IntelliTC: Automating Type Changes in IntelliJ IDEA

Oleg Smirnov
JetBrains Research
St Petersburg University
Russia
oleg.smirnov@jetbrains.com

Ameya Ketkar*
Uber Technologies Inc.
USA
ketkara@uber.com

Timofey Bryksin
JetBrains Research
HSE University
Russia
timofey.bryksin@jetbrains.com

Nikolaos Tsantalis
Concordia University
Canada
nikolaos.tsantalis@concordia.ca

Danny Dig
University of Colorado Boulder
USA
danny.dig@colorado.edu

ABSTRACT

Developers often change types of program elements. Such refactoring often involves updating not only the type of the element itself, but also the API of all type-dependent references in the code, thus it is tedious and time-consuming. Despite type changes being more frequent than renamings, just a few current IDE tools provide partially-automated support only for a small set of hard-coded types. Researchers have recently proposed a data-driven approach to inferring API rewrite rules for type change patterns in Java using code commits history. In this paper, we build upon these recent advances and introduce IntelliTC — a tool to perform Java type change refactoring. We implemented it as a plugin for IntelliJ IDEA, a popular Java IDE developed by JetBrains. We present 3 different ways of providing support for such a refactoring from the standpoint of the user experience: Classic mode, Suggested Refactoring, and Inspection mode. To evaluate these modalities of using IntelliTC, we surveyed 22 experienced software developers. They positively rated the usefulness of the tool.

The source code and distribution of the plugin are available on GitHub: https://github.com/JetBrains-Research/data-driven-type-migration. A demonstration video is available on YouTube: https://youtu.be/pdcfvADA1PY.

ACM Reference Format:
Oleg Smirnov, Ameya Ketkar, Timofey Bryksin, Nikolaos Tsantalis, and Danny Dig. 2022. IntelliTC: Automating Type Changes in IntelliJ IDEA. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516851

1 INTRODUCTION

As the program evolves, developers change the type of variables and methods for several reasons, such as library migration (*e.g.,* `org.apache.commons.logging.Log` to `org.slf4j.Logger`), security (*e.g.,* `java.util.Random` to `java.security.SecureRandom`), or performance (`StringBuffer` to `StringBuilder`). From a developer's perspective, such *type change* refactoring is much more complicated and tedious than just changing the type of some identifier. To perform a type change, developers update the declared type of a program element (*e.g.,* local variable, parameter, field, or method) 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 access, a developer might perform a *series* of type changes to propagate type constraints for the new type.

In our empirical study investigating the practice of type changes in the real world, Ketkar et al. [20] observed that type changes are performed more often than popular refactorings like *Rename*. In contrast to other refactoring types that are heavily automated by all popular integrated development environments (IDEs), no IDE actively automates type changes; thus developers have to perform most of them manually. The state-of-the-practice type change automation tool provided by IntelliJ IDEA [12] is only applicable to a small set (around ten) of hard-coded type changes that eliminate the use of deprecated types from the Guava library or pre-Java 8 APIs, and it does not allow developers to express and adapt *custom* type changes. While state-of-the-art type migration tools [2, 18, 25] are more applicable from the aspect of allowing users to automate custom type changes, these tools are either not supported or are distributed as stand-alone applications depending upon specific static analysis frameworks like Google's Error Prone [11] or clang project's LibTooling infrastructure [23]. This greatly limits their applicability and usefulness in practice, because (1) using these external tools breaks developer workflows while working in an IDE [16], (2) not all developers use (or can use) these specific static analysis frameworks, and (3) the user has to *handcraft* the transformation specifications required to perform the custom type change.

In this paper, we introduce IntelliTC, a developer-friendly IntelliJ IDEA plugin for automating type changes, and explore its UI/UX aspects. IntelliTC leverages the underlying IntelliJ's Type Migration framework [14] to allow developers to express type changes as rewrite rules over Java expressions using IntelliJ's Structural-Search-and-Replace templates [13]. IntelliTC provides three modes to automate type changes in multiple developer workflows. For instance, in the *Classic* mode IntelliTC has to be manually invoked (similar to *Rename* refactoring), while in the *Inspection* mode IntelliTC recommends type changes.

