pull Up Method	Move a method from a child class to its parent clas	Method in the subclass	Pulled up method in the superclass	• Create target method in the superclass and copy the source's method body • Remove from all subclasses the source's method declaration • If possible, change source methods calls, with call to the target method
Move Method	Move a method from one class to another class	Method to be moved	Method after being moved	• Create target method with a copy of the source's body method • If removed source's method: replace calls to target method • If did not remove source's method: add target call in source's body

Table I describes the refactorings with their corresponding source and target elements. These elements represent the main method modified, *i.e.*, the source, and the method produced after the refactoring, *i.e.*, the target. The standard modification sets are shown in the last column of Table I. These modifications are aligned with Fowler's and Opdyke's refactoring mechanics [1], [21], being the basis for the design of refactoring tools [4], [8], [22]–[25].

In several contexts, developers may need to customize the standard sets of modifications [11], adding or removing modifications from this standard set to tailor refactorings [11], [18]. These customizations make the application of refactorings more complex [10], [26]. To make it worse, existing refactoring tools (*e.g.*, [4], [27]–[31]) are mostly focused on only providing support for either the detection or the application of standard mechanics. In this way, there is a lack of tool support for these more complex refactorings, even though the interest of developers has been demonstrated in the literature [9], [10].

Previous studies investigated the motivations behind the refactoring application [32], [33]. Although these studies observed different reasons for performing refactorings, little is known about how refactorings are customized based on developers' needs or motivations. Some studies discuss the concept of *floss refactoring* [18], [34]. *Floss refactorings* are refactorings applied with other development activities, such as feature additions or bug fixes [18], [32]. The set of modifications in a floss refactoring may include some additional and non-standard modifications as part of the refactoring customization. However, these studies do not characterize which of these modifications are related to the refactoring itself. This characterization is necessary to properly support the application of customized refactoring through refactoring catalogs and tools.

A recent study investigated which modifications are combined with *Extract Method* [35]. However, this study focuses on only one refactoring type, besides investigating a limited scope of modifications. Also, the authors use a different AST diff with a higher granularity level. Finally, this study does not

investigate the support of these additional refactoring-related modifications on popular IDEs. Another study speculated the need for customized refactoring according to the development context [11]. However, this study did not empirically investigate the occurrences of refactoring customizations in those projects as well as their characteristics and support required.

In summary, the knowledge about customized refactoring is quite limited. It remains challenging and necessary to investigate: (i) in what ways refactorings are customized in practice, and (ii) whether and how to start improving IDEs refactoring tools to properly support refactoring customizations.

B. Refactoring Customization

Customized refactoring is a variation of the standard set of modifications defined for a type of refactoring [1]. This variation may occur due to the addition or even the removal of modifications from the standard set. Customization is often needed to tailor the refactoring to a program context. A customized refactoring includes only one or more non-standard modifications that have to be applied together with the standard ones to fully realize a refactoring. In other words, the non-standard modifications of a customized refactoring are also structural modifications required to implement a refactoring type. Non-standard and standard modifications of a customized refactoring interact and cohesively contribute to the realization of a refactoring type.

The conventional definition of refactoring assumes code behavior preservation [1]. This definition is in line with the notion of *root-canal refactoring* [18]. A *root-canal refactoring* occurs when the structural modifications of a refactoring are applied alone in a change and do not interact with co-occurring behavioral changes. However, certain recurring refactoring customizations may also be needed in the context of floss refactorings [18]; that is, the customized refactoring are applied in conjunction with other non-refactoring changes, such as feature addition. The customization may be required due to the interface of the refactored code with the new feature code.

The practical need for frequent floss refactorings does not make it possible for developers to always stick to the behavior-preserving aspect of the conventional definition of refactoring [10], [18], [36]. Recurring customizations may also exist in floss refactorings, and, as such, developers also need support to perform their frequent non-standard modifications for refactoring types in the context of floss refactoring. Thus, we classify the customized refactorings into two groups: (i) refactoring customizations that do not change the code behavior, *i.e, root-canal customizations*, and (ii) the refactoring customizations that change the code behavior, *i.e, floss customizations*.

Finally, we consider as *customization pattern* the recurring refactorings that satisfy both conditions: (i) they all have at least one structural modification that differs from the standard ones defined for a refactoring type; and (ii) this set of modifications, including the non-standard one(s), consistently occur together in multiple instances of that same refactoring type.

Listing 1 presents a refactoring customization, *e.g.*, a root-canal customization, of a *Move Method* that was applied to the Apache Tomcat project [37]. In this case, the developer moved a method called SETALLOWCASUALMP from the CONNECTOR class to the STANDARDCONTEXT class. This example has the following modifications: (i) a method was moved from one class to another class, and (ii) a method signature of this method was created on the interface (CONTEXT) of the target class. The first modification is part of the standard set of modifications for *Move Method* (see Table I). On the other hand, the second modification is an additional one that customizes the *Move Method*. This additional modification moved the SETALLOWCASUALMP method to the target class and made it an abstract method of the interface implemented by the target class. This additional modification is important to pass the test in the class (TESTSTANDARDCONTEXT) that calls the SETALLOWCASUALMP method directly from the Context interface. This example is an illustration of a customization pattern of the *Move Method* refactoring, in which the moved methods become part of an inherited interface.

Listing 1. Real Example of Customized *Move Method*

```
  public class Connector {
-   public void setAllowCasualMP(){ ...}    ...
  }
  public class StandardContext implements Context {
+   public void setAllowCasualMP(){ ...}    ...
  }
  public interface Context {
+   public void setAllowCasualMP(){ ...}    ...
  }
```

III. STUDY SETTINGS

As discussed in Section 2.1, little is known about refactoring customizations in practice. Thus, we investigated how developers apply and customize refactorings on their projects. We derived two research questions (RQs) that guided our study:

RQ_1: *In what ways are refactorings customized by developers?* RQ_1 aims at investigating how refactorings are applied in practice. We observe the most frequent root-canal and floss customizations by analyzing the modifications that compose each commit that includes a refactoring instance. This analysis enables us to identify and understand the most frequent customization patterns. We also discuss divergences between the modifications that compose the customized patterns and the standard mechanics of each refactoring type presented in Table I. As result, we present a catalog of customization patterns for each refactoring type. These patterns bring insights into how developers apply and customize refactorings.

RQ_2: *How to improve IDEs' automated refactoring tools to properly support customized refactorings?* Automated refactoring tools available in IDEs aim to support standard refactoring mechanics. Thus, they do not properly support customized refactorings both in the context of root-canal and floss refactorings. However, it is important that refactoring tools are in accordance with the practice; otherwise, developers may refuse to use them [9], [10]. In this way, RQ_2 aims at investigating what are the current IDE limitations and how their refactoring features should be improved to properly support the application of customized refactorings. For that,

we replicated, using popular IDE refactoring tools, customized patterns from the catalog obtained as a result of the RQ_1. The result of RQ_2 provides a list of identified limitations. This list is the basis to recommend how IDE tools can improve the support for developers to perform customized refactorings.

