Understanding Type Changes in Java

Ameya Ketkar
Oregon State University
ketkara@oregonstate.edu

Nikolaos Tsantalis
Concordia University
nikolaos.tsantalis@concordia.ca

Danny Dig
University of Colorado, Boulders
danny.dig@colorado.edu

ABSTRACT

Developers frequently change the type of a program element and update all its references for performance, security, concurrency, library migration, or better maintainability. Despite type changes being a common program transformation, it is the least automated and the least studied. With this knowledge gap, researchers miss opportunities to improve the state of the art in automation for software evolution, tool builders do not invest resources where automation is most needed, language and library designers cannot make informed decisions when introducing new types, and developers fail to use common practices when changing types. To fill this gap, we present the first large-scale and most fine-grained empirical study on type changes in Java. We develop *state-of-the-art* tools to statically mine 297,543 type changes and their subsequent code adaptations from a diverse corpus of 129 Java projects containing 416,652 commits. With this rich dataset we answer research questions about the practice of type changes. Among others, we found that type changes are actually more common than renamings, but the current research and tools for type changes are inadequate. Based on our extensive and reliable data, we present actionable, empirically-justified implications.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; *Software configuration management and version control systems.*

KEYWORDS

Empirical Study, Type Change Mining, Type Change Adaptations, Type Change Patterns, Type Migration, Git, Commit

1 INTRODUCTION

A *type change* is a common program transformation that developers perform for several reasons: *library migration* [2, 46, 81] (e.g., `org.apache.commons.logging.Log`→`org.slf4j.Logger`), *API updates* [16, 20] (e.g., Listing 1), *performance* [25, 32, 33] (e.g., `String`→`StringBuilder`), *abstraction* [82] (e.g., `ArrayList`→`List`), *collection properties* [26, 27] (e.g., `LinkedList`→`Deque`), *concurrency* [23] (e.g., `HashMap`→`ConcurrentHashMap`), *security* [30] (e.g., `Random`→`SecureRandom`), and *maintainability* [19] (e.g., `String`→`Path`).

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409725

To perform a type change, developers change the declared type of a program element (local variable, method parameter, field, method return type) and adapt the code referring to this element (within its lexical scope) to the API of the new type. Due to assignments, argument passing, or public field accesses, a developer might perform a series of type changes to propagate the type constraints for the new type.

Listing 1: Type Change example

```
- SimpleDateFormat formatter= new SimpleDateFormat("yyyy");
+ DateTimeFormatter formatter= DateTimeFormatter.ofPattern("yyyy");
- Date d = formatter.parse(dateAsString);
+ LocalDate d = LocalDate.parse(dateAsString, formatter);
```

In contrast to refactorings that are heavily automated by all popular IDEs [5, 6, 35, 44], developers perform the vast majority of *type changes* manually. Ideally, type changes should be automated in a similar way as a developer renames a program element in an IDE, although we recognize that it is a far more challenging problem. The first step to advance the science and tooling for automating type change is to thoroughly understand its practice.

Most of the prior work studied type changes in the context of other evolution tasks such as API updates [16, 20, 22] and library migration [2, 46, 81]. However, there is a gap in understanding type changes in the general context. This gap in knowledge negatively impacts four audiences:

(1) **Researchers** do not have a deep understanding of type changes and the role they play in software evolution. Thus, they might not fully understand and support higher level tasks, such as automated program repair [10, 58] that are composed from type changes.

(2) **Tool builders** do not have an insight into the practice of type changes. Thus, they (i) are not aware if the type changes they automated [35, 44, 45, 49, 82] are representative of the ones commonly applied in practice, (ii) fail to identify new opportunities for developing automation techniques.

(3) **Language and Library Designers** continuously evolve the types their clients use. However, designers are not aware of *what* types are commonly used and *how* the clients adapt to new types. Without such knowledge they cannot make informed decisions on how to improve or introduce new types.

(4) **Developers** miss educational opportunities about common practices applied when changing types in other projects, which could benefit their own projects.

To fill this gap, in this paper we present the first longitudinal, large-scale, and most fine-grained empirical study on type changes performed in practice. We analyze the commit histories of 129 large, mature, and diverse Java projects hosted on GitHub. To do this, we developed novel tools, which efficiently and accurately mined 416,652 commits and detected 297,543 instances of type changes. We thoroughly evaluated our tools and they have 99.7% precision and 94.8% recall in detecting type changes. To advance the science and tools for type change automation, we use this rich and reliable dataset to answer six research questions:

RQ1 *How common are type changes?* We found 35% more instances of type changes than renames. Given that type changes are so common, it is worth to investigate how they can be automated.
RQ2 *What are the characteristics of the program elements whose type changed?* We found that 41.6% of the type changes are performed upon *public* program elements that could break the code. In addition, developers frequently change between *Internal* types. The current tool support for such changes is non-existent.
RQ3 *What are the edit patterns performed to adapt the references?* Developers often adapt to the primary type change by performing a secondary type change. For example, in Listing 1 type change `SimpleDateFormat→DateTimeFormatter` triggers a secondary type change `Date→LocalDate`. However, current techniques cannot infer mappings for such cascading type changes.
RQ4 *What is the relationship between the source and target types?* Among others, we found that in 73% of type changes the types are not related by inheritance. In contrast, most of the current IDEs automate type changes for types related by inheritance (e.g., Replace Supertype where Possible [44, 49, 82]). This reveals another important blind spot in the current tooling.
RQ5 *Are type changes applied globally or locally?* In 62% of cases developers perform type changes locally. In contrast, the current tools [40, 45, 49, 62, 82] perform a *global migration* in the entire project. This shows that the tool builders do not invest resources where automation is most needed.
RQ6 *What are the most commonly applied type changes?* From our entire corpus, we filter type changes that developers perform in at least two different projects. We found that these 1,452 type changes represent 2% of all type change patterns, yet they are responsible for 43% of all type change instances. Tool builders should prioritize automating these popular type changes. Developers and educators can learn from these common practices.

Our findings have actionable implications for several audiences. Among others, they (i) advance our understanding of type changes which helps our community improve the science and tools for software evolution in general and specifically type change automation, (ii) help tool designers comprehend the struggles developers face when performing type changes, (iii) provide feedback to language and API designers when introducing new types, (iv) identify common practices for developers to perform type changes effectively, and (v) assist educators in teaching software evolution.

This paper makes the following contributions:
Questions: To the best of our knowledge, this is the first large-scale and most fine-grained (at commit level) empirical study of type changes in Java. We answer six research questions using a corpus of 297,543 type changes. We believe this makes our findings representative.
Tools: We developed novel tools to efficiently detect type changes from a corpus of 416,652 commits. We also manually validated our tools and show they have high precision (99.7%) and recall (94.8%). To help our community advance the science and practice of type changes, we make these tools and the collected data available at [47, 48, 50].
Implications: We present an *actionable, empirically justified set of implications* of our findings from the perspective of four audiences: Researchers, Tool Builders, Language Designers, and Developers.

2 RESEARCH METHODOLOGY

In the rest of the paper we refer to the tuple `(SourceType,TargetType)` as a *Type Change Pattern* (TCP). A *Type Change Instance* (TCI) is applying a TCP on a program element (i.e., variable, parameter, field, method declaration) in a given commit and adapting its references.

2.1 Subject Systems

Our corpus consists of 416,652 commits from 129 large, mature and diverse Java projects, used by other researchers [55] to understand language constructs in Java. This corpus [55] is shown to be very diverse, from the perspective of LOC, age, commits, and contributors. This ensures our study is representative. It is also large enough to comprehensively answer our research questions. The complete list of projects is available online[1].

In this study, we consider all commits in the epoch January 1, 2015 – June 26, 2019, because researchers observed an increasing trend in the adoption of Java 8 features after 2015. Java 8 introduced new APIs like `FunctionalInterface`, `Stream`, `Optional` and enhanced the `Time`, `Collection`, `Concurrency`, `IO` and `Math` APIs. Thus, we use these particular projects and their commits in this particular epoch, because it allows us to collect and study type changes involving the new (≥`Java 8`) built-in Java types. We excluded all merge commits, as done in other studies [78], to avoid having duplicate reports of the same type changes.

2.2 Static Analysis of Source Code History

2.2.1 ***Challenges:*** Most refactoring detection tools [21, 69, 88] take as input two fully built versions of a software system that contain binding information for all named code entities, linked across all library dependencies. However, a recent study [85] shows that only 38% of the change history of software systems can be successfully compiled. This is a serious limitation for performing our longitudinal type change study in the commit history of projects. It poses a threat to the external validity of our empirical study, since only a small number of project versions can be compiled successfully for extracting type changes. Since the majority of versions cannot be compiled, we would not be able to retrieve fine-grained details of the types (e.g., the methods and fields declared, super types), thus making it challenging to understand the characteristics of the types involved in a type change. Moreover, even if we built 38% of the commits in the project history this would be extremely time and resource consuming, preventing our study from scaling beyond a few thousand commits.