In our accompanying paper [19], we describe TC-Infer — a tool that automatically infers API rewrite rules required to perform type

*Ameya Ketkar performed this work as part of his PhD at Oregon State University.

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05.
https://doi.org/10.1145/3510454.3516851

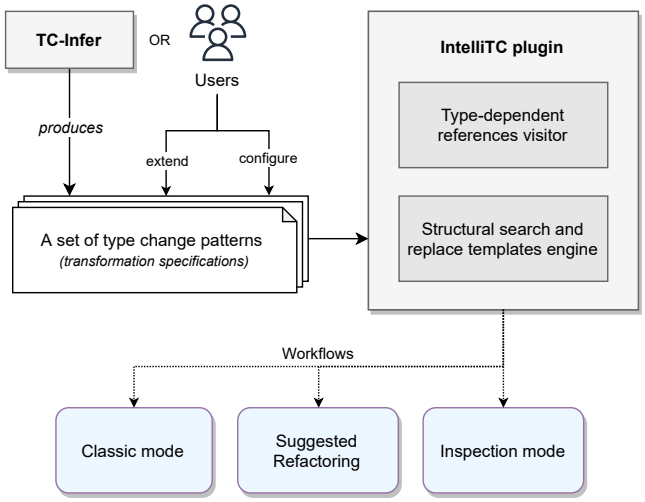

Figure 1: The pipeline behind IntelliTC.

```
{
  "From": "java.io.File",
  "To": "java.nio.file.Path",
  "ID": 1,
  "Priority": 2,
  "Mode": "Suggested Refactoring",
  "Rules": [
    {
      "Before": "new File($1$, $2$)",
      "After": "$1$.resolve($2$)"
    },
    {
      "Before": "$1$.exists()",
      "After": "Files.exists($1$)",
    },
    {
      "Before": "$1$.toPath()",
      "After": "$1$"
    },
    ...
  ]
}
```

Figure 2: A fragment of the "`File→Path`" type change pattern. In the "Rules" section, the template variable `$1$` is responsible for matching the *root* element of the type change, whereas the template variable `$2$` is used for matching any other AST expression node greedily, except the root.

changes, by analyzing the version history of Java projects. This reduces the burden on the developers, since they no longer have to handcraft the rules for popular type changes. We identified 59 most popular and useful type changes (and the associated rewrite rules) reported by TC-Infer. We then applied IntelliTC (a plugin we developed in this work) to replicate 3,060 instances of these 59 type changes encountered in commit histories. Our results showed that IntelliTC has 99.2% accuracy at automating type changes.

We have also surveyed 22 experienced software developers to evaluate the potential usefulness of different features that IntelliTC offers. The participants have positively rated the chosen ways of UI envisioning, and confirmed the usefulness of the idea of employing popular type changes from open-source projects to improve built-in IDE refactoring capabilities.

The instructions for downloading, installing, and using the plugin are available online.[1]

2 INTELLITC

2.1 Overview

The high-level overview of the plugin is presented in Figure 1. The plugin allows developers to automatically perform a set of preconfigured type changes (inferred by TC-Infer) and to configure any custom type changes by themselves. IntelliTC's core components are implemented with the use of the IntelliJ Platform SDK.[2] Using the plugin, the developer can follow 3 different workflows. First, IntelliTC is manually invoked at a variable or a method (the *root* element of the transformation) to automatically perform the desired type change. Second, IntelliTC tracks the developer's activity in the code editor to understand their intent and appropriately *suggests* a type change refactoring. Third, IntelliTC recommends certain type changes by *inspecting* the code.