A. Study Steps

This section details the steps performed to build the dataset and perform the data analysis in our study. All the dataset-building steps and analyses were conducted by at least two authors and then discussed with other authors. In the presence of conflicting views, further discussion was required to converge. The dataset can be found on our website [38].

Step 1: Project Selection. We selected 13 active Java open-source projects of different sizes and domains. These projects are often used in previous studies of refactoring [2], [19], [20] given their frequency and diversity of refactorings. We took into account the stars count to prioritize popular projects [39]. We focused on open-source projects to facilitate the replication of our study. Finally, choosing Java projects allow us to use RMiner, a refactoring detection tool with high recall and precision, as discussed in Step 2. The selected projects were Elasticsearch-hadoop [40], Hystrix [41], Fresco [42], Achilles [43], Ikasan [44], ExoPlayer [45], Signal-Android [46], Netty [47], MaterialDrawer [48], Derby [49], Tomcat [50], HikariCP [51] and Material-dialogs [52]. The domains are: (i) data search and analysis; (ii) Android systems such as messaging applications and visual design; (iii) application server; and (iv) database construction. These projects have from 1,121 (doanduyhai Achilles) to 17,787 (Tomcat) commits.

Step 2: Refactoring Detection. We used RMiner [4] to detect refactorings performed in the selected projects. We chose RMiner because it is widely used in the literature [13], [19], [20], [53] and has high recall (87.2%) and precision (98%). With a high recall, the tool captures almost all refactoring instances performed in different contexts for each project. Thus, these instances may represent a variety of modifications used to customize refactorings for these diverse contexts.

We focused on four refactoring types (Table I), which are frequent in multiple projects [13], [19] and are present in popular IDEs. These refactorings constantly occur in a unique commit, affecting the same code fragment, known as composite refactorings [20]. Thus, to avoid modifications of composite refactoring instances to be erroneously considered as part of a unique refactoring type, we selected only commits with one detected refactoring. Lastly, any customization exclusively occurring with particular refactoring compositions (*e.g.*, *Extract Method* with *Move Method*) would be an addition to the customizations already present in our study; in other words, they would complement but not invalidate our results.

Step 3: Modification Detection. We used Eclipse's JDT 3.10 to collect the code modifications [54]. This library parses Java code into an Abstract Syntax Tree (AST). ASTs are widely used in the literature to detect refactorings [4], [55]. The Eclipse JDT is also used by the Gumtree framework [56], a popular framework used in literature to compare ASTs in Java [57], [58]. We used JDT directly because it provides Java language syntax information, allowing us to distinguish the same node type in different contexts. For example, using JDT we could observe whether the SIMPLE_NAME node is associated with a class variable, interface, class name or other Java tokens. These differences are relevant to detecting refactorings customizations and their patterns.

For each refactoring detected in Step 2, we collected the information before (v) and after (v_{+1}) the refactoring occurrence. We collected information related to the classes affected by the refactoring and their clients. We classified a class as affected by a refactoring when the modifications occurred within that class. For instance, an *Extract Method* has at least one affected class. On the other hand, a refactoring of the type *Pull Up Method* or *Move Method* has at least two affected classes, once a method is moved from one class to another one. Finally, we classified as a client of a class or method every other class or method that interacts with the client, *e.g.*, importing it and/or calling a method of the affected class. Once we have two subsequent versions of a class, the AST nodes are defined as $AST_v = \{node_i, node_{i+1}, ..., node_n\}$ where AST_v is the set of nodes belonging to the AST in version v. The set of added nodes to the source code between two subsequent versions is given by the resulting set of the difference between $AST_{v+1} - AST_v$. Similarly, the set of removed nodes from the source code is given by the difference of $AST_v - AST_{v+1}$.

Listing 2. Modifications between Two Subsequent Versions

```
+ public void clear() {
+   if (mAnimatedDrawableCachingBackend ! = null) {
+     mAnimatedDrawableCachingBackend.dropCaches();
+   }
+   ClosebleReference.closeSafely(mLastDrawnFrame);
+   mLastDrawnFrame = null;
+ }
  public void onInactive() {
-   if (mAnimatedDrawableCachingBackend ! = null) {
-     mAnimatedDrawableCachingBackend.dropCaches();
-   }
-   ClosebleReference.closeSafely(mLastDrawnFrame);
-   mLastDrawnFrame = null;
+   clear();
  }
```

Listing 2 illustrates the difference between two subsequent versions of a class from the Facebook Fresco project [59]. Table II presents a partial list of nodes obtained when analyzing the code in Listing 2, indicating the node type, scope, and whether the node was added or removed. We grouped the nodes based on semantic similarities of their modifications, creating coarse-grained categories, shown in Table III. For instance, the nodes related to conditional control, such as SWITCH_STATEMENT, CONDITIONAL_EXPRESSION and IF_STATEMENT, were grouped into the group *Conditional*. These categories enabled us to perform analysis and comparison focusing on the semantics of the modifications.

Step 4: Dataset Construction. The collected modifications of all refactoring instances might include modifications related to different software engineering activities, *e.g.*, feature

TABLE II
NODES DETECTED IN THE SUBSEQUENT VERSIONS

AST Node	Statement	Element	Status
METHOD_DECLARATION	Animated...Wrapper.clear()	Class	Added
IF_STATEMENT	mAnimated...Backend != null	clear()	Added
IF_STATEMENT	mAnimated...Backend != null	onInactive()	Removed
METHOD_INVOCATION	CloseableR...closeSafely(m...Frame) mAnimated...Backend.dropCaches()	onInactive()	Removed
METHOD_INVOCATION	CloseableR...closeSafely(m...Frame) mAnimated...Backend.dropCaches()	clear()	Added
METHOD_INVOCATION	clear();	onInactive()	Added