We overcome these challenges for performing our fine-grained and large-scale study in two ways. First, we extended the *state-of-the-art* refactoring detection tool, RefactoringMiner [83, 84], to accurately and efficiently detect type changes at commit-level. We built upon RefactoringMiner, because it has been shown to have a superior accuracy and faster execution time [83, 84] than competitive tools also operating at commit-level, such as RefDiff 1.0 [79] and RefDiff 2.0 [77]. Second, we created a novel tool, TypeFactMiner, which accurately and efficiently retrieves detailed information about the types involved in the type change, without requiring to build the software system.

[1] http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/projects.html

Table 1: Precision and recall of our extended version of RefactoringMiner

Refactoring Type	TP	FP	FN	Precision	Recall
Change Parameter Type	597	1	35	99.8%	94.5%
Change Return Type	386	2	14	99.5%	96.5%
Change Variable Type	649	2	42	99.7%	93.9%
Change Field Type	212	1	10	99.5%	95.5%
Average	1843	6	101	99.7%	94.8%

2.2.2 ***Detecting TCIs:*** Detecting accurately a TCI is not straightforward, as these changes can get easily obfuscated by the other changes (i.e., overlapping refactorings) in a commit, where methods and classes containing the TCI get moved, renamed or removed. For this purpose, we extend **RefactoringMiner** [83, 84] to detect 4 kinds of type changes, namely (i) *Change Variable Type*, (ii) *Change Parameter Type*, (iii) *Change Return Type*, and (iv) *Change Field Type*. RefactoringMiner uses a state-of-the-art *code matching algorithm* to match classes, methods and statements inside method bodies, and accurately detect refactorings at commit level. It also records AST node *replacements* when matching two statements, such as type replacements, which we utilize to infer the aforementioned type change kinds.

To evaluate the precision and recall of RefactoringMiner, we extended the oracle used in [84], which contains true refactoring instances found in 536 commits from 185 open-source GitHub projects, with instances of the four type change kinds. To compute precision, the first two authors manually validated 1843 TCIs reported by RefactoringMiner. Most of the cases were straightforward, and thus were validated individually, but some challenging cases were inspected by both authors to reach an agreement. To compute recall, we need to find all true instances for the 4 type change kinds. We followed the same approach as in [83] by executing a second tool, namely GumTree [36], and considering as the ground truth the union of the true positives reported by RefactoringMiner and GumTree. GumTree takes as input two abstract syntax trees (e.g., Java compilation units) and produces the shortest possible edit script to convert one tree to another. We used all *Update* edit operations on types to extract TCIs and report them in the same format used by RefactoringMiner. Table 1 shows the number of true positives (TP), false positives (FP), and false negatives (FN) detected/missed by RefactoringMiner. Based on these results, we conclude that our extension of RefactoringMiner has an almost perfect precision and a very high recall (ranging between 94 and 96.5%) in the detection of TCIs. Thus, our results in Section 3 are reliable.

2.2.3 ***Detecting the adaptations of references:*** RefactoringMiner analyzes the matched statements referring to a certain variable to increase the precision in the detection of variable-related refactorings, such as variable renames. We use these references, to understand how developers adapt the statements referring to the variable/parameter/field whose type changed. For *Change Local Variable Type*, *Change Parameter Type*, and *Change Field Type*, RefactoringMiner reports all matched statements within the variable's scope referring to the variable/parameter/field on which the TCI was performed. While for *Change Return Type*, it returns all matched `return` statements inside the corresponding method's body.

If these matched statements are not identical, RefactoringMiner reports a set of AST node *replacements*, which if applied upon the statement in the parent commit would make it identical to the matched statement in the child commit. Using these AST node replacements, we extract the 11 most common *Edit Patterns (RQ3)* performed to adapt the statements referencing a variable whose type changed. RefactoringMiner reported 532,366 matched statements for 297,543 mined TCIs, creating a large data set of real world edit actions performed to adapt the references in a type change.

2.2.4 ***Qualifying Type Changes:*** Analyzing only the syntactic difference of AST Type nodes is not enough for correctly detecting type changes. For instance, when a TCI qualifies the declared type (e.g., **`Optional`**`<String>` → **`java.util.Optional`**`<String>`) there is no actual type change. In such cases, it is important to know the qualified name of the type before and after the TCI is applied. If `Optional` was bound to `java.util.Optional`, there is no type change. But if `Optional` was bound to `com.google.common.base.Optional`, then there is a type change.

Moreover, to record accurately the type changes that are more commonly performed *(RQ6)* and their characteristics *(RQ2)*, we need to know the fully-qualified types that changed when a TCI was performed. For example, if the type change is **`List`**⟶**`Optional`**, depending upon the context (i.e., import declarations) where the types are used, `List` could correspond to `java.util.List` or `io.vavr.List` and `Optional` could correspond to `java.util.Optional` or `com.google.common.base.Optional`. Finally, knowing further details about these types, such as the fields and methods they declare, or their super types, would allow us to do an in-depth investigation of the relations between the changed types.

However, there are certain challenges in extracting the qualified types, without building the commit: (1) we have an incomplete source code of the project, because we analyze only the modified/added/deleted java files in a commit, (2) we do not have the source code of the types declared in external libraries. To mitigate these challenges we developed a novel tool **TypeFactMiner**, which efficiently and accurately infers the fully qualified name of the type to which a variable declaration type is bound.

Collecting Type Bindings from commit history: At the core of TypeFactMiner are heuristics, which reason about the import statements, package structure, and the types declared in the entire project, similar to the ones discussed by Dagenais et al. [17]. To represent the types declarations, TypeFactMiner uses *Type Fact Graphs* (TFG), recently proposed by Ketkar et al. [49]. These are abstract semantic graphs that capture each declared class/interface/enum, the qualified signature of methods/fields it declares, and the local variables/parameters declared inside methods. For the oldest commit, which contains at least one Java file, we map all existing type declarations to a TFG. For the subsequent commits, we incrementally update this TFG by analyzing only the added, removed, moved, renamed and modified files. This optimization allows us to scale our fine-grained study to hundreds of thousands of commits. The TFG representation allows us to infer transitive inheritance or composition relationships between types (RQ4).

Collecting Type Bindings from external libraries: A type change often involves types that are declared in the standard

Java library (e.g., `java.lang.String`) or third party libraries (e.g., `org.slf4j.Logger`). For this purpose, TypeFactMiner analyzes the bytecode of the project's library dependencies to extract information for the publicly visible type declarations (classes/interfaces/enums). For the types declared in JDK, TypeFactMiner analyses the jars contained in the **`openjdk-8`** release. For external dependencies, TypeFactMiner fetches the corresponding jar files for the dependencies required by the project at each commit. Since a project can contain multiple `pom.xml` files (for each module) with dependencies amongst them, TypeFactMiner generates an `effective-pom` [4] at each commit and parses this file to identify the external dependencies. It then connects all external type bindings to the nodes in the TFG of the analyzed project according to their usage.

The validity of the answers to our research questions relies upon how accurately TypeFactMiner infers the fully qualified names of the types. To compute the precision of TypeFactMiner, we create a golden standard, based on the qualified names returned by the Eclipse JDT compiler. However, to obtain these qualified names one will have to build the commits, which is time consuming because each commit might require a different build tool version, Java version, and build command arguments. This prevented us from randomly sampling commits from our dataset. So, we selected 4 projects, namely `guava`, `javaparser`, `error-prone` and `CoreNLP` and automated the process to build each commit (and its parent) that contained a TCI. We were able to build successfully 467 commits that contained 4715 TCIs. For the program elements involved in the TCIs, we obtained the qualified types from Eclipse JDT compiler, which we use as the golden standard. We found that TypeFactMiner correctly inferred the qualified names for the types involved in 4652 TCIs (i.e., 98.7% precision).

Out of 428,270 TCIs found in 40,865 commits, TypeFactMiner filtered out 130,727 TCIs, where (i) the corresponding types were simply renamed or moved to another package within the examined project (i.e., an internal Rename/Move Type refactoring triggered the type change) (ii) no actual type change happened (i.e., a non-qualified type changed to qualified and vice-versa), leading to a total of 297,543 *true* TCIs. The **efficient** and **accurate** tools we created and validated allow us to collect *extensive* and *reliable* results, to empirically justify our implications.

