Immutable Source-Mapped Abstract Syntax Tree: A Design Pattern for Refactoring Engine APIs

JEFFREY L. OVERBEY, Auburn University

Many automated refactoring tools provide extensibility mechanisms that allow third parties to contribute new refactorings. However, implementing nontrivial refactorings requires deep syntactic knowledge about the program being refactored. Refactoring tools almost always use abstract syntax trees (ASTs) to describe the structure of source code. Therefore, it may be desirable to expose these trees via an API. This pattern describes the common attributes that ASTs typically exhibit in order to be useful for implementing nontrivial refactorings while, at the same time, maintaining the characteristics of good API design. In particular, these trees tend to have an immutable structure, and they maintain very detailed information about the source code, including exact character positions. This pattern is used in the Eclipse Java Development Tools' refactoring API as well as the Microsoft "Roslyn" CTP, among other uses.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**] Coding Tools and Techniques

General Terms: Languages

Additional Key Words and Phrases: abstract syntax trees, APIs, ASTs, immutability, refactoring

ACM Reference Format:

Overbey, J. 2013. Immutable Source-Mapped Abstract Syntax Tree. jn 0, 0, Article 0 (May 2013), 8 pages.

Many integrated development environments (IDEs) provide extensibility mechanisms, which allow third-party developers to add new functionality to the IDE. For IDEs that provide automated refactoring support, it is common to provide the ability to contribute new automated refactorings.

However, automated refactorings manipulate source code, so implementing an automated refactoring requires a data structure that describes the structure of source code. Refactoring tools almost always use a data structure called an *abstract syntax tree* (AST). ASTs describe source code hierarchically: e.g., classes contain methods, methods contain statements, statements contain expressions, etc. ASTs are used in compilers and static analysis tools, but refactoring tools impose more stringent requirements—specifically, the ASTs must be suitable for manipulating the user's source code.

If an IDE provides an application programming interface (API) that allows third parties to contribute new refactorings, then it is often helpful for that API to provide access to ASTs as well. This pattern considers ASTs exposed by such APIs, identifying the common elements necessary to provide both a clean API and a data structure suitable for implementing nontrivial refactorings.

1. CONTEXT

You are implementing a refactoring engine or a similar tool that makes automated edits to source code. You have provided a mechanism that allows third parties to contribute new refactorings/transformations. You have an API for adding, deleting, and changing source code files (so the mechanism to, say, "delete the first 27 characters of *main.c*" is well-defined).

2. PROBLEM

Although you have an API for contributing refactorings/transformations, and you have provided the ability to make string-like edits to source code, this is a very primitive API that eschews any real knowledge about the syntactic structure of the language being refactored.

Consider a simple Java refactoring that adds braces to the bodies of *if* statements, if they do not already exist. For example, it would make the following change.

```
if (a < b)                              if (a < b) {
    if (c < d)                              if (c < d) {
       System.out.println("a");                 System.out.println("a");
    else                            ⟹      } else {
       System.out.println("b");                 System.out.println("b");
                                            }
                                        }
```

Even this simple refactoring requires nontrivial knowledge of Java's syntax, including how the dangling *else* is matched with a corresponding *if*.

How can the refactoring engine provide an API that exposes the syntactic structure of programs in a way that is useful for implementing refactorings?

3. FORCES

—The API must expose fine-grained details about the syntax of a program. For example, the preceding example requires being able to determine the presence or absence of braces `{ }`, even though they are semantically irrelevant.

—Many refactorings are initiated only after the user selects a range of source code in a text editor. Therefore, the API must allow clients to determine what syntactic construct(s) are selected.

—Clients must be able to use the API to manipulate source code in a way that is minimally invasive: refactorings that only affect a small, isolated region of the source code should not change any source code outside that region. This includes preserving the program's comments and formatting [Sommerlad et al. 2008].

—The API must follow general principles of good API design, including being easy to learn and difficult to misuse.

—The API should be usable when building refactorings for large code bases. Automated refactorings are most useful for maintaining such codes.

4. SOLUTION

Use a parser for the language being refactored to construct an abstract syntax tree*, or* AST*. Make the AST* immutable*—i.e., its structure cannot be changed—but provide* source mapping *information so that every node in the AST can be mapped to a particular substring of the original source code.*

4.1 Abstract Syntax Trees

(Readers familiar with ASTs may want to skip to §4.2.)