[1] IntelliTC: https://type-change.github.io/
[2] The IntelliJ Platform: https://plugins.jetbrains.com/docs/intellij/intellij-platform.html

2.2 Transformation Specifications

Developers can edit existing rules, add new rules to existing type change patterns, or even add new type change patterns by updating the *transformation specifications* JSON exposed by IntelliTC via the *Settings* tab of IntelliJ IDEA, as shown in Figure 2. The schema followed by this JSON is analogous to the output *transformation specifications* produced by TC-Infer, where each type change pattern (*i.e.,* `(source-type, target-type)`) is associated with a set of rewrite rules over Java statements (and expressions). One of these associated rewrite rules will be applied to adapt each reference when the type change is performed.

In addition to this, IntelliTC also allows users to rank the type changes as they will appear in the UI (*Priority* in Figure 2), and specify how to surface the type change suggestion (*Classic mode*, *Inspection mode*, *Suggested Refactoring* — described in Section 2.4). By default, all type change patterns specified in this JSON can be applied only by manually invoking the plugin and are not automatically surfaced.

In IntelliTC, we include a set of manually vetted rewrite rules for 59 popular and useful type change patterns that we constructed as a part of the evaluation of TC-Infer (described in our accompanying paper's RQ 3 [19]). There, we investigated a corpus of 129 large, mature, and diverse Java projects and identified 40,865 commits where type changes were performed. We then analysed these commits and found 605 *popular* type change patterns that were performed in more than one unique project. We applied TC-Infer on these commits to infer the associated rewrite rules for these 605 type change patterns. From these patterns, we manually selected 59 type change patterns to be used as an *initial* input for IntelliTC, based on their popularity and relevance for the end-use developers (we only considered the type changes between built-in JDK types). We also automatically calculated *Priority* based on the number of commits where type change was performed, and manually labelled

Figure 3: An example usage of the IntelliTC plugin applied upon a code snippet from the Apache Flink project. We submitted a pull request with 9 type changes (including this) aimed at eliminating such misuse, and it was accepted by the maintainers [19].

the suggestion level for these 59 type change patterns (field *Mode* in Figure 2).

2.3 Implementation

To handle the search of type-dependent references in Java code, we decided to leverage the capabilities of the existing IntelliJ's Type Migration framework [14]. IntelliTC reuses its core visitor components to search for all the candidate source code locations for update (by inter-procedural analysis across type constraints) and then performs the actual update by executing IntelliJ's Structural-Search-and-Replace (SSR) templates [13]. Such templates allow matching the code fragments in a regex-like manner, considering their tree structure and also leaving *holes* (like `$1$` and `$2$` in Figure 2) for tree nodes, which is especially useful when matching identifiers. Using SSR lowers the barrier of entry for new users of the tool, since SSR is a well-documented part of the IntelliJ Platform [13].

IntelliTC lets users define a *scope* for reference search, including *Local* scope (the enclosing method of the identifier which type is being changed), *Current File/Class* scope, and *Global/Project* scope. The plugin also supports *undoing* the type change.

Under the hood, IntelliTC runs the visitors provided by IntelliJ's Type Migration framework, and updates the type-dependent references with the rewrite rules contained in the chosen type change pattern. For each reference, the final rewrite rule is chosen by the largest number of matched code tokens between the reference itself (or its parent in the AST) and the before-part of some rewrite rule.

2.4 Modalities

IntelliTC operates in three different modes:

2.4.1 Classic mode. In this mode, the plugin operates as a general *code intention*,[3] providing the developers with the ability to apply a type change refactoring only when it is invoked directly from the context menu of some type element in the code. For instance, as it is shown in Figure 3, developers can invoke intention action for the type of the field `validation` (which defines a *root* element here), aiming to change it from `Function<String, Boolean>` to `Predicate<String>`. They can specify the type change pattern along with a search scope in the shown dialog box. If IntelliTC cannot update any references of the *root* element, it will show the *Tool Window* with the failed usages. This allows developers to fix the problems manually or use built-in quick-fixes.