3 RESULTS

3.1 RQ1: How common are type changes?

To provide some insight about how commonly developers perform type changes, we compare this practice with another commonly applied source code transformation, namely the renaming of identifiers (i.e., Rename refactoring) [7, 60]. Such a comparison is feasible, because both type change and rename can be performed on the same kind of program elements, i.e., local variables, parameters, fields, and method declarations. All these program elements have a name and a type (return type in the case of method declarations). Thus, it is possible to make a direct comparison between the number of type changes and renames for the same kind of program elements to understand which practice is more common. Morever, RefactoringMiner detects rename refactorings with an average precision of 99% and recall of 91% [83], which are very close to the average precision/recall values reported in Table 1 for type changes, allowing for a fair comparison of the two practices.

Table 2 shows the number of type changes and renames on variables, parameters, fields, and methods that RefactoringMiner detected in 95,576 commits. As we can see in Table 2, type changes are around 50% more populous than renames on variables, parameters and fields, while return type changes are slightly more populous than method renames. Moreover, we observed that 297,543 type changes occur in 40,865 commits, while 219,356 renames occur in 46,699 commits, i.e., the density of type changes is higher (7.3 per commit) than that of renames (4.7 per commit).

Table 2: Mined Source Code Transformations

	Variable	Parameter	Field	Method	Total
Type Change	83,393	93,229	48,279	72,642	297,543
Rename	53,416	63,612	30,852	71,476	219,356
Δ Percentage	+56.1%	+46.6%	+56.5%	+1.6%	+35.6%

RQ1 Conclusion: Type changes are more commonly and frequently performed than renames. In comparison to renaming, there is negligible tool support and research for type changes.

3.2 RQ2: What are the characteristics of the program elements whose type changed?

To answer this question, we studied various characteristics of the program elements involved in TCIs. We explore characteristics, such as (i) *kind:* field, method or variable, (ii) *visibility:* public, private, protected or package, (iii) *AST Type node:* simple, primitive, array, or parameterized and (iv) *namespace:* internal, standard Java library, or third-party library.

Assume project p has n commits, and $TCI(e, i)$ is a type change instance on program element e in commit i, and x is a value for characteristic y of program elements, we define:

$$\textbf{\textit{proportion(p, x)}} = \frac{\sum_{i=1}^{n} |\{TCI(e,i) \mid e \text{ has } x \text{ value for characteristic } y\}|}{\sum_{i=1}^{n} |\{TCI(e,i) \mid e \text{ has any value for characteristic } y\}|}$$

Further, assume that $\text{elements}(x, y, i)$ is the set of all program elements having value x for characteristic y in the modified files of commit i, regardless of whether their type changed or not, we define:

$$\textbf{\textit{coverage(p, x)}} = \frac{\sum_{i=1}^{n} |\{TCI(e,i) \mid e \text{ has } x \text{ value for characteristic } y\}|}{\sum_{i=1}^{n} |\text{elements}(x,y,i)|}$$

Only studying the proportions of the different values a characteristic can take (e.g., the *visibility* characteristic takes values `public`, `private`, `protected`), may result in misleading findings, because it does not take into account the underlying population distribution. For example, by studying proportions we could find that most TCIs are performed on `public` elements, just because there are more `public` elements in the source code of the examined projects. Therefore we study the *coverage* of the different values a characteristic can take, with respect to all program elements having the same value for that characteristic (whose type did or did not change). The *project-level* distributions for the proportion and coverage of the different values of a characteristic are shown as Violin plots. To assess if there is a statistical difference among these distributions, we perform the Kruskal-Wallis test (the result of the test is shown on top of each Violin plot). A p-value ≤ 0.05

rejects the null hypothesis that the medians of the distributions are equal. To compare two distributions with visibly close medians, we report p-values obtained by performing the pair-wise post-hoc Dunn's test.

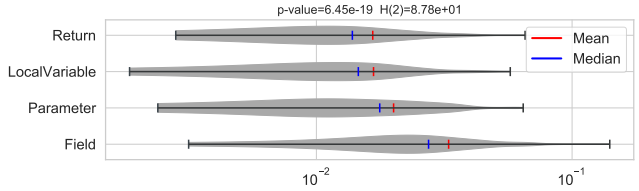
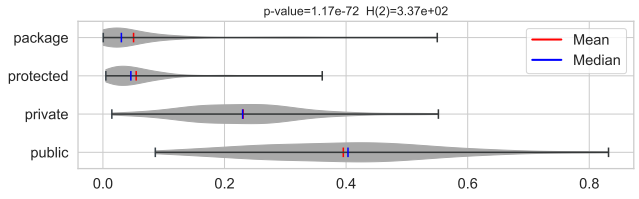


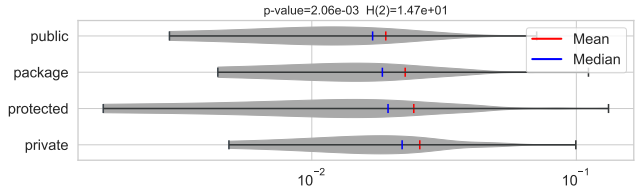
Figure 1: Project-level distribution of type change coverage per program element kind

3.2.1 ***Program element kind:*** The scope of the element on which a type change is applied determines the impact the change has upon the program. Transforming a *field*, *method return type* or *method parameters* affects the API of the program, while transforming *local variables* affects the body of a single method only [60]. Table 2 shows that the largest proportion of type change affects method parameters, followed by local variables. However, Figure 1 shows that the median coverage of performing *Change Field Type* is the largest. Our results are in congruence with the results obtained by Negara et. al [61] who surveyed 420 developers, and ranked *Change Field Type* as the most relevant and applicable transformation that they perform. They also report that *Change Field Type* is the most highly desired feature in IDEs.

3.2.2 ***Program element visibility:*** If a type change affects the signature of a *package* visible method, a developer should update the call sites of this method within the same package. However, if this method is *public* visible, a developer should update the call sites of this method in the entire program, but more importantly this type change could introduce backward incompatibility for the clients of the library. Figure 2 shows the proportion and coverage for the access levels - *public*, *private*, *protected* and *package*.



(a) Proportion of type change instances



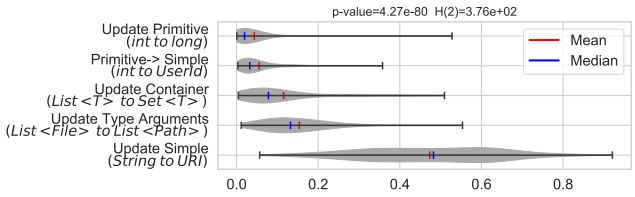
(b) Coverage of type change instances

Figure 2: Project-level distribution per visibility kind

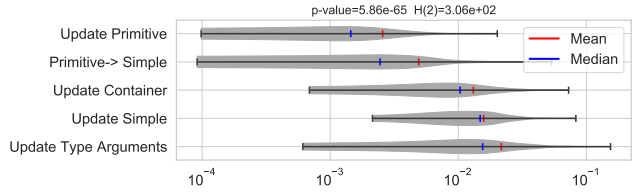
Figure 2a shows that type changes are most commonly applied on `public` program elements. However, in Figure 2b the coverage of type changes on `public` elements is lower than `private` elements (p-value=0.0013). This result shows that although the raw number of type changes on `private` elements is less than the number of type changes on `public` elements, developers tend to change more often the types of `private` elements compared to `public` ones. This indicates that developers are more cautious when performing type changes on `public` elements, possibly taking into account backward incompatibility issues.

Researchers [16, 20] have thoroughly studied the impact of different kind of changes on software evolution. Cossette et al. [16] categorize type change as a hard to automate breaking change. Dietrich et al. [20] categorize a change based on binary and source code incompatibility. We analyze the type changes that are performed on `public` elements and observe that 14.2% introduce binary but no source incompatibility, while the remaining introduce both. Below we report the occurrences of type changes applied on `public` elements based on the proposed categories in [20]:

(1) **Binary and Source Incompatible:** We found 106,329 TCIs (35.8%) that can potentially introduce breaking changes.
(2) **Binary Incompatible but Source Compatible:** This interesting phenomenon appears in Java programs when the code compiles, but results in a runtime failure, due to mismatch of rules between compiler and JVM. We found the following instances in our corpus: *method return type replaced by subtype* (4,249), *method parameter type replaced by supertype* (6,437), *primitive narrowing of return type* (1,373), *wrapping and unwrapping of primitive parameter and return types* (1,663), and *primitive widening of method parameters* (3,258).



(a) Proportion of type change instances



(b) Coverage of type change instances

Figure 3: Project-level distribution per AST Type node kind

3.2.3 ***Program element AST Type node:*** Java developers have to explicitly define the type for all declared methods and variables. Java 8 allows nine kinds of syntactic structures to express the declared type of elements [43]. For example, Simple (`String`), Parameterized (`List<Integer>`), Primitive (`int`), Array (`int[]`).