TABLE III
GROUPED MODIFICATIONS

Category	AST Nodes
Annotation	ANNOTATION_TYPE_DECLARATION, ANNOTATION_TYPE_MEMBER_DECLARATION, MEMBER_VALUE_PAIR, QUALIFIED_TYPE, NAME_QUALIFIED_TYPE, MARKER_ANNOTATION, NORMAL_ANNOTATION, SINGLE_MEMBER_ANNOTATION
Enum	ENUM_DECLARATION, ENUM_CONSTANT_DECLARATION
Method Declaration	FIELD_DECLARATION, METHOD_DECLARATION, INITIALIZER, LAMBDA_EXPRESSION, MODIFIER
Exception Handler	TRY_STATEMENT, CATCH_CLAUSE, THROW_STATEMENT, UNION_TYPE
Comments	JAVADOC, BLOCK_COMMENT, LINE_COMMENT, METHOD_REF, METHOD_REF_PARAMETER, MEMBER_REF, TAG_ELEMENT, TEXT_ELEMENT
Array Modifier	ARRAY_CREATION, ARRAY_INITIALIZER, ARRAY_ACCESS, ARRAY_TYPE, DIMENSION
Literal Modifier	BOOLEAN_LITERAL, CHARACTER_LITERAL, NULL_LITERAL, NUMBER_LITERAL, STRING_LITERAL, TYPE_LITERAL
Class Creation	CLASS_INSTANCE_CREATION, ANONYMOUS_CLASS_DECLARATION, TYPE_PARAMETER, CREATION_REFERENCE, TYPE_METHOD_REFERENCE
Conditional	CONDITIONAL_EXPRESSION, IF_STATEMENT, SWITCH_CASE, SWITCH_STATEMENT
Method Access	FIELD_ACCESS, METHOD_INVOCATION, SUPER_METHOD_REFERENCE SUPER_FIELD_ACCESS, SUPER_METHOD_INVOCATION, THIS_EXPRESSION, CONSTRUCTOR_INVOCATION, SUPER_CONSTRUCTOR_INVOCATION, EXPRESSION_METHOD_REFERENCE
Operator Expression	INFIX_EXPRESSION, POSTFIX_EXPRESSION, PREFIX_EXPRESSION, ASSIGNMENT
Cast	INSTANCEOF_EXPRESSION, CAST_EXPRESSION, INTERSECTION_TYPE
Variable Declaration	VARIABLE_DECLARATION_EXPRESSION, VARIABLE_DECLARATION_FRAGMENT, VARIABLE_DECLARATION_STATEMENT, SINGLE_VARIABLE_DECLARATION
Class Control	IMPORT_DECLARATION, PACKAGE_DECLARATION
Loop Flow Control	DO_STATEMENT, FOR_STATEMENT, BREAK_STATEMENT, WHILE_STATEMENT CONTINUE_STATEMENT, ENHANCED_FOR_STATEMENT
Type Modifier	SIMPLE_TYPE, TYPE_DECLARATION, TYPE_DECLARATION_STATEMENT, PRIMITIVE_TYPE, PARAMETERIZED_TYPE, WILDCARD_TYPE
Return Modifier	RETURN_STATEMENT

addition. Thus, in this step, we focused on filtering out non-refactoring modifications, which are modifications not related to the refactoring activity. Then, in order to facilitate the identification and removal of non-refactoring modifications, we split the modifications into two groups based on their code location: (i) the *internal modifications* that occurred within the source and target methods, and (ii) *external modifications* that occurred somewhere else.

The *internal modification* group includes the modifications within the source/target methods that are identified by the refactoring detection tool. These modifications are cataloged and represented on the RMiner detection rules [4]. For those modifications not detected by RMiner detection rules, we manually observed that different (non-)refactoring modifications depend on particular project aspects, such as design patterns and modularization. We concluded that these situational modifications were not frequent and, thus, did not follow any pattern. Therefore, we decided to discard these modifications from the internal group in our results.

The *external modification* group includes modifications performed externally to the source/target methods and that satisfy one of the following conditions: (i) the modifications are included in the standard mechanics of the analyzed refactoring type, *e.g.*, the creation of the target method during a *Extract Method*; or (ii) the modifications are related to additional software engineering activities, *e.g.*, feature addition, but interact with the source/target methods of the refactoring instance. We consider as interaction with the source/target any invocation of these methods in the source code of the refactoring modifications. For this later case, additional modifications are part of the refactoring activity once they only exist due to the structural change aimed by the refactoring. Those modifications typically determine "the interface" between the refactoring activity and the co-occurring software engineering activities. For instance, existing IDEs support developers in customizing an *Extract Method* refactoring by enabling them to qualify a method as public, protected or private, which is not a standard modification in the *Extract Method* definition, to bind the refactoring modifications with the non-refactoring modifications. This binding is made only due to the refactoring activity (and, therefore, is part of it) as a new method creation is an intrinsic goal of the refactoring. Making the method accessible is a compulsory modification to introduce method calls from client methods that compose the most frequent customizations.

For collecting external modifications, we applied a pattern matching algorithm. This algorithm visits all modifications related to a refactoring instance and collects the Java tokens, *e.g.*, variable and methods names. Then, tokens are filtered out based on whether there is a mention of the source/target method name such as the own method declaration, in the case of external modification (i); or method invocations, in the case of external modification (ii). When there is a mention of the source/target method name, the algorithm counts the number of parameters that were passed in the method invocation. In this way, we avoid misidentifying method invocations on refactoring instances that have more than one method with the same name. For this to be true, we also needed to remove from our dataset instances that have the source or target methods with the same name and an equal number of parameters.

In summary, we considered as refactoring modifications the modifications that satisfy one of the following conditions: (i) are explicitly listed in refactoring mechanics [1], [4], [21], or (ii) occurred externally to the source/target methods along with other (non-)modifications, but that also interacts with the source/target methods through a method invocation. Altogether, we found 1,162 refactoring instances and more than 100K modifications related to those refactorings. We found the following amount of instances and modifications for each refactoring type: (i) 856 instances and 77,306 modifications related to *Extract Method*, (ii) 174 instances and 14,126 modifications related to *Inline Method*, (iii) 78 instances and 5,856 modifications related to *Move Method*, and (iv) 54 instances and 3,734 modifications related to *Pull Up Method*. Additionally, we collected the commits' comments related to each refactoring instance. In this way, we could observe if developers mentioned any reference to the customizations.

Step 5: Survey with developers. To complement the results of our study, we conducted a survey to evaluate the relevance of the most frequent patterns (RQ_1) and the need for tooling support for such patterns (RQ_2) based on the developers' opin-