2.4.2 Suggested Refactoring. Previous researchers [8, 10] observed that *discoverability* and *late awareness* led to the underuse of refactoring tools. To counter this problem, we leverage IntelliJ's *Suggested Refactoring* mechanism. In this mode, a corresponding type change refactoring is suggested by IntelliTC when the user manually changes the type of some element in Java code. The plugin tracks such changes in the document model and renders the button in the left-side panel of the code editor, allowing the user to click it, configure the necessary search scope, and run the refactoring.

Note that not all type changes are applicable in each context (*e.g.*, `String` to `Pattern`, or `String` to `Path`), and receiving spurious suggestions from the IDE could confuse developers. Thus, we decided to make Suggested Refactoring available only for *isomorphic* types (that are interchangeable), like `File` and `Path`, or `Date` and `LocalDate`. Currently, the user needs to manually label the isomorphic types (via the `"Mode"` field in Figure 2) in the input JSON. We believe that automatically detecting isomorphic type changes and suggesting them is a challenging yet promising direction for future work.

2.4.3 Inspection mode. The last but not least is the mode in which IntelliTC runs as a *code inspection*.[4] Code inspections are performed automatically by the IntelliJ Platform's engine in the background, and are useful for detecting possible problems with the code. IntelliTC's *Inspection mode* leverages this interface to provide quick fixes involving type changes. Currently, IntelliTC promotes the clean-code recommendations from Effective Java [4], like eliminating misuses of Java 8 functional interfaces (*e.g.*, `Function<T, Boolean>`→`Predicate<T>`, see Figure 3). IntelliTC highlights such occurrences of misused types in the program and provides the appropriate intention actions to replace them.

3 EVALUATION

Our previous work [19] provides an in-depth empirical evaluation for TC-Infer and shows the merit of using a data-driven approach. For this demo paper, we complement the previous thorough evaluation with a survey to determine how developers apply type changes

[3] Intention Actions: https://www.jetbrains.com/help/idea/intention-actions.html

[4] Code Inspections: https://www.jetbrains.com/help/idea/code-inspection.html

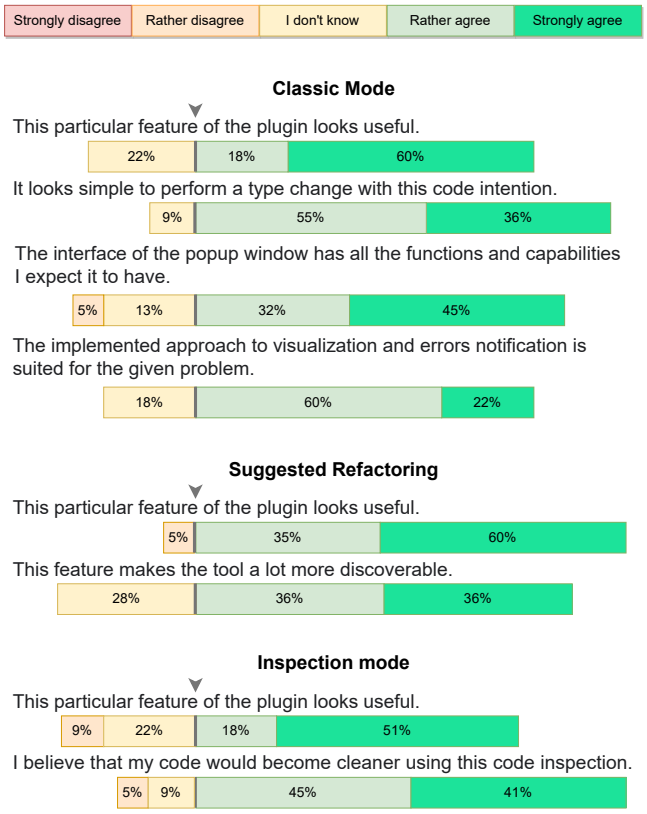

Figure 4: The results of the conducted preliminary survey.

in their everyday work, and asked them to evaluate the usefulness of the IntelliTC for different usage modalities. We intentionally designed the study such that participants are able to see a short and feature-focused demo for each mode of the plugin, instead of forcing them to install and run it. The previous research [7, 17] has shown that such method could be helpful to evaluate the core concepts and ideas behind the tool, whereas high-fidelity prototypes might distract survey participants with a lot of implementation details.