According to Figure 3a, *Simple Types* are more commonly changed than other AST Type nodes. However, in Figure 3b, we can observe that the coverage of changing the *Type Arguments* of parameterized types is the most (p-value=4.09×10^{-32}). Changing the *Type Arguments* of parameterized types is a more complex task than changing *Simple Types*, because there are

additional type changes that propagate through the parameterized container. For example, in Listing 2 to perform the change `IgniteBiTuple<String,AtomicLong>`→`IgniteBiTuple<String,LongAdder>` one would have to propagate the type changes to the call sites of method `Map.Entry.getValue()`, because `IgniteBiTuple` implements the `Map.Entry` interface. Propagating such changes requires inter-procedural points-to and escape analysis, which is not supported by any current tool automating type changes.

Listing 2: Updating the argument of a parameterized type

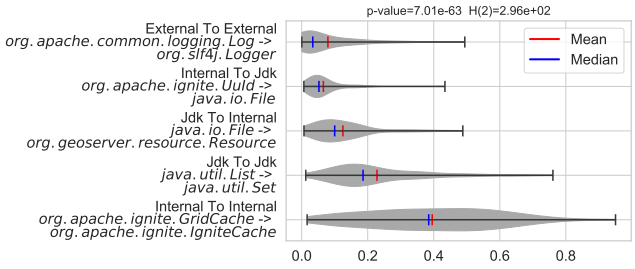
```
- IgniteBiTuple<String, AtomicLong> m = getTuple();
+ IgniteBiTuple<String, LongAdder> m = getTuple();
- AtomicLong l = m.getValue();
+ LongAdder l = m.getValue();
```

3.2.4 ***Program element namespace:*** We categorize program elements based on the **relative location** of the source and target types with respect to the project under analysis. We find the fully qualified name of each type using TypeFactMiner, and label it as: (i) ***Internal*** (type declared in the project), (ii) ***Jdk*** (type declared in the standard Java library), or (iii) ***External*** (type declared in a third party library). We assume that developers can perform more easily type changes involving *Internal* than *External* types, as they are more familiar with the types defined internally in the project, or can ask co-developers in the project who have more expertise on these internal types. On the other hand, type changes involving *External* types are more difficult to perform, as developers need to study external documentation, which might be outdated or unavailable, or refer to Q&A forums for more information.

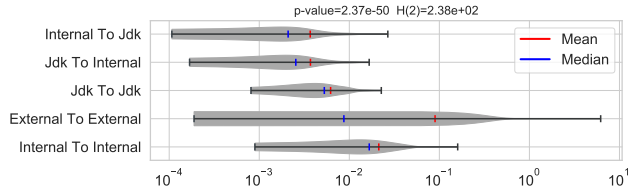
For *Simple Type* changes, we qualify the types before and after the change. For *Type Argument* changes to parameterized types, we qualify the changed type arguments (e.g., `List<File>`→`List<Path>`, the source type is `java.io.File` and the target type is `java.nio.file.Path`). For composite type changes (e.g., `List<Integer>`→`Set<Long>`), we qualify the base type changes (e.g., `java.util.List`→`java.util.Set` and `java.lang.Integer`→`java.lang.Long`).

Figure 4a shows that developers most commonly change `Internal`→`Internal` and `Jdk`→`Jdk` types. Type changes between `External` types are rarely performed (5.14%), of which only 27.8% have source and target types defined in different external libraries. This confirms the findings of Teyton et al. [81], who conclude that third-party library migration is not a common activity.

However, in Figure 4b, we can observe that the median of `External`→`External` type change coverage is greater than that of `Jdk`→`Jdk` (p-value=5.3×10^{-35}). In addition, the mean of `External`→`External` type change coverage is the largest. To further understand this distribution, we identify outliers using the Q3 + 1.5 × IQR rule. We investigate the TCIs performed in 21 outlier projects and find that developers perform hundreds of such `External`→`External` TCIs. For example, we found 1902 `org.apache.common.Log`→`org.slf4j.Logger` TCIs in 8 projects for library migration and 1254 TCIs in 7 projects to update from `google.protobuf-2` to `google.protobuf-3`. The results also show the importance of inferring the type-mappings to perform a library migration or update. Migration mapping mining techniques [3, 18, 72, 87] have focused on mining method-level mappings and have missed the type-mappings across the libraries.



(a) Proportion of type change instances



(b) Coverage of type change instances

Figure 4: Project-level distribution per namespace kind

RQ2 Conclusion: (i) 41.6% of type changes affect *public* elements, introducing binary and/or source incompatibilities. (ii) Updating *Type Arguments* of parameterized types has the largest type change coverage; however, the current state-of-the-art tools do not support the changes that need to be propagated.

3.3 RQ3: What are the edit patterns performed to adapt the references?

From the 297,543 TCIs mined, we found that developers applied some edit pattern to adapt the references in 130,331 TCIs (43.8%). To further shed light on these cases, in Table 3 we report the percentage of TCIs for different edit patterns. We observe that for 54.85% of TCIs with edited references, developers rename the program element whose type changed (e.g., `String filepath` → `File file`). This makes sense as developers try to use variable names that are intention-revealing. This makes it easy to understand and maintain the program because the names reflect the intention of the new type. Arnaoudova et al. [7] were the first to observe this qualitatively, but we are the first to measure this quantitatively.

Table 3: Mined edit patterns

Description	%TCI	Example
`Rename variable`	54.85%	`String filepath`→ `File file`
`Rename Method call`	7.09%	`applyAsLong`→`applyAsDouble`
`Modify arguments`	2.70%	`apply(id)`→`apply(usr.getId())`
`Modify Method call`	25.69%	`f.exists()`→`Files.exist(p)` `s.length()`→`s.get().length()`
`Replace with Method call`	0.82%	`new Long(5)`→`Long.valueOf(5)`
`(Un)Wrap argument`	0.99%	`read(p)`→`read(Paths.get(p))`
`Update Literal`	0.51%	3→3L or "1"→"1.0"
`(Un)Cast`	0.13%	5/7 →(double)5/7
`Cascade same types`	12.06%	See Listing 2
`Cascade different types`	5.81%	See Listing 1
`Assignment↔Call`	0.26%	`b = true`↔`b.set(true)`

The second most applied pattern to adapt references is to adapt method calls by updating the name, modifying the call chain structure or modifying the receiver or the arguments. This requires inferring method-mapping between the source and target types and the type-mapping between the return type and arguments of these methods. This result motivates the work of researchers that infer API mappings [72].

The third most commonly applied pattern to adapt references is *cascade* type changes, which involves additional type changes to other places in the code. For example, in Listing 1 the type change `SimpleDateFormat→DateTimeFormatter` applied to variable `formatter` triggers another type change `Date→LocalDate` (i.e., cascade type change) applied to variable `d`. We found that in 12.06% of TCIs developers perform a *cascade* type change, which is similar to the original type change, while in 5.81% of TCIs the *cascade* type change is different from the original. To perform such cascade type changes, the replacement rules must cover all potential type changes between the source and target type. This requirement was initially discovered by Li et al. [53], but our study is the first one to empirically show that this happens often in practice.

For 167,212 TCIs (56.2%), we did not find any edit patterns that developers performed to adapt the code to the type change. On further investigation, we found that (i) the variables whose type changed are passed as arguments to method calls before getting actually consumed (i.e., some API method is called), (ii) the type changes involved wildcard types (T → ? extends T), hierarchically related types, primitive widening/narrowing, or (un)boxing.

RQ3 Conclusion: In 54.85% type changes, developers rename the variables to reflect the changed type. In 17.87% of type changes developers perform a *cascade* type change involving the same or different types.

3.4 RQ4: What is the relationship between the source and target types?

To answer this question, we check whether the source and target types are related by (i) inheritance i.e., the types have a subtype or supertype relationship, or the types share a common super type (other than `java.lang.Object`), or (ii) composition i.e., one type is composed of the other.

We found that in 7.5% of the TCIs the types are composition-related. In 27.08% of the TCIs the types have an inheritance relationship. The tools [35, 45, 82] for performing type changes have exclusively focused on parent-child relationships (e.g., Use Supertype Refactoring). However, we found that 44.56% of the inheritance-related type changes actually have a sibling relationship (e.g., `List→Set`). This highlights a blind spot in the current tooling for 85.1% type changes, where the source and target types are siblings, composition-related, or have no relationship.