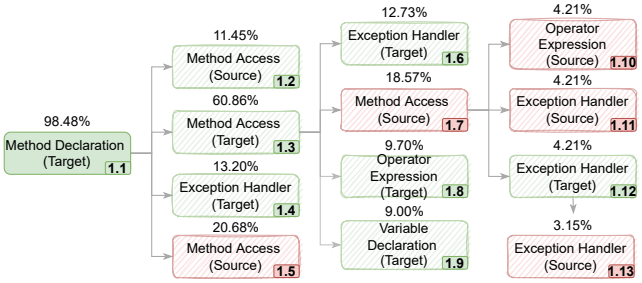

Fig. 1. Most Common Patterns for *Extract Method*

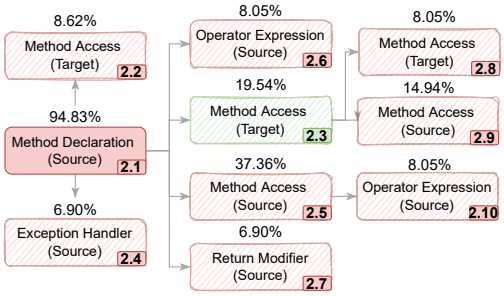

Fig. 2. Most Common Patterns for *Inline Method*

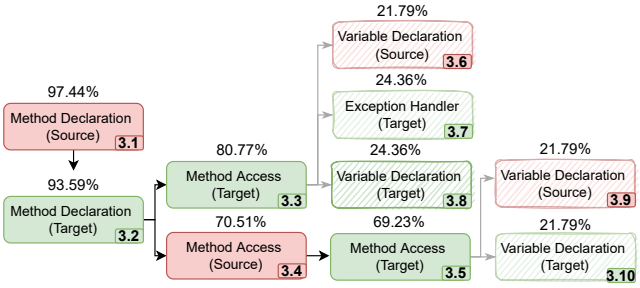

Fig. 3. Most Common Patterns for *Move Method*

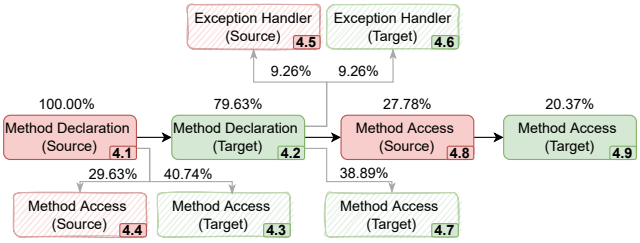

Fig. 4. Most Common Patterns for *Pull Up Method*

ions. We invited participants for the survey using convenience sampling, *i.e.*, developers who are easily accessible [60]. We invited developers using professional and academic mailing lists. The invitees were free to accept or not to participate and we have not provided any reward for participation. The participants answered questions regarding their experience with refactoring applications. In order to level the knowledge of participants, we introduced in detail the definition and mechanics of customized refactorings and described the current refactoring tooling support. For the survey questions, we selected the most frequent patterns that, together, include all modification categories observed in our catalog. Each question included a refactoring instance with the code fragments before and after the application of a selected customization pattern. Then, the developers were asked to analyze those code fragments and indicated whether the presented customization patterns are relevant and whether it would be necessary to have tooling support for their application. We made clear that the developers could ask for clarification during any part of the survey. The survey was composed of true/false and open-ended question types. The first question type allows us to precisely identify the interviewee's final decision regarding the customization support needed. The second question type allows us to understand which factors motivate their answer. The complete survey including its questions and answers can be found on our website [38].

IV. Results and Discussion

The following subsections present RQs' results and analysis.

A. Refactoring Customization in Practice

Figures 1 to 4 describe the most frequent customization patterns found for each refactoring type. These patterns are presented as a tree structure. Each node (box) in the figures represents a modification of a specific category performed to apply the customization. The nodes with dark background colors represent default modifications of Table I. The nodes with light background colors represent additional modifications. The green and red colors indicate whether the modification is an addition modification (+) or removal modification (-). The labels source (S) and target (T) indicate whether the modification interacts with the source or the target method. Each path, starting from the root node, characterizes a pattern of the respective refactoring type. The nodes belonging to the path are the modifications that compose the respective pattern. For example, we have pattern 1.3, for the *Extract Method* type, composed by Method Declaration (target) and Method Access (target). This pattern is expressed in text as {Method Declaration.T+, Method Access.T+}. Finally, we also present the percentage of occurrence of the respective pattern. We consider an occurrence if the pattern is included among all the modifications of a commit.

Developers constantly apply non-standard modifications. We observed that *Pull Up Method* was the only refactoring type in which a modification from the standard set (removal of the source method) occurred in 100% of their instances. On the other hand, modifications in the standard set (the addition of the target method), only occurred in 79.63% of the instances. This means that developers, in their customizations, might even occasionally not perform some standard modifications. Not applying a default modification does not necessarily imply a change in code behavior. In the case of the *Pull Up Method*, the absence of the creation of the target method is due to the existence of a method with the same signature. For *Pull Up Method*, the existent method is found in one parent class of the hierarchy. A similar behavior is also observed for the method declarations of Pattern 3.2 for *Move Method* and Pattern 1.1

for *Extract Method*. Thus, these patterns are supported in root-canal customization classification.

Patterns that simply added a single additional modification to the standard set occurred in 60.86% of *Extract Method* instances, 37.36% of *Inline Method* instances, 80.77% of *Move Method* instances, and 40.74% of *Pull Up Method* instances. More complex patterns, with at least two additional modifications, are less frequent. However, these patterns still occurred in over 10% of cases for all refactoring types. This is especially true for *Move Method*, which had patterns with five modifications that occurred in 21.79% of the instances. Thus, although the standard modification set is frequent, developers customize this set of modifications to include more possible modifications during the application of each refactoring type.

The most frequent additional modification among refactorings is *Method Access*. This modification indicates the addition or removal of a call to the source or target methods in the client method. The application of this modification unaccompanied by the replacement by the code of the source or target that had the call changed indicates a change in behavior, therefore a floss customization. Other additional modifications such as *Operator Expression* and *Variable Declaration* are related to code readability and thus do not affect the code behavior, being root-canal customizations. Finally, the modifications *Return Modifier* and *Exception Handler* do not exclusively indicate a change in behavior, since they tend to make the code more robust. Patterns with these two latter modifications can be floss or root-canal customizations, depending on the scenario.

Customization pattern modifications are similar for different refactoring types. Figure 1 presents the most frequent patterns for *Extract Method*. We observed that the addition of *Method Declaration* of the target method occurred in 98.48% of the *Extract Method* instances. In the remaining patterns, the refactoring mechanic differed from what is considered the default. In these cases, the developers extracted code statements and added them to an existing method. The addition of the extracted code elements to a method containing only the signature would not change the code behavior.

For Pattern 1.3, we observed the occurrence of a *Method Declaration* along with a *Method Access* in 60.86% of the cases. This means that client methods usually add a call to the target method after the extraction. This behavior reinforces the findings that developers extract fragments of code in order to be reused by new clients [13]. Also, for Pattern 1.7, the developers added a *Method Declaration* and *Method Access* to the target as well as removed *Method Access* to the source. This pattern suggests a swap between the source and target call. However, only in 11.45% of the instances, the developers switched the call from the source to a call to the target, indicating a possible code change behavior.

Figure 2 presents the most frequent patterns for *Inline Method*. We observed that, for 5.17% of the instances, developers preferred to keep the source method when applying *Inline Method* contrasting what is considered the standard. The results also indicate that in 37.36% of the *Inline Method* instances the client methods removed a call to the source method (Pattern 2.5), but only in 14.94% there was also the addition of a call to the target method (Pattern 2.9). Thus, similarly to Pattern 1.7 of *Extract Method*, the client methods that removed the call to the source method and did not replace that call to a call to the target method had their functionality reduced. This reduction in functionality may be related to unexpected code behavior.

Finally, most of the modifications are of the removal type and interact with the source method. This indicates that the clients of the source method needed to be adjusted to remove the interactions that they have with the source method. However, this adjustment is more complex than just removing calls to the source method. We can observe that the client methods also needed to adjust logical expressions (8.05%) and exception handling (6.9%).

Figure 3 presents the most frequent patterns for *Move Method*. We observed that most of the patterns tend to add calls to the target method (80.77%, Pattern 3.3) and remove calls to the source method (70.51%, Pattern 3.4). A manual validation indicated that in 57.69% of the instances of *Move Method*, developers added a target method call in client methods that did not call the source method before the refactoring (floss customization). We also noticed that developers performed more complex patterns that include exception handler and variable declaration, both occurred in 24.36% (Patterns 3.7 and 3.8) of the instances. Finally, we noticed that developers were often aware of the need to move the source method in order to improve exception handling. That is, by moving this method, new methods could take advantage of this handling, avoiding unexpected behavior [15], [16].