Programs have a natural hierarchical structure. For example, a Java source file contains a package declaration, import statements, and one or more type declarations (i.e., class/interface declarations). A type declaration contains field and method declarations. A method's body consists of a sequence of statements. And so forth.

An *abstract syntax tree* (AST) is a data structure that describes a particular program in terms of this hierarchy. Continuing with the Java example, an AST typically describes the structure of a single Java source file. The root node has three child nodes, corresponding to the three parts of a Java source file: one child node describes the

`package` statement, another contains a list of `import` statements, and a third contains a list of type declarations. A type declaration has several child nodes, which describe (among other things) its visibility, the name of the class or interface, and the field/method declarations in its body. A field declaration has child nodes describing its visibility, type, and so forth.

Figure 1 shows a visualization of an AST for the following program:

```
public class Example {
  public int FIVE = 5;
}
```

Notice how deeper levels of the tree describe progressively more fine-grained syntactic structures. The root node represents the entire Java source file. The highlighted node—a *VariableDeclarationFragment*—represents `FIVE = 5`, while its child nodes represent its constituent parts: the identifier `FIVE` and the numeric literal `5`. These are leaf nodes, since these tokens (like keywords, identifiers, and numeric literals) represent the smallest syntactic units in the language.

Developers familiar with the HTML or XML DOM (Document Object Model) will find this structure familiar. The DOM is effectively an abstract syntax tree for HTML/XML documents.

Deeper discussions on ASTs can be found in many introductory textbooks on compiler construction, such as [Aho et al. 2006] and [Torczon and Cooper 2011]. Such books also discuss how parsers are constructed, including how they can be modified to construct ASTs.

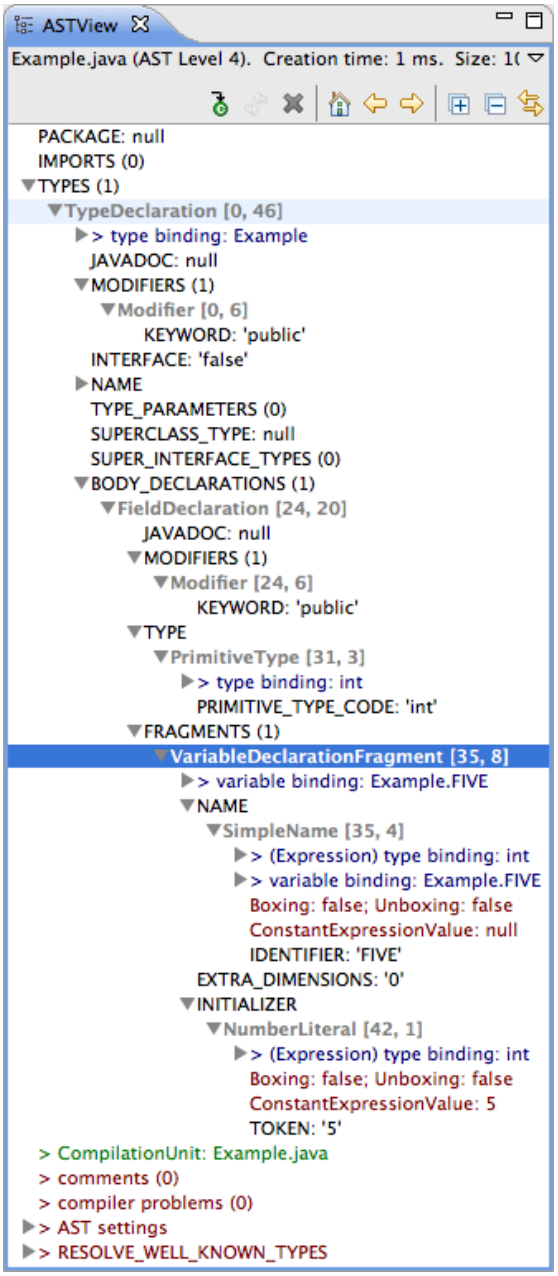


Fig. 1. Visualization of an AST produced by the Eclipse Java Development Tools (JDT).

4.2 Fulfilling the Requirements of Refactoring

ASTs are not unique to refactoring tools. They were used in compilers long before refactoring tools existed. They are also used in static analysis tools, prettyprinters, and other tools that analyze source code. This makes reuse very tempting: If there is already a compiler for a language, and it contains an AST, why not reuse it in a refactoring tool?