Composition and Inheritance are two ways of designing types based on *what they are* or *what they do*. The seminal work on Design Patterns by the Gang of Four [38] often advocates composition over inheritance. We are interested to find the effect of this design choice when performing a type change. Thus, we define:

$$\textit{Adapted Statement Ratio} = \frac{|\text{Adapted Statements}|}{|\text{Referring Statements}|}$$

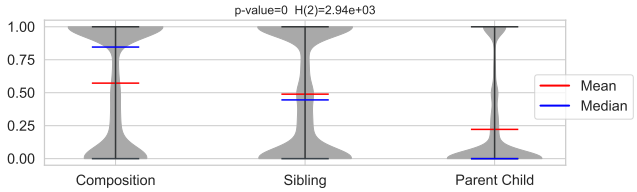


Figure 5: Distribution of adapted statement ratio

Referring Statements is the set of matched AST statement pairs within the scope of the variable on which a type change was applied, that reference this variable. These statements can be considered as the statements belonging in the def-use chain [66] of the variable whose type changed. *Adapted Statements* is the subset of *Referring Statements*, where an edit was performed to adapt to the type change. This ratio is (i) non-negative and normalized within the interval [0, 1], (ii) has true null value of *0*, when no edits are performed (e.g., `ArrayList→List`, where both types share a similar API), (iii) has a maximum value of *1* when all *Referring Statements* are edited (e.g., `java.io.File→org.neo4j.io.DatabaseLayout`, where the two types share no common API).

Figure 5 shows the distribution of *adapted statement ratio* corresponding to TCIs where the source and target types have a composition, sibling, and parent-child relationship, respectively. The violin plots show that the median *adapted statement ratio* is higher when the source and target types have a composition relationship than when they have an inheritance relationship (sibling or parent-child). Moreover, the median *adapted statement ratio* is higher when the source and target types have a sibling relationship than when they have a parent-child relationship.

Table 4: Edit patterns to adapt TCIs grouped by relationship

Edit pattern	Composition	Sibling	Parent-Child
Rename Identifier	77.36%	45.03%	40.9%
Rename Method Call	3.91%	9.9%	6.4%
Modify Method Call	34.26%	30.54%	24.8%
Cascade Type Change	10.28%	19.56%	8.93%

To gain further insight into this, we analyze the edit patterns applied w.r.t. the relationship of the source and target types. Table 4 shows that developers rename identifiers more often when the source and target types have a composition relationship than a hierarchical relationship. Since, identifier names represent defined concepts [71], one possible explanation is that developers assign names to program elements based on *what they represent* and not based on *what they do*. Performing type change between hierarchically related types does not change what the element represents (e.g., `ArrayList` is a `List`, whereas `List` and `Set` are both `Collections`), while this is not always true when types are related by composition (e.g., `File` and `DatabaseLayout` represent different concepts).

Furthermore, developers modify method calls more often when the source and target types are related by composition. For example, when `neo4j` developers performed the type change `File→DatabaseLayout`[2], they consistently replaced the references to variables representing directories with getter calls `layout.getDirectory()`.

[2] http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/neo4j/tci_project3859.html

Sibling types often provide different methods (e.g., `List` provides methods to add and remove an element in a specific index through methods `add(int index, E element)` and `remove(int index)`, while `Set` does not offer such functionality). They also provide similar methods through their common supertype. Table 4 shows that, modifying or renaming a method invocation is a common edit pattern, when adapting to a type change between *sibling* types. In fact, previous researchers who proposed techniques to perform such type changes, identify one-to-one and one-to-many method mappings to modify or rename method invocations. For example, Dig et al. [23] replace `ConcurrentHashMap` with `HashMap`, Tip et al. [82] replace `Vector` with `ArrayList`, Li et al. [53, 68] replace `HashTable` with `HashMap`, and Ketkar et al. [49] replace `Function` with `UnaryOperator`.

RQ4 Conclusion: In 65.42% of type changes, the source and the target types have no hierarchical or composition relationship. Despite the advantages of using composition over inheritance, when it comes to changing types, composition requires more adaptations than inheritance.

3.5 RQ5: How common are type migrations?

Are type changes applied globally in the form of a type migration, or selectively on specific parts of the code? How often do developers perform type migration? What are the most common migrations? Answering such questions is important for software evolution researchers and tool builders to better support the common development practices.

To compute the percentage of migration for a given `SourceType` in a project, we need to count the instances where `SourceType` has been changed to any `TargetType`, and the instances where `SourceType` has not been changed in the commit history of the project. We decided to study the migration phenomenon on a `SourceType` level, instead of a type change level (`SourceType`→`TargetType`), because we found that in many cases developers change a given `SourceType` to multiple `TargetTypes` depending on the context. For example, in project `google/closure-compiler` Guava type `ImmutableEntry` has been changed in some places to `BiMapEntry` and in other places to Java's `Map.Entry` depending on desired property.

Assume project p has n commits ($1 \leq n \leq$ |all commits in p|), where each of these commits contains at least one occurrence of a TCI involving `SourceType` t. Further, $TCI(e, i)$ is a TCI involving `SourceType` t on program element e in commit i, and elements(t, i) is the set of all program elements having type t in all Java files of commit i, we define:

$$\textbf{\textit{coverage(p, t)}} = \frac{\sum_{i=0}^{n-1} |\{TCI(e,i)\}| \ + \ |\{TCI(e,n) \mid e \text{ has SourceType } t\}|}{\sum_{i=0}^{n-1} |\{TCI(e,i)\}| \ + \ |\text{elements}(t,n)|}$$

In the formula above, elements(t, n) represents the program elements having type t in commit n of project p. If all these elements are involved in a TCI in the last commit where `SourceType` t has been changed, then `SourceType` t is *migrated* (i.e., *coverage(p, t)* = 100%).

Figure 6 shows the distribution of type change coverage for three categories of `SourceType`, namely *Internal*, *External* and *Jdk* types. We can clearly observe that *Jdk* types are more selectively changed, while *Internal* and *External* types tend to be more globally changed (median = 0.51 and 0.34 respectively). In addition, *Internal* types are more globally changed than *External* types (p-value = 3.6×10^{-32})

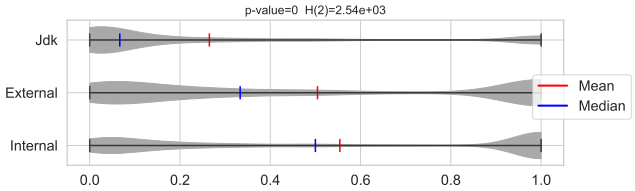


Figure 6: Project-level distribution of type change coverage

with relatively more migrations, i.e., developers migrate *Internal* types (45.2%) more than *External* (38.3%) and *Jdk* (16.1%) types. This highlights a major blind spot in previous research on type migration [49, 53, 82] that focuses mainly upon migration between *Jdk* types (e.g. `Vector` ⟶ `ArrayList` or `HashTable` ⟶ `HashMap`).

We found that 16 projects migrated `FinalizerThread` from Jdk to `FutureTask` from Jdk, `TrustedFuture` and `InterruptibleTask` from Guava, or `LeaderSwitcher` from neo4j. Moreover, 8 projects migrated `GeneratedMessage` from google/protobuf to `GeneratedMessageV3`. The complete list of migrations[3] can be found at our website [50].

RQ5 Conclusion: In 61.71% of cases developers perform type changes in a *selective* rather than a *migration* fashion. *Type Migration* is most commonly performed on *Internal* types.

3.6 RQ6: What are the most commonly applied type changes?

We group all 297,543 TCIs by the tuple <`SourceType`, `TargetType`>, expressing a TCP. For instance, in Listing 1 there are two type changes, namely <`SimpleDateFormat`, `DateTimeFormatter`> and <`Date`, `LocalDate`>, and in Listing 2 there is one type change <`AtomicLong`, `LongAdder`>. We found a total of 50,640 distinct TCPs. To find the most popular TCPs from our dataset, we select those that were performed in at least 2 projects. This results in 1,452 TCPs[4] that collectively account for 64,310 TCIs.

We found that 70.2% of the popular TCPs involve `Jdk` types. The **10 most** popular TCPs involve (i) *primitive types*: `int`, `long`, `void`, `boolean`, and (ii) sibling types with a common supertype: `java.util.List`, `java.util.Set`, `java.util.Map`. Other popular TCPs involve types declared in `java.util`, `java.lang`, `java.time` and `java.io`. None of these TCPs are supported by the current tools.