Figure 4 presents the most frequent patterns for *Pull Up Method*. We observed that the removal of the source method together with the addition of the target method occurred in 79.63% of the instances (Pattern 4.2). A manual validation indicated that in the cases without the addition of the target method, the superclass in the hierarchy already had a method with the same signature or an abstraction of it. Based on the commit's messages, developers chose to perform this customization to simplify future implementations and avoid code duplication [61], [62]. For that, they pulled up only the method's content into a superclass in order to create a standard implementation of this method. That way, each child class that implements this abstraction will no longer be forced to implement this method anymore. That is, this scenario required the customization of the *Pull Up Method* refactoring to fit in this different structure. This scenario is described by the commit's author [62], as follows:

'Move generic code to HttpOrSpdyChooser to simplify implementations. Motivation: HttpOrSpdyChooser can be simplified so the user not need to implement getProtocol(...) method.'

Similar to the other refactoring types, we also observed more complex patterns that also involve recurring exception handling modifications. In those cases, developers were concerned about ensuring the correct flow of the moved functionality, avoiding duplicate executions and unexpected behavior [62], [63]. When moving the handling to the super-

TABLE IV
LIST OF THE LIMITATIONS OF IDES' REFACTORING TOOLS

Id	Limitation
1	Modification only supported if occurred in source/target methods
2	It is not possible to remove source method invocation in client methods
3	It is not possible to remove target method invocation in client methods
4	It is not possible to add source method invocation in client methods
5	It is not possible to add target method invocation in client methods
6	There is no exception support for methods different than source and target ones
7	No exception handler is added if there is an exception error before the refactoring application
8	It is not possible to manage who should handle the exception
9	It is necessary that the extracted code is duplicated and the duplication recognized by the IDE *-Exclusive for Extract Method*
10	It is not possible to remove the modification without replacing it with the inlined method body *-Exclusive for Inline Method*
11	The swap of the call from source to target must occur in the same client *-Exclusive for Pull Up Method and Move Method*
12	It is mandatory to create the moved method, even if there is already a method with the same name in the destination class *-Exclusive for Pull Up Method and Move Method*

class, new implementations of this superclass will have the appropriate standard treatment that already handles possible exceptions, avoiding further problems for users, as mentioned by the commit's author [63].

In general, the standard set of modifications for each refactoring type occurred frequently. However, most of the refactoring instances involved additional modifications, especially method calls for both the target and source methods, and exception handling. These additional modifications turn the refactoring application more complex. The comments of the commits indicated that developers were constantly aware of the need for customization motivated mainly by the addition of new features and the improvement of program correctness, avoiding unexpected behavior in the code. These customizations are recurring and focus on adjusting the refactoring to specific scenarios, *e.g.*, move a method across hierarchies.

RQ_1: In what ways are refactorings customized by developers? Several recurring refactoring customizations are consistently present in multiple projects. The standard refactoring modifications (Table I) are far from being enough to address developers' needs. As such, developers frequently perform additional modifications, as those involving *Method Access* and *Exception Handler*, which extend or remove default refactoring modifications discussed in the literature [1]. Based on that, customized refactorings should be properly documented in order to better assist developers in performing code refactoring.

B. IDEs' Support for Customized Refactorings

In the previous RQ, we identified the most frequent patterns applied by developers when performing four refactoring types. In this RQ_2, we investigated how to improve the automated refactoring tools provided by the IDEs Eclipse, IntelliJ, and NetBeans to properly support the application of these patterns. We analyzed the source code of the instances of each pattern described in Figures 1 to 4. We minimally adapted the code to be reproducible in the IDEs' environment. Then, we manually invoked the IDEs' refactoring tools in order to reproduce the refactoring applied by the developer. For each IDE, we: (i) used the same code, (ii) selected the same statements, and (iii) applied the corresponding refactorings. Table IV lists the main limitations (identified from 1 to 12) that hinder the application of custom refactoring patterns when using existing IDEs' refactoring tools. The limitations 1 to 8 occurred in more than one refactoring type.

All IDEs share similar customization impediments. Tables V to VIII present the IDEs support for each pattern and associate them to the limitations shown in Table IV. We classified the IDEs' support into three categories: (i) *Full Support*, the refactoring tool is able to reproduce the pattern completely for all reproduced scenarios; (ii) *Partial Support*, the refactoring tool is able to reproduce the pattern completely only if some preconditions are met; and (iii) *No Support*, the refactoring tool is not able to reproduce the complete pattern in any circumstance. Once the IDEs refactoring tools follow the standard modifications, we observed that all IDEs had the same limitations. Thus, we used only one column to indicate the support category for all of them. The last column indicates the limitation *id*.

Table V presents the limitations for applying *Extract Method*. Except for Method Declaration.T+, all the other patterns have *No Support* or *Partial Support*. Limitation 2 is the most frequent among the *Extract Method* patterns, since most of the patterns include the removal of a *Method Access* in a client method. Limitations 2 to 5 refer to the addition, removal, or swap of methods calls to the source or target method.

Limitation 5 is also related to Pattern 3.3, in which developers add more calls to the target method in Move Methods. We observed this limitation, mainly, when developers apply *Move Methods* to support a feature addition. In the commit FC14CA31CB36 [64] of the Netty project, the developer moved the SAFEEXECUTE method from the SINGLETHREADEVENTEXECUTOR class to the ABSTRACTEVENTEXECUTOR class. The developer also called this moved method in other classes, mainly in classes that were created to support the NON STICKY EVENT EXECUTOR GROUP feature addition, as mentioned in the commit message [64]. A refactoring tool could mitigate this limitation by identifying when a *Move Method* is being applied in the feature addition context. For example, if the developer creates new classes after the *Move Method* application, then the tool can suggest the addition of a call to the previously moved method.

Limitations 6 to 8 affect the modification Exception Handler. For instance, if the selected statements for *Extract Method* throw an exception, the target method will throw this exception, even if the exception thrower is completely extracted. Thus, the IDEs do not allow developers to define which (source/target/client) method must handle that exception. This inflexibility forces all the methods that invoke the target to handle the exception themselves. Due to the lack of automated support, developers may not apply this exception handling correctly, causing an unintended behavior change.

Tooling support for each refactoring type has particular

TABLE V
LIMITATIONS OF *Extract Method* REFACTORING TOOLS

Patterns	IDEs' Support	Limitation Id
(1.1) Method Declaration.T+	Full support	
(1.2) Method Declaration.T+, Method Access.S+	No support	4
(1.3) Method Declaration.T+, Method Access.T+	Partial support	9
(1.4) Method Declaration.T+, Exception Handler.T+	Partial support	6,7,8
(1.5) Method Declaration.T+, Method Access.S-	No support	2
(1.6) Method Declaration.T+, Method Access.T+, Exception Handler.T+	Partial support	6,7,8,9
(1.7) Method Declaration.T+, Method Access.T+, Method Access.S-	No support	2,9
(1.8) Method Declaration.T+, Method Access.T+, Operator expression.T+	Partial support	1 (Operator Exp.),9
(1.9) Method Declaration.T+, Method Access.T+, Variable Declaration.T+	No support	1 (Variable Decla.),9
(1.10) Method Declaration.T+, Method Access.T+, Method Access.S-, Operator Expression.S-	No support	1 (Operator Exp.),2,9

