Refactoring in the Eclipse JDT: Past, Present, and Future

Robert M. Fuhrer[1], Adam Kieżun[2], Markus Keller[3]

[1] IBM T.J. Watson Research Center, rfuhrer@watson.ibm.com

[2] MIT Computer Science and AI Lab, akiezun@mit.edu

[3] Eclipse Refactoring Engine, IBM, markus_keller@ch.ibm.com

Abstract

In this position paper, we present the history, the present and our view of the future of refactoring support in Eclipse.

Past and Present

Eclipse was among the first IDEs to help bring refactoring to the mainstream developer. Eclipse version 1.0 included several highly useful Java refactorings, which are nowadays staple tools in most Java developers' toolbox. These included Extract Method, Rename and Move. Eclipse 2.0 added statement-level refactorings such as Extract and Inline Local Variable and also higher-level ones, e.g., Change Method Signature, and Encapsulate Field. Some refactorings, such as Rename, offer great leverage because of the potential scale of the changes they perform automatically. Others, like Extract Method, are more local in scope, but relieve the developer from performing the analysis required to ensure that program behavior is unaffected. In both cases, the developer benefits from reduction of a complex and numerous changes to a single operation. This helps to maintain his focus on the big picture. Moreover, the ability to quickly roll back the changes enables exploration of design possibilities more easily, and without fear of irreparable damage to the code base.

Eclipse 2.1 included several type-oriented refactorings such as Extract Interface and Generalize Type, that address the problems of both scale and analytic complexity. These used a common analysis framework [8] based on theoretical work from Palsberget al. [6] for expressing the system of constraints that ensure the type-correctness of the resulting program. Such frameworks are important because they speed the development of entire families of refactorings, e.g., [7]. Our belief is that the incorporation of reusable and extensible frameworks for the various classical static analyses (type, pointer, data flow) into Eclipse will be critical to the expansion of the suite of refactorings.

Eclipse 3.0 introduced refactoring over multiple artifact types, which in part dealt with a problem faced by Eclipse plug-in developers: the Rename refactoring had been previously oblivious to references located in plug-in metadata, so that renaming an extension implementation class would break the reference, leaving the extension class unreachable by the extension point framework. Since extension points are the sole mechanism for providing functionality in Eclipse, this was a serious problem. As a solution, the Eclipse Language ToolKit (LTK) provided a "participant" mechanism, allowing additional entities to register interest in a given type of refactoring, and participate in both checking pre-conditions and contributing to the set of changes required to effect the refactoring. Using this mechanism, breakpoints, launch configurations, and other artifacts outside the source itself can be kept in sync with source changes. As applications are increasingly built using multiple languages, this ability becomes critical to the applicability of automated refactoring to the mainstream developer.

Eclipse 3.1 included Infer Type Arguments [3], a migration refactoring that helps Java 5 developers migrate client code of libraries to which type parameters have been added. The migration is important because it increases static type safety. The necessary analysis, however, is subtle and pervasive enough that many developers might hesitate to perform the migration manually. Of particular interest was the Java 5 Collections library, which had been retrofitted with type parameters. In particular, the Infer Type Arguments refactoring infers the types of objects that actually flow into and out of the instances of these parametric types, and inserts the appropriate type arguments into the corresponding variable declarations as needed. In some sense, the underlying analysis reconstructs an enhanced model of the original application, recovering lost or implicit information that may be critical to maintenance or further development. As such, this kind of refactoring offers benefits in maintaining or even "revitalizing" legacy code. Before Infer Type Arguments can

be applied, however, the library itself must be parameterized. Java 5 Collections were parameterized manually, but many other existing libraries would benefit from added type-safety and expressiveness, if they were converted to use generics. A recently described refactoring, Introduce Type Parameter [5], addresses this complex issue. With the addition of such a refactoring, the Eclipse JDT would support developers in a wide spectrum of generics-related maintenance tasks.

Eclipse 3.2 introduced a team-oriented innovation: storing API refactorings with the library itself, along with a "playback" mechanism to automatically perform the necessary transformations on API client code when the new library is imported [4, 1]. Such tools help smooth the interactions amongst team members and between teams, by automatically propagating changes from one component to another, or by automatically making the necessary changes implied by another. As software development becomes increasingly distributed, we believe this sort of tooling will become vital.

Eclipse 3.3 offers an Introduce Parameter Object refactoring. Additionally, a great number of CleanUps that can also be applied to source files on save, for example Organize Imports, Format, or adding missing J2SE-5-style annotations.

Future

With all of the functionality now in Eclipse, we are still a long way from achieving the benchmark of complete coverage of Martin Fowler's refactoring catalog [2]. Moreover, this is only a start; many more transformations are possible. The future promises more emphasis on parallelism in the form of multi-core platforms, clusters and massive machines consisting of thousands or even millions of processors. Concurrency-aware and concurrency-targeted refactorings will be important tools to speed the development of such software. Additionally, most current refactorings effect changes on relatively fine-grained structures such as methods, fields, expressions, statements, and individual types; refactorings that manipulate coarser-grained structures (e.g., packages, entire type hierarchies, components etc.) could enable refining software at nearly the architectural level.

What do we need to deliver on the promise of such a rich suite of transformations? In Eclipse, a refactoring consists of several phases: precondition checking, detailed analysis, and source rewriting. We make three recommendations that, in our opinion, would ease the development of refactorings. First, we need a simpler source rewriting mechanism to avoid writing painful imperative code that creates AST nodes one-by-one. Such a mechanism would be especially helpful if it provided assurances of correctness of the generated constructs, or at least performed run-time checks to help check correctness. The AST and import rewriters already shield refactoring implementors from low-level formatting issues, but higher level APIs would foster more reuse of "refactoring components". Second, we need a better means of specifying the underlying analyses, which maps onto efficient and scalable implementations that permit the application of refactorings to large code bases. Third, much research is needed in understanding the semantics of and manipulating the increasingly prevalent mixtures of languages.

Additional avenues to pursue that would greatly expand the reach of our tooling include that of the Holy Grail of developer-specified refactorings, and that of more general (non-behavior-preserving) developer-specified transformations. The latter tooling could replace the dangerous but prevalent language-oblivious macro processors, giving developers static safety and the power to greatly reduce the difficulty of creating regular code structures. With the combination of these meta-tools at our disposal, both Fowler's catalog and an even richer space of transformations could be within reach; without them, they are likely to take years to attain.

References

[1] D. Dig. Using refactorings to automatically update component-based applications. In *OOPSLA Companion*, pages 228–230, 2005.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[3] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP*, pages 71–96, July 2005.

[4] J. Henkel and A. Diwan. Catchup! capturing and replaying refactorings to support API evolution. In *ICSE*, pages 274–283, May 2005.

[5] A. Kieżun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE*, May 2007.

[6] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley and Sons, 1994.

[7] F. Steimann, P. Mayer, and A. Meißner. Decoupling classes with inferred interfaces. In *SAC*, pages 1404–1408, 2006.

[8] F. Tip, A. Kieżun, and D. Bäumer. Refactoring for generalization using type constraints. In *OOPSLA*, pages 13–26, Nov. 2003.