We further analyse the popular TCPs and find that 40% of these involve `Internal` types, while the rest involve `External` types. This result is surprising, as we did not expect to find any TCPs involving `Internal` types, because they would get filtered out by the "at least two projects" predicate. On further investigation, we found that TCPs involving `Internal` types for a given project, affect dependent projects that need to adapt to the `External` type change. For example, the developers of *apache/hbase* changed the return type of 22 public methods from `HRegion` to `Region`. When the projects *apache/hadoop*, *phoenixframework/phoenix* bumped their *hbase* dependency to version `1.1.3`, they adapted the invocations of these methods by performing the same type change (`HRegion`⟶`Region`). Such scenarios occur when library developers introduce a breaking change and all the clients adapt to that change when they update their dependencies. The results highlight a blind spot in the current

[3] http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/Migrations.html
[4] http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/A/popular.html

research that has primarily focused on library migration and update [3, 16, 20, 22, 46, 81] and ignored the majority of type changes that occur intra-project.

To provide an overview of the reasons for performing these popular TCPs, we studied the related research literature and developer documentation. In Table 5, we report 10 common reasons for performing type changes, along with some representative type changes extracted from our corpus, for each reason.

RQ6 Conclusion: 2.27% of the most popular type change patterns shared across projects account for 43% of type change instances. None of the top-10 most popular type change patterns are automated by current tools.

4 IMPLICATIONS

We present actionable, empirically-justified implications for four audiences: (i) researchers, (ii) tool builders and IDE designers, (iii) language and library designers, and (iv) developers and educators.

4.1 Researchers

R1. Foundations for Software Evolution (RQ1, 2 & 6) We found that type changes are more frequently applied than renames, but they are less studied. Previous studies on program transformations focused on refactorings like renames, moves and extractions [59, 60, 78]. We also observe that 41.6% of the type changes can potentially introduce breaking changes. Moreover, we find empirical evidence showing that when a library developer introduces a breaking change by performing a type change, the clients adapt to it by performing similar type changes. Our dataset [50] contains fine-grained information, including links to the exact lines of code in GitHub commits, where developers performed type changes and adapted the references to the program elements whose type changed. Such detailed information can help researchers to better design longitudinal studies to understand software evolution [12, 13], to perform more accurate API updates [16, 20, 22, 37], library migrations [2, 3, 46, 81], and automated program repairs [58, 75] that involve type changes.

R2. Naming Conventions (RQ3) In 54.85% of TCIs, the program element gets renamed too (e.g., `File file` → `Path filePath`). Our results also show that developers tend to rename elements more often when the source and target types have a *composition* or *sibling* relationship. Previous researchers [7] who studied renaming in depth, missed the opportunity to explore the impact of type changes on the renaming of program elements. The tools we developed can be used by researchers to further explore this relationship. Researchers [1, 9, 86] have developed techniques, which recommend an element's name based on its usage context. These techniques could be applied whenever developers perform type changes.

R3. Infer Type Mappings (RQ2 & 3) Our findings show that *cascade* type change is a frequently performed edit action, when developers adapt the references of variables whose type changed. This edit action applies a secondary type change, often involving different types than the primary type change. Moreover we observe that 41.6% type changes are applied on public elements, introducing binary and source code incompatibilities. Thus, in order to perform safe API updates or migrations, it is imperative for the current techniques to infer type mapping for performing the cascade type changes, in addition to inferring method mappings for primary type change.

R4. Support Parameterized Types (RQ2) The performed type changes frequently update the argument of a parameterized type. Current techniques [49, 53, 82] can modify the Parameterized type container (`Vector<String>` → `List<String>`) or replace Parameterized types with Simple types (`Function<Integer, Integer>` → `IntUnaryOperator`). However, they cannot adapt the program correctly when the type argument changes (e.g., `Map<String,Integer>` → `Map<String, Long>`). Performing this type change correctly requires inter-procedural points-to and escape analysis, which is not supported by any of the current techniques.

R5. Generalize Techniques for Sibling Types (RQ1 & 5) The most popular type changes are performed between sibling types that share a common super type (e.g., `java.util.List` → `java.util.Set`). These types represent similar concepts with some differences in their properties. Previously, researchers have solved specific instances of these type changes, such as `HashMap`→`ConcurrentHashMap` [23], `Vector` → `ArrayList` [82], replace `HashTable`→`HashMap` [53, 68]. However, these techniques hardcode the semantic differences between the sibling types. Our data highlights a need for more general techniques to encode the differences between the properties of the two types.

R6. Crowdsource Type Changes (RQ1 & 6) We found that type changes are highly repetitive, within individual commits (7.3 TCIs per commit) but also across multiple commits from distinct projects. This confirms the findings of others on the repetitiveness of code changes [61, 63, 64, 74, 75], and calls for new research on crowdsourcing type change mappings from previously applied type changes. Our dataset [50] could be used as a starting point.

4.2 Tool Builders and IDE Designers

T1. Automate Reference Adaptation (RQ1) Type change is a very commonly applied transformation. This highlights that IDEs should provide support for advanced composite refactorings, which perform a type change and adapt the code referring to the variable whose type changed. While current IDEs support refactorings like *Change Method Signature* [35] or type migration [45], these tools only update the declaration of the method or variables, but do not adapt the references. The *adaptation process* requires identifying (i) the method mappings between the source and target types to update the method call sites, and (ii) replacement rules that cover all cascade type changes (Listing 1).

However, better tool support for type changes is desperately needed. A survey of 420 developers [61] ranked type change as the most highly desired feature (among commonly applied transformations) for IDE automation. Moreover, Nishizono et al. [67] found that among other source code maintenance tasks, *Change Variable Type* requires the longest comprehension time.

T2. Support Selective Type Changes (RQ5) All existing tools that perform type changes [45, 49, 53, 82] follow a migration approach, where the type change patterns are exhaustively applied within a particular scope. However, we observed that in 61.71% of type changes, developers apply them *selectively* on an element, based on the context of code. The existing techniques should give the user more fine-grained control (other than specifying the scope)

Table 5: A few popular type change patterns

Type Change Pattern	#projects, #commits	#Instances	Known Reasons
`int` ⟶ `long`	(75, 623)	4,600	`Widening Primitive Conversion` [34]
`int` ⟶ `double`	(22, 47)	115	
`java.util.List` ⟶ `java.util.Set`	(68, 281)	742	`Different properties` [26–29, 31]
`java.util.LinkedList` ⟶ `java.util.DeQue`	(8, 8)	9	
`java.util.ArrayList` ⟶ `java.util.List`	(52, 152)	560	`Use Supertype Where Possible` [82]
`java.util.HashMap` ⟶ `java.util.Map`	(38, 112)	348	
`java.util.Map` ⟶ `java.util.concurrent.ConcurrentMap`	(37, 93)	193	`Concurrency` [23]
`int` ⟶ `java.util.concurrent.atomic.AtomicInteger`	(32, 65)	86	
`java.util.concurrent.atomic.AtomicLong` ⟶ `java.util.concurrent.atomic.LongAdder`	(11, 16)	227	`Performance` [25, 32, 33]
`java.lang.StringBuffer` ⟶ `java.lang.StringBuilder`	(38, 109)	280	
`java.util.List` ⟶ `com.google.common.collect.ImmutableList`	(18, 66)	145	`Immutability` [24, 28]
`java.util.Set` ⟶ `com.google.collect.ImmutableSet`	(9, 50)	95	
`java.lang.String` ⟶ `java.nio.file.Path`	(23, 56)	502	`Conceptual Types` [19]
`long` ⟶ `java.util.Date`	(2, 2)	9	
`org.apache.commons.logging.Log` ⟶ `org.slf4j.Logger`	(8, 123)	1,902	`Third Party library migration` [3, 46, 81]
`com.mongodb.BasicDBObject` ⟶ `org.bson.Document`	(3, 5)	45	
`java.util.Random` ⟶ `java.security.SecureRandom`	(6, 6)	8	`Security` [30]
`java.lang.String` ⟶ `java.security.Key`	(2,2)	2	
`org.joda.time.DateTime` ⟶ `java.time.ZonedDateTime`	(5, 6)	126	`Deprecation` [70]

on *where* a type change should be applied. For example, developers perform the type change `String`⟶`URL` [19] judiciously, rather than eradicating all usages of type `String` in the project.

T3. Support Internal Project Type Changes (RQ2) Developers most often perform custom type changes between types which are declared in the project itself (i.e., *Internal*). The most appropriate techniques for performing such custom transformations are through DSLs [8, 68], however researchers [11] found that text-based DSLs are awkward to use. More research is needed to make it easier to express custom type changes.

4.3 Language and Library Designers

L1. Understand Library Usage (All RQs) Language and library designers continuously evolve types. They enhance existing types, deprecate old types (e.g., `Vector`), introduce new types for new features (e.g., `java.util.Optional` for handling `null` values), or provide alternate types with more features (e.g., `java.util.Random`⟶`java.security.SecureRandom`). Our findings, the accompanying dataset [50], and the tools we developed, can help language and library designers to understand *what* types are most commonly used, misused, and underused, and *how* the clients adapt to new types. Thus, they can make informed and empirically-justified decisions on how to improve existing features or introduce new ones.