TABLE VI
LIMITATIONS OF *Inline Method* REFACTORING TOOLS

Patterns	IDEs' Support	Limitation Id
(2.1) Method Declaration.S-	Full support	
(2.2) Method Declaration.S-, Method Access.T-	No support	3
(2.3) Method Declaration.S-, Method Access.T+	No support	5
(2.4) Method Declaration.S-, Exception Handler.S-	Partial support	6,7,8
(2.5) Method Declaration.S-, Method Access.S-	Partial support	10 (Method Access)
(2.6) Method Declaration.S-, Operator expression.S-	Partial support	10 (Operator Exp.)
(2.7) Method Declaration.S-, Return modifier.S-	Partial support	10 (Return modifier)
(2.8) Method Declaration.S-, Method Access.T+, Method Access.T-	No support	3,5
(2.9) Method Declaration.S-, Method Access.T+, Method Access.S-	No support	5,10
(2.10) Method Declaration.S-, Method Access.S-, Operator expression.S-	Partial support	10 (Operator Exp.), 10 (Method Access)

TABLE VII
LIMITATIONS OF *Move Method* REFACTORING TOOLS

Patterns	IDEs' Support	Limitation Id
(3.1) Method Declaration.S-	No support	12
(3.2) Method Declaration.S-, Method Declaration.T+	Full support	
(3.3) Method Declaration.S-, Method Declaration.T+, Method Access.T+	No support	5
(3.4) Method Declaration.S-, Method Declaration.T+, Method Access.S-	No support	2
(3.5) Method Declaration.S-, Method Declaration.T+, Method Access.S-, Method Access.T+	Partial support	11
(3.6) Method Declaration.S-, Method Declaration.T+, Method Access.T+, Variable declaration.S-	No support	1 (Variable Decla.), 5
(3.7) Method Declaration.S-, Method Declaration.T+, Method Access.T+, Exception Handler.T+	No support	5,6,7,8
(3.8) Method Declaration.S-, Method Declaration.T+, Method Access.T+, Variable declaration.T+	No support	1 (Variable Decla.),5
(3.9) Method Declaration.S-, Method Declaration.T+, Method Access.S-, Method Access.T+, Variable declaration.S-	No support	1 (Variable Decla.),11
(3.10) Method Declaration.S-,Method Declaration.T+, Method Access.S-, Method Access.T+, Variable declaration.T+	No support	1 (Variable Decla.),11

limitations. IDEs' refactoring tools have the same limitations, as discussed for *Extract Method*, for the remaining refactoring types. However, there are some particularities for each refactoring type. For *Extract Method*, we observed the exclusive Limitation 9. This limitation indicates that it is not possible to manually choose two similar or equal fragments of code to be extracted in a new method. In this way, developers depend on the tool to consider the codes as duplicates, otherwise, developers will need to perform the extraction manually.

For *Inline Method* (Table VI), we have the exclusive Limitation 10. In this refactoring, developers can choose to replace the call to the source method with the body of the source method. However, the refactoring tool does not let the developer only remove the call to the source method or replace the call to the source method with a call to the target method, both modifications are often applied. Therefore, developers are forced to: (i) make these not supported modifications manually or (ii) apply the refactoring as suggested by the tool and then remove manually some modifications applied. In both situations, because of the manual step, more effort is needed. This limitation increases the misalignment between refactoring tools and custom refactorings, increasing tool misuse [9], [10].

Limitation 11, exclusive for both *Move Method* (Table VII) and *Pull Up Method* (Table VIII), states that the refactoring tool allows developers to exchange a call to the source method for a call to the target. However, it does not allow only the addition of a call to the target method or only the removal of a call to the source method. For instance, developers may choose to call the target method on methods that did not call the source before refactoring because these methods did not have access to the source method or are in an inappropriate place. Inappropriate places are one of the main reasons why developers apply the *Move method* [13].

Finally, Limitation 12 is also exclusive for *Move Method* and *Pull Up Method*. This limitation indicates that it is not possible to move only the method content to a method with the same signature in the destination class. Thus, developers are forced to: (i) apply these refactoring manually, or (ii) force the method to be moved, leaving the destination class with two methods with the same signature.

We believe that the current tools are helpful for supporting refactoring activities. However, as hypothesized, these tools are not able to properly support the customizations performed by developers due to several limitations. In this way, developers are forced to manually apply the modification set of a customized refactoring either partially or completely, which is cumbersome and error-prone [10]. In general, the IDEs' refactoring tools present similar behavior. These tools do not allow users to change the modification set of a refactoring; that is, adding modifications besides those predefined by the IDE for each refactoring type or even removing a predefined one. We agree that IDEs should prioritize supporting code behavior preservation as default. However, even modifications that are not supposed to change code behavior, such as *Variable Declaration* and *Operator Expression*, are not properly supported.

RQ_2: How to improve IDEs' automated refactoring tools to properly support customized refactorings? Refactoring tools should make the configuration of refactoring modifications more flexible, allowing developers to adjust it based on their needs [8]. Existing tools would better adhere to developers' needs if they were designed to (i) support a comprehensive catalog of a mutable set of code modifications; (ii) have a configuration that allows developers to handle the clients that will be affected by the refactoring; (iii) allow developers to choose which element(s) should handle possible exceptions; and (iv) allow developers to choose between creating new methods or using existing ones.

TABLE VIII
Limitations of Pull Up Method Refactoring Tools

Patterns	IDEs' Support	Limitation Id
(4.1) Method Declaration.S-	Full support	
(4.2) Method Declaration.S-, Method Declaration.T+	Full support	
(4.3) Method Declaration.S-, Method Access.T+	No support	5
(4.4) Method Declaration.S-, Method Access.S-	No support	2
(4.5) Method Declaration.S-, Method Declaration.T+, Exception Handler.S-	Partial support	6,7,8
(4.6) Method Declaration.S-, Method Declaration.T+, Exception Handler.T+	Partial support	6,7,8
(4.7) Method Declaration.S-, Method Declaration.T+, Method Access.T+	No support	5
(4.8) Method Declaration.S-, Method Declaration.T+, Method Access.S-	No support	2
(4.9) Method Declaration.S-,Method Declaration.T+, Method Access.S-, Method Access.T+	Partial support	11

C. Developers' Opinion About Customization Refactoring

In the last step of our study, we conducted a survey to enrich RQs' results taking into account developers' opinions. All the survey details and results, including those not covered here, are presented on our study website [38]. Altogether the survey was answered by 40 developers. We observed that most of the respondents are familiar with the refactoring application. The majority of respondents (70.8%) declared themselves quite experienced with refactoring, performing refactoring constantly, whereas the remaining indicated applying refactorings periodically. Among the respondents, Eclipse is the most used IDE with 68.2% of them using it, followed by IntelliJ and Netbeans with 46.3% and 14.6%, respectively. Notably, 43% of them also indicated using multiple IDEs.

