Integrating Refactoring Support into a Java Development Tool

Dirk Bäumer, Erich Gamma, and Adam Kiezun
Object Technology International
Oberdorfstrasse 8
CH-8001 Zurich
{dirk_baeumer, erich_gamma, adam_kiezun}@oti.com

Manual program refactoring is tedious and error prone. Several refactoring tools have recently emerged for Java. Most of them are add-ons to existing Java development environments [6; 4]. The Eclipse Java tooling [1] is a new open source development environment that comes with integrated refactoring support for Java. Its distinctive characteristics are:

- refactoring support is built on top of an infrastructure that was designed with a focus on refactoring.
- refactoring is seamlessly integrated into the user interface and always available at the developers' fingertips.
- the set of available refactorings can be extended by other contributors.
- undo support to enable exploratory refactoring.

Implementing a refactoring typically involves the following three steps (see Figure 1 and [2]):

1. Determining the set of compilation units affected by a refactoring. Consider the refactoring *Self Encapsulate Field*. Fowler [2] defines it as follows: "Create getting and setting methods for the field and use only those to access the field." In this example all compilation units referencing the field and the one declaring the field belong to the set of affected compilation units.
2. Analyzing the program structure itself (e.g. the methods of a type, the control flow inside a method, etc.) is necessary to implement a refactoring. In the *Self Encapsulate Field* refactoring information about the value assigned to the field is needed. In addition, the transformation of a read-write access depends on the kind of the read-write. Code like `int i= field++;` must be transformed into `int i= getField(); setField(i + 1);`.
3. Transforming the program.

Detecting Affected Code: a program database is used to determine the set of affected compilation units. It stores information about declarations of and references to program elements (e.g. types, methods, fields, etc.). The database is updated in a separate thread whenever there are changes (new, remove, save) to Java compilation units. The database is always up to date independent of whether a program is compiled or not. The stored information can be inaccurate since a compilation unit is not always syntactically correct. A search engine implemented on top of the program database provides search queries like: which program element references the method `foo()` or, where is type `A` declared. The search engine annotates each search result with accuracy information (potential match, precise match). The program database stores reference information at the granularity of compilation units. Therefore an additional step is required to narrow the reference information to a precise location inside the compilation unit. To do so the search engine parses the Java compilation units in question. The parsing step also adds additional information like the position of the reference or the access kind (read, write, etc.) of a field reference. This two step approach (coarse grained program data base combined with on the fly parsing) enables precise searching with a minimal footprint and fast incremental updating of the database. The search engine together with its underlying program database is also used by the IDE's general search infrastructure.

Structural Analysis: in addition to information about declarations and references a more detailed analysis of the program structure itself is required to verify the refactoring's preconditions and to collect additional information for the code transformation. Refactorings use two different components to perform structural analysis: the Java Model and abstract syntax trees. The Java Model provides API for navigating the Java element tree. The Java element tree represents projects, packages, compilation units and binary classes. For compilation units and binary classes the Java Model provides access to the different declarations like imported declarations, declared types, declared methods and fields of a type, etc. The Java Model is typically used to implement preconditions like "does type exist," or "does method exist in class." Refactorings below the method level (e.g. *Extract Method*) require an abstract syntax tree analysis. A parsing framework, shared with the IDE's Java compiler, is used to provide access to abstract syntax trees. This allows sharing information required by the incremental compiler during the abstract syntax tree

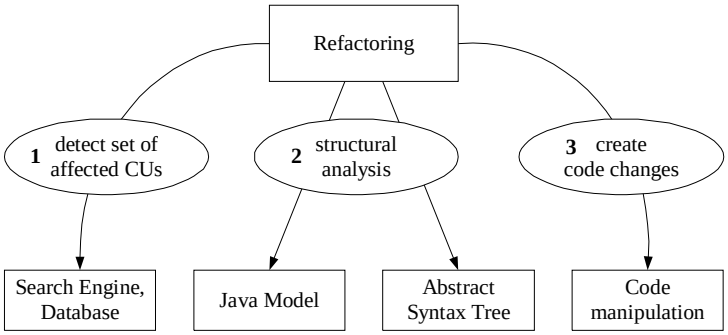

Figure 1: *The core architecture.*

construction. This tight architectural integration avoids the duplication of structural information between the base IDE and the refactoring support. This ensures both performance and scalability.

Create Code Changes with Support for Undo: once the program structure is analyzed the code needs to be transformed. To do so, a refactoring creates change objects to manipulate the code. Change objects provide full undo/redo support (implemented using the command pattern [3]). Two change kinds are distinguished: textual changes and non-textual changes. Non-textual changes manipulate the file system (e.g. rename or delete a file). Textual changes transform compilation units. Examples for textual changes are: add a method, or update a type reference. In contrast to other refactoring tools [5; 6] we don't use abstract syntax tree rewriting to change a compilation unit's content. The reasons are: direct text manipulation can also be used for files other than Java compilation units (for example Java Server Pages, which can have embedded Java code) and it is easier to keep existing code formatting since only parts of a file are changed. For example, keeping gratuitous parenthesis is difficult, since they are normally not present in an abstract syntax tree. In addition, database search results contain precise positions making the generation of corresponding text changes straightforward.

UI: the user interface for performing a refactoring uses a multi-page wizard. All refactoring wizards have the same structure: the first page gathers the arguments to perform a refactoring. For example the new name of a class that is to be renamed. Once this information is available, the user presses 'Finish' to perform the refactoring or 'Next' to see a preview of the changes to be carried out by the refactoring. In both cases precondition checking is activated. If it reports any problems they are presented to the user. The refactoring wizard as well as the pages for error reporting and previewing the changes are available as reusable building blocks.

Experience: sharing the abstract syntax tree with the Java compiler has the mentioned benefits but there are also some challenges. A compiler's abstract syntax tree is tuned for code generation. Therefore information that is not required for code generation is not present. For example, no information is maintained about the position of semicolons or parenthesis for a method call. But those positions are needed for refactorings like *Extract Method* to determine if the selection covers a valid set of statements.

Using the infrastructure provided by the Eclipse Java Tooling enables the developer to focus on the key issues when implementing a refactoring: precondition checking and code manipulation. While transforming the code is straightforward with the provided framework, implementing precondition checking is a significant effort (for example, 80% of the rename type refactoring code is for precondition checking). To address this problem work on reusable precondition checking has started. For example, "does a type conflict (hide or obscure) with some other Java element," or "does this method override or overload an existing method."

The Eclipse platform and its Java Tooling are available as open source and we hope that the infrastructure provided will encourage other developers to contribute to the set of supported refactorings.

References

[1] The Eclipse Project. On-line at http://www.eclipse.org/.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison Wesley, 1999.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[4] JRefactory. On-line at http://jrefactory.sourceforge.net/.

[5] The Smalltalk Refactoring Browser. On-line at http://st-www.cs.uiuc.edu/users/brant/Refactory/.

[6] Transmogrify.On-line at http://transmogrify.sourceforge.net/.