L2. Adopt Value Types (RQ6) We found 3,747 type changes which box or unbox primitive types (e.g., `int` to `java.lang.Integer` or `java.lang.Boolean` to `boolean`). This practice is widespread in 101 projects from our corpus. The proposed *value types* feature in Project Valhalla (JEP 169 [76]) could help eliminate these changes, by enabling developers to abstract over primitive types without the boxing overhead.

4.4 Software Developers and Educators

D1. Rich Educational Resources (RQ1, 2 & 6) Developers learn and educators teach new programming constructs through examples. Robillard and DeLine [73] study what makes large APIs hard to learn and conclude that one of the important factors is the lack of usage examples. We provide 50,640 real-world examples of type changes in 129 large and mature Java projects. Because developers might need to inspect the entire commit, we provide link to the exact line of code in the GitHub commits (e.g., see [15]).

5 THREATS TO VALIDITY

(1) **Internal Validity** The findings of our study depend on the accuracy of our tools to mine type changes from the commit history of the projects. We mitigate this threat by validating our tools. RefactoringMiner detects type changes with a high precision (99.7%) and recall (94.8%) and TypeFactMiner qualifies the type name with 98.7% precision. Moreover, the use of RefactoringMiner makes our analysis immune to the noise created by refactorings such as extract, rename or move program elements. While the edit patterns we mined capture most of the adaptations, we acknowledge that these are not exhaustive. For example, the new Java 8 features (e.g., `Streams`, `Optional` or `StringJoiner`) dissolve control structures (e.g., `if`, `for`) into a functional-style statement. In the future, we

plan to extend our mining technique to analyse and identify new patterns from such many-to-one statement mappings.
(2) **External Validity** We studied 129 projects on GitHub, from a wide range of application domains, making the results of the study *generalizable* to other projects in similar domains. However, a study of proprietary code-bases might reveal different results.
(3) **Verifiability** We release [47, 48, 50] all the developed tools, collected data, and Python scripts used for statistical analysis and generating the plots of the paper, so that the study is fully reproducible.

6 RELATED WORK

(1) ***Empirical studies on type changes***: Previous work has studied type changes from the vantage point of higher-level maintenance and evolution tasks, such as API update and library migration, for decades [14]. Dig et al. [22] and Cossette et al. [16] performed retroactive studies into the presence and nature of the incompatibilities between API versions. Dietrich et al. [20] studied the risk of introducing runtime failures due to API updates, and McDonnell et al. [56] and Hora et al. [42] have studied the API evolution and adoption for the Android and Pharo ecosystems, respectively. Kula et al. [52] who studied how extensively developers update their libraries, highlighted that most systems keep their dependencies outdated. Similarly, researchers [46, 81] studied the practice of library migration in the Java ecosystem. Teyton et al. [81] studied the practice of library migration in Java to understand how frequently, when, and why they are performed. Kabinna et al. performed a case study on the practice of logging library migration [46]. In contrast, we study the practice of type change as a whole (including *Internal* and *Jdk* changes), by answering six broad research questions. Our longitudinal study of a large corpus helps us gain a deep understanding of the current gaps in research and tooling for type changes.
(2) ***Extracting Change Patterns***: Numerous approaches have been developed to infer properties of APIs, intended to guide their use by developers [72]. Previous work [41, 80, 89] proposed tools, which analyze the changes applied by library developers and recommend adaptations to the clients, when they update the library version. Nguyen et al. [65] proposed advanced graph based techniques, which assist developers to perform library updates by learning from examples. Similarly, Kim et al. [51] propose an approach to automatically discover and represent systematic changes as logic rules. Researchers [3, 39, 87] focus on mining method mappings between two API versions or libraries and helping clients to adapt the code to different libraries or versions. Researchers developed tools, such as LASE [57], Genesis [54] and Refazer [74] that synthesize transformations from examples. Recently, Fazzini et al. [37] and Xu et al. [90] developed techniques that mine adaptation examples from source code history and adapt the client code to the API Update. Recently, researchers proposed advanced tools to mine source code histories of projects to generate transformations for reasons, such as repairing bugs [10], removing bad style and performance bugs [75], fixing compilation errors [58]. These tools often mine transformation patterns which involve type changes like `String`⟶`Path`, `String`⟶`StringBuilder`, or `ImmutableList`⟶`ImmutableSet`. In contrast, our study does not analyze type changes in the context of a particular higher level software evolution task, but rather investigates all type changes performed in practice, providing deeper insights to facilitate further research on software evolution (implications R1, R3 & R6).
(3) ***Transformation tools performing type-related changes***: Researchers have developed tools specifically to perform type-related changes safely, such as a class library migration tool which uses type constraints [8], T2R ultra-large-scale type migration tool for specializing functional interfaces [49], and SWIN [53] which performs safe API updates based on Twinning [68]. These tools perform type migration (i.e., they exhaustively apply a type change in entire code base) using the changes that the user expresses with a text-based DSL. While in our study, we try to understand how representative are the type changes that these tools can perform in the real world. We uncover some blind spots, so that researchers and tool builders can make their tools more (i) safe (implications T1, R3 & R4), (ii) practical and applicable in more contexts (implications R2, T2 & T3), and (iii) extensible (implications R5 & R6).

7 CONCLUSION

This paper presents a fine-grained and large-scale empirical study to understand the type changes performed in 129 open source Java projects. To perform this study, we developed and validated tools to statically mine type changes and their subsequent code adaptations from the commit history of Java projects. We employed these tools to create an extensive and reliable data set of 297,543 type changes to answer six questions about the practice of type change. Some of our key surprising findings are:

(1) Type changes are *more common* than renaming.
(2) To adapt the code, developers often perform secondary *cascade* type changes, which are different than the primary type change.
(3) Developers often rename elements, when changing their type.
(4) Type changes between types having a *composition* relation need more adaptation effort than those with *inheritance* relation.
(5) Developers more often perform type changes on `public` program elements rather than `private`, `package-private`, and `protected` elements, introducing potential breaking changes.
(6) Although the raw number of type changes on `private` elements is less than the number of type changes on `public` elements, developers tend to change more often the types of `private` elements compared to `public` ones, indicating possible considerations for preserving backward compatibility.
(7) Developers most often perform *selective* type changes, rather than *migrating* types in the entire project.
(8) Type *migrations* are most commonly performed on *internal* project types.

The results presented in this study call for more intelligent tool support and further research to assist the developers by automating the task of type changes. We hope that this paper motivates the community to advance the science and tooling for type change automation.

8 ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers of ICSE'2020 and FSE'2020 for their insightful and constructive feedback for improving the paper. This research was partially supported by NSERC grant RGPIN2018-05095 and NSF grant CCF-1553741.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[2] Hussein Alrubaye, Deema AlShoaibi, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. How Does API Migration Impact Software Quality and Comprehension? An Empirical Study. (Jul 2019). https://arxiv.org/abs/1907.07724

[3] Hussein Alrubaye and Mohamed Wiem Mkaouer. 2018. Automating the Detection of Third-party Java Library Migration at the Function Level. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18)*. IBM Corp., Riverton, NJ, USA, 60–71. http://dl.acm.org/citation.cfm?id=3291291.3291299

[4] Apache. 2019. effective-pom. https://maven.apache.org/plugins/maven-help-plugin/effective-pom-mojo.html.

[5] Apache. 2019. Netbeans Refactoring. http://wiki.netbeans.org/Refactoring.

[6] Apache. 2019. Visual Studio-Refactor code. https://docs.microsoft.com/en-us/visualstudio/ide/refactoring-in-visual-studio?view=vs-2019.