Respondents agree with the relevance of customization patterns. Survey answers indicate that the majority of the respondents agree with the relevance and support for customization patterns. Their answers were positive for all the types of non-standard refactoring modifications covered in the survey. For instance, the survey revealed 92.7% of agreement concerning both the relevance and the support needed for patterns that include addition and removal of *Method Access*. Interestingly, this modification category is present in the most frequent customization patterns. Also, this category is responsible to allow developers to select which method should access the source and target methods after the refactoring; this issue is related to the IDE Limitations 1 to 5 (Section IV.B).

With an agreement of 87.8%, the respondents also mentioned the importance of supporting customizations for *Method Declaration*. They agreed that developers should be in charge of deciding whether the method should be entirely (including its declaration) or partially moved. Regarding code exceptions, 75.6% of the respondents agreed that developers should be also given the flexibility of selecting where *Exception Handler* is introduced; this issue is associated with Limitations 6 to 8. Lastly, the respondents also pointed out the importance of tool assistance for refactoring customizations involving *Variable Declaration*, *Return Modifier* and *Operator Expression* with agreement of 70.7%, 65.9%, and 63.4%, respectively.

Customization assistance: spontaneously mentioned positive and negative factors. Finally, we also asked the respondents to openly justify their answers with free text by explaining which factors motivate tooling support for customization patterns. We manually categorized and grouped their answers into positive (*i.e.*, motivating) factors and negative (*i.e.*, demotivating) factors emerging from their answers. These factors are listed in Table IX with the their corresponding percentages of explicit mentions from developers.

TABLE IX
FACTORS MOTIVATING REFACTORING CUSTOMIZATION SUPPORT

Positive factors		**Negative factors**	
Awareness of side effects	44%	Own refactoring style	26%
Less error prone	41%	Low relevance	22%
Good coding practice	30%	Simplicity of modification	11%
Awareness of refactoring alternatives	26%	May not be a refactoring	11%
Behavior preservation	19%	Manual preference	11%

We observed that positive factors were much more frequently mentioned than negative ones. Most importantly, the majority of the negative factors have to do with personal preferences or uncertainties of the respondents, including: (i) freely follow their specific programming styles (26%); (ii) preference to apply customizations manually (11%); and (iii) not able to determine if one of the customization patterns (addressed explicitly in the survey) was indeed a refactoring (11%). There were a few cases of respondents that concerning one particular case of customization pattern: (i) was too simple (11%) to be supported by the IDE, or (ii) could not tell whether it was relevant to software maintainability (22%).

Developers argue that explicit customization support would improve code quality and correctness. They reported that IDE assistance for refactoring customization would improve their awareness with respect to bug proneness (41%), guarantees of behavior preservation (19%) and other possible side effects (44%) as well as adherence to good coding practices (30%). Finally, some respondents also found interesting the possibility of they becoming aware of multiple refactoring configuration alternatives (26%).

D. Actionable results

Until now, we discussed the practical occurrence of customized refactoring, the IDE limitations, and the developers' opinion regarding the need for refactoring customization support. Here, we discuss how to incorporate our findings into tooling support. A direct way is by integrating it into a semi-automated strategy of stepwise refactoring [65]. In this strategy, each refactoring modification is a step selected (or approved) by the developer, through which she/he can visualize, understand, and decide about each step. A graphical interface can support these steps by displaying the known alternatives of customizations, *e.g.*, our results shown in Figures 1-4, which can either be selected or adjusted based on developers' preferences, *e.g.*, following their code styles or team quality standards. Developers can also save their own performed customizations per refactoring type for later reuse.

The stepwise strategy is aligned with developers' expectations observed in our survey (Table IX), including (i) their "awareness of refactoring side effects" by tracking each of the refactoring modifications and their code effects; (ii) "reduced

error-proneness", allowing the developer to reason about the impact of each modification individually on the behavior of the code; and (iii) "awareness of refactoring alternatives", as the modifications are progressively shown to the developer depending on their previously-selected options, *e.g.*, following a path in the refactoring trees of Figures 1 to 4. Stepwise customized refactoring favors those developers requiring full control and predictability [9], [66] of customized refactorings ((i) and (ii) above) as they decide on the refactoring application step by step, thereby making them feel more confident using tooling support. This strategy is also aligned with recent and emerging proposals for step-wise refactoring in a range of different contexts, *e.g.*, [11], [65].

V. Threats to Validity

We describe here the threats to validity and their mitigation.

Internal and Construct Validity. RMiner [13], [53] may yield false positives and false negatives. It has an effectiveness of 87.2% for recall and 98% for precision [4], which is the best effectiveness among detection tools. To alleviate this threat, we manually inspected some instances of our database. Although we are currently analyzing refactorings detected only by RMiner, it is possible to observe that this tool has detection rules quite flexible, allowing several customizations [4].

RMiner detects 15 types of refactorings in version 1.0 [4], but we analyzed only four types of refactorings. Although these four refactorings may not fully embrace all forms of refactoring customizations, they have been frequently applied [13], [14]. Also, these refactorings affect the program structure differently at method-level and class-level. For instance, *Extract Method* is a method-level refactoring, affecting directly one class. Different from *Extract Method*, *Move Methods*, and *Pull Up Methods* affect at least two classes, including changes affecting a class hierarchy. Yet, these refactorings have similarities with other refactoring types, *e.g. Move Method* moves a method from one class to another, similarly to *Push Downs* and *Pull ups*. We chose *Pull Up* to understand this method movement in the context of a class hierarchy. We avoid textual refactorings such as renames. Given their simpler and lexical nature, they have less room for structural customization.

The collected modification types may not consider all possible modification types. We used Eclipse JDT library because this library has a very fine level of granularity. In this way, we could detect a large number of modifications. Besides, this library is commonly used to build automated refactoring tools for Eclipse and RMiner.

Finally, the use of other tools and a larger refactoring interval (considering more than one commit) could present complementary results, such as new customization patterns. However, these supplemental patterns do not invalidate the ones currently reported in our paper nor the limitations of the IDEs.

External Validity. We performed an in-depth analysis of refactoring instances from 13 Java projects. However, our results might not necessarily hold to other projects involving other primary programming languages and/or from domains not covered by our dataset. Moreover, we focused our analysis on open-source projects. The nature of refactoring in closed-source projects is not necessarily the same as refactoring in open-source ones. However, popular open-source projects have a major concern with software modularity, tending to continuously refactor the source code. We analyzed projects with differing sizes/domains and all key findings were uniform. These projects have an active community, according to Github metrics.

VI. Conclusion

We presented a study to understand in what ways developers customize refactoring in practice and how to improve refactoring tools to properly support these customizations. We investigated the most frequent customization patterns for four refactorings types in 13 Java projects. The results revealed that developers frequently added new modifications, or remove some, of the standard set for each refactoring type. These changes to the standard set customize the refactorings for the specific developer's scenarios. We then listed the current limitations of popular IDEs that should be improved to provide adequate support for these customizations. We also observed that developers agree with the relevance of customizations and show interest in having tool support for recurring customizations.

Finally, it is important to highlight that the modification sets currently considered standard ones are far from being enough to address practical needs. It is also important to consider the fact that the lack of support for refactoring customization might intensive side effects. As future work, we plan to design and implement tool support for better assisting developers in performing customized refactorings. We also intend to expand the number of refactoring types and projects, reducing the threats to external validity.