We have questioned 22 qualified software developers with 2 to 5 years of professional experience on average (in particular, four of them had been working with Java for 10+ years). All the respondents confirmed that they frequently use IntelliJ IDEA's code intentions, inspections, and automated refactoring features. Even so, almost half of the participants had also mentioned that in everyday practice, their automated refactoring experience comes down to some simple scenarios, such as renames. Twelve people stated their awareness of existing Type Migration refactoring, but only *3* of them had actually used it. It should also be noted that our evaluation is limited by considering the developers who *already* use refactoring tools, thus assessment of people who have no experience with such tools could affect the final results.

We asked the respondents to evaluate the usefulness and compare the three different modes of providing type change refactoring (see Section 2) in the plugin. We have used Likert-type [5] questions and followed the best-practices from existing usability studies [3].

Figure 4 shows the results. Notice that the participants have highly positive attitudes towards the usefulness and the interfaces of IntelliTC. We also asked them to compare the existing plugin modes, and *Suggested Refactoring* was chosen as the most useful by 80% of developers. However, approximately half of all participants expressed their convictions to receive such suggestions for *isomorphic* types only (see Section 2.4.2).

The developers have also left constructive feedback on how they envision improving IntelliTC. They expressed their need to see a preview of the refactoring, and also to receive additional motivation for the recommended type changes in the *Inspection* mode. We plan to implement these features as a part of our future work.

We are also grateful to receive such inspiring positive reviews:

> "It really does save a lot of time and energy, as a String can be used in lots of places in code. I usually coped with changing Strings to Patterns manually, which was really boring and time consuming."

> "If I knew that this exists, I would definitely use it."

4 RELATED WORK

Previous researchers and tool-builders have proposed several frameworks [2, 18, 25] to address the problem of applying custom type changes in source code. These approaches require API transformation specifications to be manually encoded in the corresponding *domain specific language*. However, Kim et al. [21] have shown that encoding refactorings via DSL might be overwhelming, and such activity has a steep learning curve. To overcome this barrier of using type migration tools, we proposed TC-Infer in our accompanying paper [19] to automatically infer rewrite rules for common type changes, while IntelliTC focuses on automating them in the IDE in a developer-friendly manner.

From the standpoint of production-ready tools, there is no actual support for *custom* type change refactoring or class library migration in the current IDEs. IntelliJ IDEA [12], being the most popular IDE for Java developers in 2021 [15], provides *type migration* refactoring only for a small set of pre-defined types, however it does not support custom API rewrite rules. To the best of our knowledge, other popular IDEs for Java (such as Eclipse [9], Visual Studio Code [24], NetBeans [1]) do not provide any functionality for performing type changes. IntelliTC allows developers to define custom type change rules using the *Structural Search and Replace* templates and automates these for the developer in the IDE.

Previously researchers have also developed tools to perform library migrations or library updates at the source code level: (1) Xi et al. [26] proposed a tool called DAAMT which automates the migration of deprecated Java APIs based on its replacements from the documentation, (2) Lamothe et al. [22] proposed a way to migrate deprecated Android APIs by learning from code examples, (3) Collie et al. [6] employed a probabilistic program synthesis to tackle cases when there is no prior knowledge of the target library API usage. In contrast to these works, this paper largely focuses on the *user experience* aspect of the tool, and addresses the problem of effectively performing a *type change* refactoring in an IDE.