[7] Venera Arnaoudova, Laleh M. Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2014. REPENT: Analyzing the nature of identifier renamings. *Software Engineering, IEEE Transactions on* 40 (05 2014), 502–532. https://doi.org/10.1109/TSE.2014.2312942

[8] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring Support for Class Library Migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 265–279. https://doi.org/10.1145/1094811.1094832

[9] Rohan Bavishi, Michael Pradel, and Koushik Sen. 2018. Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. (Aug 2018). https://arxiv.org/abs/1809.05193

[10] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

[11] Marat Boshernitsan, Susan L. Graham, Susan L. Graham, and Marti A. Hearst. 2007. Aligning Development Tools with the Way Programmers Think About Code Changes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 567–576. https://doi.org/10.1145/1240624.1240715

[12] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 465–475. https://doi.org/10.1145/3106237.3106259

[13] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How Does Refactoring Affect Internal Quality Attributes?: A Multi-project Study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. ACM, New York, NY, USA, 74–83. https://doi.org/10.1145/3131151.3131171

[14] Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes. In *Proceedings of the International Conference on Software Maintenance*. 359–368. https://doi.org/10.1109/ICSM.1996.565039

[15] Guacamole Client. 2011. commit with type change. https://tinyurl.com/yx2npj8g

[16] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 55, 11 pages. https://doi.org/10.1145/2393596.2393661

[17] Barthélémy Dagenais and Laurie Hendren. 2008. Enabling Static Analysis for Partial Java Programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 313–328. https://doi.org/10.1145/1449764.1449790

[18] Barthélémy Dagenais and Martin P. Robillard. 2009. SemDiff: Analysis and recommendation support for API evolution. In *Proceedings of the IEEE 31st International Conference on Software Engineering*. 599–602. https://doi.org/10.1109/ICSE.2009.5070565

[19] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym: Using Names to Refine Types. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 107–117. https://doi.org/10.1145/3236024.3236042

[20] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE Computer Society, Washington, DC, USA, 64–73. https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[21] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, Berlin, Heidelberg, 404–428. https://doi.org/10.1007/11785477_24

[22] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (March 2006), 83–107. https://doi.org/10.1002/smr.328

[23] Danny Dig, John Marrero, and Michael D. Ernst. 2009. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 397–407. https://doi.org/10.1109/ICSE.2009.5070539

[24] Guava Documentation. 2019. ImmutableList. https://guava.dev/releases/23.4-jre/api/docs/com/google/common/collect/ImmutableList.html.

[25] Java Platform Documentation. 2019. Autoboxing and unboxing. https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html.

[26] Java Platform Documentation. 2019. DeQue. https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html.

[27] Java Platform Documentation. 2019. LinkedList. https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html.

[28] Java Platform Documentation. 2019. List. https://docs.oracle.com/javase/9/docs/api/java/util/List.html.

[29] Java Platform Documentation. 2019. Map. https://docs.oracle.com/javase/8/docs/api/java/util/Map.html.

[30] Java Platform Documentation. 2019. SecureRandom. https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html.

[31] Java Platform Documentation. 2019. Set. https://docs.oracle.com/javase/9/docs/api/java/util/Set.html.

[32] Java Platform Documentation. 2019. StringBuffer. https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html.

[33] Java Platform Documentation. 2019. StringBuilder. https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html.

[34] Java Platform Documentation. 2019. Widening Primitive Conversion. https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.2.

[35] Eclipse. 2019. Refactoring Actions. https://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm.

[36] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[37] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 204–215. https://doi.org/10.1145/3293882.3330571

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[39] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. 2013. Inferring Likely Mappings Between APIs. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 82–91. http://dl.acm.org/citation.cfm?id=2486788.2486800

[40] Google. 2011. Error Prone. https://github.com/google/error-prone

[41] Johannes Henkel and Amer Diwan. 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering*. 274–283. https://doi.org/10.1145/1062455.1062512

[42] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. 2015. How do developers react to API evolution? The Pharo ecosystem case. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 251–260. https://doi.org/10.1109/ICSM.2015.7332471

[43] Eclipse JDT. 2019. Type. https://help.eclipse.org/luna/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Type.html.

[44] JetBrains. 2019. IntelliJ - Refactoring Code. https://www.jetbrains.com/help/idea/refactoring-source-code.html.

[45] JetBrains. 2019. Type Migration. https://www.jetbrains.com/help/idea/type-migration.html.

[46] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging Library Migrations: A Case Study for the Apache Software Foundation Projects. In *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories*. 154–164. https://doi.ieeecomputersociety.org/10.1109/MSR.2016.025

[47] Ameya Ketkar. 2020. ameyaKetkar/TypeChangeMiner: Type change miner. https://doi.org/10.5281/zenodo.3906493

[48] Ameya Ketkar. 2020. Type Change Study Data collected. https://doi.org/10.5281/zenodo.3906503

[49] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. 2019. Type Migration in Ultra-large-scale Codebases. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1142–1153. https://doi.org/10.1109/ICSE.2019.00117

[50] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2019. Type Facts Companion website. http://changetype.s3-website.us-east-2.amazonaws.com/docs/.

[51] Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson. 2013. Identifying and Summarizing Systematic Code Changes via Rule Inference. *IEEE Transactions on Software Engineering* 39, 1 (Jan 2013), 45–62. https://doi.org/10.1109/TSE.2012.16

[52] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? *Empirical Softw. Engg.* 23, 1 (Feb. 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[53] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. 2015. SWIN: Towards Type-Safe Java Program Adaptation Between APIs. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation (PEPM '15)*. ACM, New York, NY, USA, 91–102. https://doi.org/10.1145/2678015.2682534

[54] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[55] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 85 (Oct. 2017), 31 pages. https://doi.org/10.1145/3133909

[56] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the IEEE International Conference on Software Maintenance*. 70–79. https://doi.org/10.1109/ICSM.2013.18

[57] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 502–511. http://dl.acm.org/citation.cfm?id=2486788.2486855

[58] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 925–936. https://doi.org/10.1145/3338906.3340455

[59] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[60] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 552–576. https://doi.org/10.1007/978-3-642-39038-8_23

[61] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining Fine-grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 803–813. https://doi.org/10.1145/2568225.2568317

[62] Netbeans. 2011. Netbeans - Jackpot wiki. http://wiki.netbeans.org/Jackpot

[63] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API Code Recommendation Using Statistical Learning from Fine-grained Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2950290.2950333

[64] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-based Mining of In-the-wild, Fine-grained, Semantic Code Change Patterns. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 819–830. https://doi.org/10.1109/ICSE.2019.00089

[65] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-based Approach to API Usage Adaptation. *SIGPLAN Not.* 45, 10 (Oct. 2010), 302–321. https://doi.org/10.1145/1932682.1869486

[66] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[67] Kazuki Nishizono, Shuji Morisaki, Rodrigo Vivanco, and Kenichi Matsumoto. 2011. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - An empirical study with industry practitioners. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*. 473–481. https://doi.org/10.1109/ICSM.2011.6080814

[68] Marius Nita and David Notkin. 2010. Using Twinning to Adapt Programs to Alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 205–214. https://doi.org/10.1145/1806799.1806832

[69] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *Proceedings of the IEEE International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609577

[70] The Joda project. 2019. Joda-Time. https://www.joda.org/joda-time/.

[71] Juergen Rilling and Tuomas Klemola. 2003. Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC '03)*. IEEE Computer Society, USA, 115–124. https://doi.org/10.1109/WPC.2003.1199195

[72] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39, 5 (May 2013), 613–637. https://doi.org/10.1109/TSE.2012.63

[73] Martin P. Robillard and Robert Deline. 2011. A Field Study of API Learning Obstacles. *Empirical Softw. Engg.* 16, 6 (Dec. 2011), 703–732. https://doi.org/10.1007/s10664-010-9150-8

[74] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

[75] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. (2018). http://arxiv.org/abs/1803.03806

[76] John Rose. 2019. Value Objects. https://openjdk.java.net/jeps/169.

[77] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2020. RefDiff 2.0: A Multi-language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* (2020), 1–17. https://doi.org/10.1109/TSE.2020.2968072

[78] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[79] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 269–279. https://doi.org/10.1109/MSR.2017.14

[80] Kunal Taneja, Danny Dig, and Tao Xie. 2007. Automated Detection of Api Refactorings in Libraries. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 377–380. https://doi.org/10.1145/1321631.1321688

[81] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A Study of Library Migrations in Java. *J. Softw. Evol. Process* 26, 11 (Nov. 2014), 1030–1052. https://doi.org/10.1002/smr.1660

[82] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring Using Type Constraints. *ACM Trans. Program. Lang. Syst.* 33, 3, Article 9 (May 2011), 47 pages. https://doi.org/10.1145/1961204.1961205

[83] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 1–21. https://doi.org/10.1109/TSE.2020.3007722

[84] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[85] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017), 11. https://doi.org/10.1002/smr.1838

[86] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering Clear, Natural Identifiers from Obfuscated JS Names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 683–693. https://doi.org/10.1145/3106237.3106289

[87] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: A Hybrid Approach to Identify Framework Evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 325–334. https://doi.org/10.1145/1806799.1806848

[88] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/1101908.1101919

[89] Zhenchang Xing and Eleni Stroulia. 2007. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (Dec 2007), 818–836. https://doi.org/10.1109/TSE.2007.70747

[90] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. IEEE Press, Piscataway, NJ, USA, 335–346. https://doi.org/10.1109/ICPC.2019.00052