Acknowledgements

This study was in part financed by CNPq (141276/2020-7, 141054/2019-0); FAPERJ (22520-7/2016, 010002285/2019, E-26/211.033/2019, 202621/2019), FAPEAL (60030.0000000161/2022)and PDR-10 Fellowship (202073/2020); and IEEA-RJ (001/2021).

References

[1] M. Fowler, *Refactoring*, 1st ed. Addison-Wesley Professional, 1999.

[2] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the impact of refactoring on the relationship between quality attributes and design metrics," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

[3] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes? A multi-project study," in *31st Brazilian Symposium on Software Engineering (SBES)*, 2017, pp. 74–83.

[4] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *40th International Conference on Software Engineering*. ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[5] Eclipse. (2022) Eclipse ide website. [Online]. Available: https://www.eclipse.org/

[6] netbeans. (2022) Netbeans ide website. [Online]. Available: https://netbeans.org/

[7] Jtbrains. (2022) Intelij ide website. [Online]. Available: https://www.jetbrains.com/

[8] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE software*, vol. 25, no. 5, pp. 38–44, 2008.

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

[10] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 7, pp. 633–649, 2014.

[11] D. Tenorio, A. C. Bibiano, and A. Garcia, "On the customization of batch refactoring," in *3rd International Workshop on Refactoring*. IEEE Press, 2019, pp. 13–16.

[12] J. Oliveira, R. Gheyi, M. Mongiovi, G. Soares, M. Ribeiro, and A. Garcia, "Revisiting the refactoring mechanics," *Information and Software Technology*, vol. 110, pp. 136–138, 2019.

[13] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *24th International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 858–870.

[14] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi, "Does refactoring improve software structural quality? a longitudinal study of 25 projects," in *30th Brazilian Symposium on Software Engineering*, 2016, pp. 73–82.

[15] Netty. (2017) Removing a seekaheadnobackarrayexception to avoid exception handling. Available at: https://github.com/netty/netty/commit/b03b0f22d1e.

[16] A. Tomcat. (2014) Apply patch 12 from jboynes to improve cookie handling. Available at: https://github.com/apache/tomcat/commit/0cdfed561d.

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

[18] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 5–18, 2012.

[19] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells," in *Foundations of Software Engineering (FSE)*, 2017, pp. 465–475.

[20] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim, "A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells," in *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–11.

[21] W. F. Opdyke, "Refactoring: A program restructuring aid in designing object-oriented application frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[22] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for smalltalk," *Theory and Practice of Object systems*, vol. 3, no. 4, pp. 253–263, 1997.

[23] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering (TSE)*, 2020.

[24] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *18th ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 371–372.

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

[26] P. Meananeatra, "Identifying refactoring sequences for improving software maintainability," in *27th International Conference on Automated Software Engineering (ASE)*, 2012, pp. 406–409.

[27] D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, 2020.

[28] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of JDeodorant: Lessons learned from the hunt for smells," in *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 4–14.

[29] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *24th International Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 535–546.

[30] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *22nd International Conference on Program Comprehension*, 2014, pp. 146–156.

[31] S. Xu, A. Sivaraman, S.-C. Khoo, and J. Xu, "Gems: An extract method refactoring recommender," in *IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 24–34.

[32] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio, "Behind the intents: An in-depth empirical study on software refactoring in modern code review," in *17th International Conference on Mining Software Repositories*, 2020, pp. 125–136.

[33] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta, "Why developers refactor source code: A mining-based study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–30, 2020.

[34] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 287–297.

[35] J. S. Moreira, E. L. Alves, and W. L. Andrade, "An exploratory study on extract method floss-refactoring," in *35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1532–1539.

[36] J. Brant and F. Steimann, "Refactoring tools are trustworthy enough and trust must be earned," *IEEE Software*, vol. 32, no. 6, pp. 80–83, 2015.

[37] Tomcat. (2011) Re-fixed bug #49711: Httpservletrequest#getparts() does not work. Available at: https://github.com/apache/tomcat/commit/f69c17895.

[38] (2022) The untold story of code refactoring customizations in practice. Complementary materials. [Online]. Available: https://customrefactoring.github.io

[39] H. Borges and M. T. Valente, "What's in a GitHub star? Understanding repository starring practices in a social coding platform," *J. Syst. Softw. (JSS)*, vol. 146, pp. 112–129, 2018.

[40] (2022) Elasticsearch-hadoop. [Online]. Available: https://github.com/elastic/elasticsearch-hadoop

[41] (2022) Hystrix. [Online]. Available: https://github.com/Netflix/Hystrix

[42] (2022) Fresco. [Online]. Available: https://github.com/facebook/fresco

[43] (2022) Achilles. [Online]. Available: https://github.com/doanduyhai/Achilles

[44] (2022) Ikasan. [Online]. Available: https://github.com/ikasanEIP/ikasan

[45] (2022) Exoplayer. [Online]. Available: https://github.com/google/ExoPlayer

[46] (2022) Signal-android. [Online]. Available: https://github.com/signalapp/Signal-Android

[47] (2022) Netty. [Online]. Available: https://github.com/netty/netty

[48] (2022) Materialdrawer. [Online]. Available: https://github.com/mikepenz/MaterialDrawer

[49] (2022) Derby. [Online]. Available: https://github.com/apache/derby

[50] (2022) Tomcat. [Online]. Available: https://github.com/apache/tomcat

[51] (2022) Hikaricp. [Online]. Available: https://github.com/brettwooldridge/HikariCP

[52] (2022) Material dialogs. [Online]. Available: https://github.com/afollestad/material-dialogs

[53] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *23rd Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2013, pp. 132–146.

[54] Eclipse. (2022) Using the help system. [Online]. Available: https://help.eclipse.org/mars/index.jsp

[55] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer, "Comparison of similarity metrics for refactoring detection," in *8th working conference on mining software repositories*. ACM, 2011, pp. 53–62.

[56] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[57] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, "B-refactoring: Automatic test code refactoring to improve dynamic analysis," *Information and Software Technology*, vol. 76, pp. 65–80, 2016.

[58] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon, "A closer look at real-world patches," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 275–286.

[59] F. Fresco. (2016) Added dropcache to animationbackend. Available at: https://github.com/facebook/fresco/commit/2d82c6c185.
[60] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.
[61] A. Tomcat. (2013) Implment a standard isblocking() method for output. Available at: https://github.com/apache/tomcat/commit/53617a2011.
[62] Netty. (2016) Move generic code to httporspdy-chooser to simplify implementations. Available at: https://github.com/netty/netty/commit/33a810a513.
[63] ——. (2016) Throw exception if keymanagerfactory is used with opensslclientcontext. Available at: https://github.com/netty/netty/commit/ebfb2832b2.
[64] ——. (2016) Add nonstickyeventexecutorgroup. Available at: https://github.com/netty/netty/commit/fc14ca31cb.
[65] A. M. Eilertsen and G. C. Murphy, "Stepwise refactoring tools," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 629–633.
[66] ——, "The usability (or not) of refactoring tools," in *IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2021, pp. 237–248.