5 CONCLUSION

In this paper, we present IntelliTC — an IntelliJ IDEA plugin for automating type changes. Our approach uses custom API transformation specifications, which are automatically mined and inferred from the history of code changes by TC-Infer [19], or added and tweaked by the users themselves. We have presented three ways for providing type change refactoring opportunity to the developer from the standpoint of user experience and conducted a preliminary evaluation of its potential usefulness with 22 software developers. IntelliTC was warmly received by the participants of the study, and we plan to continue improving it based on their feedback.

6 ACKNOWLEDGEMENTS

We would like to thank Zarina Kurbatova for her help with the technical details of working with the IntelliJ Platform, Souti Chattopadhyay for helping us design the evaluation, Yaroslav Golubev and the reviewers for their constructive feedback to improve the work. We also thank our survey participants for their insightful answers.

REFERENCES

[1] Apache. 1997. *NetBeans.* https://netbeans.apache.org/
[2] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring support for class library migration. *ACM SIGPLAN Notices* 40, 10 (2005), 265–279.
[3] Carol M Barnum. 2010. *Usability testing essentials.* Elsevier.
[4] Joshua Bloch. 2008. *Effective java.* Addison-Wesley Professional.
[5] Harry N Boone and Deborah A Boone. 2012. Analyzing likert data. *Journal of extension* 50, 2 (2012), 1–5.
[6] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael FP O'Boyle. 2020. M3: Semantic api migrations. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 90–102.
[7] Scott Davidoff, Min Kyung Lee, Anind K Dey, and John Zimmerman. 2007. Rapidly exploring application design through speed dating. In *International Conference on Ubiquitous Computing.* Springer, 429–446.
[8] S. R. Foster, W. G. Griswold, and S. Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *ICSE.* 222–232. https://doi.org/10.1109/ICSE.2012.6227191
[9] Eclipse Foundation. 2001. *Eclipse.* https://www.eclipse.org/
[10] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling Manual and Automatic Refactoring. In *ICSE* (Zurich, Switzerland) *(ICSE '12).* IEEE Press, 211–221.
[11] Google. 2011. *Error Prone.* https://github.com/google/error-prone
[12] JetBrains. 2021. *IntelliJ IDEA official website.* https://www.jetbrains.com/idea/
[13] JetBrains. 2021. *IntelliJ's Structural Search and Replace.* https://www.jetbrains.com/help/idea/structural-search-and-replace.html
[14] JetBrains. 2021. *IntelliJ's Type Migration framework.* https://www.jetbrains.com/help/idea/type-migration.html
[15] JetBrains. 2021. *The State of Java Developer Ecosystem.* https://www.jetbrains.com/lp/devecosystem-2021/java/
[16] Philip M Johnson. 2001. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. In *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications.* Citeseer.
[17] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology.* 140–151.
[18] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. 2019. Type migration in ultra-large-scale codebases. In *ICSE.* IEEE, 1142–1153.
[19] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and Applying Type Changes. In *44th International Conference on Software Engineering (ICSE '22)* (Pittsburgh, United States) *(ICSE '22).* ACM. https://doi.org/10.1145/3510003.3510115
[20] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. Understanding type changes in java. In *FSE.* 629–641.
[21] Jongwook Kim, Don Batory, and Danny Dig. 2015. Scripting parametric refactorings in Java to retrofit design patterns. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 211–220. https://doi.org/10.1109/ICSM.2015.7332467
[22] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering* (2020).
[23] LLVM. 2007. *Clang-tidy.* https://clang.llvm.org/extra/clang-tidy/
[24] Microsoft. 2015. *Visual Studio Code.* https://code.visualstudio.com/
[25] Hyrum K Wright. 2020. Incremental type migration using type algebra. In *ICSME.* IEEE, 756–765.
[26] Yaoguo Xi, Liwei Shen, Yukun Gui, and Wenyun Zhao. 2019. Migrating deprecated api to documented replacement: Patterns and tool. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware.* 1–10.