Unfortunately, an AST designed for use in another context (e.g., a compiler) is not necessarily suitable for refactoring. This is because refactoring tools impose two very significant—and unusual—requirements on their ASTs.

(1) To be useful for refactoring, an AST must accurately and precisely model the user's source code at a very fine level of detail. It may even be necessary to include information about punctuation and comments in order to refactor code appropriately [Sommerlad et al. 2008].

(2) It must be possible to map nodes in the AST to locations in the source code, and vice versa.

To illustrate the first requirement, consider an example from Fortran. The Fortran language contains a number of input/output statements. Two of these are PRINT and WRITE: the statements `print *, "X"` and `write (*,*) "X"` can be used interchangeably to print the string `X` to standard output. In general, the statement `print` *fmt*`,` *expr* is equivalent to the statement `write (*,` *fmt*`)` *expr*. Most compilers type check and translate them identically. So, a compiler may choose to represent PRINT statements as WRITE statements in its AST. The GNU Fortran compiler does this, for example. However, the ability to convert PRINT statements to WRITE statements actually makes for a useful refactoring. (Consider a program that uses PRINT statements to write to standard output, but the programmer later needs to use WRITE statements to write to a file instead.) To implement this refactoring, a refactoring tool needs to be able to distinguish PRINT statements from WRITE statements—a difficult task if they are represented identically in the AST.

To explain the second requirement—that it must be possible to map nodes in the AST to locations in the source code, and vice versa—recall that many refactorings are initiated only after the user selects a range of source code in a text editor. Usually, this selection is represented by two numbers: the *offset* of the starting character and the *length* of the selection. In the simple Java program whose AST is shown in Figure 1, the substring `FIVE = 5` is eight characters long and begins with the 36th character in the source code (offset 35, assuming the first character is considered to be at offset 0). If the user selects the text `FIVE = 5` in the text editor, it must be easy to determine that this corresponds to the *VariableDeclarationFragment* node in the AST.

The easiest way to map AST nodes to source locations is to store source location information in the AST nodes themselves. Again, consider the JDT AST shown in Figure 1, and note the numbers in square brackets. The highlighted node is displayed as "VariableDeclarationFragment [35, 8]": this AST node corresponds to the eight-character string beginning at offset 35 (`FIVE = 5`). Its child node corresponding to the identifier `FIVE` is attributed with the offset-length pair [35, 4], while the numeric literal `5`'s source location is [42, 1].

Now, consider what happens when a user selects text in the text editor. This can be represented as an offset-length pair. When source location information is available for every AST node, it becomes fairly easy to determine what syntactic construct corresponds to the text selection corresponds to: simply walk the tree, searching for a node whose source location aligns with the user's selection.

Since AST nodes at deeper levels of the tree represent finer-grained syntactic constructs than their ancestor nodes, ASTs with source location information typically exhibit a nice property: the source location of a child node will always be contained within the source location of its parent. This property can be used to expedite searches based on source location information. For example, if a particular node corresponds to the source location [35, 8], and you are searching offset 63, then the desired node cannot possibly be a descendent of that node.

4.3 Mutable vs. Immutable ASTs

When ASTs are used in compilers, it is common to make them *mutable*, meaning that nodes can easily be added, changed, or deleted from the tree. For example, a compiler might determine that `a = b*1` can be simplified to `a = b`, and then change the AST accordingly.

At first, this seems like a reasonable mechanism for refactoring tools to use as well. To remove a field declaration from a program, simply delete its *FieldDeclaration* node from the AST (along with all its children). To change the name of `FIVE`, change the contents of its *SimpleNameNode* in the AST. Of course, some mechanism is needed to prettyprint the modified AST, so these changes can be translated back into source code, but that is relatively straightforward to implement.

Unfortunately, while a mutable AST can provide an elegant means of manipulating source code, that are other issues that can make mutable ASTs problematic from an API design perspective. Some notable issues include the following.

—*AST nodes contain source location information.* After the structure of the AST is changed, what does the source location information in each node represent: the "old" source location (in the original code) or the "new"

location (after it has been modified)? If it represents the old location, what should the source location be when new nodes are inserted into the tree? If it represents the new location, then every time a change is made to the tree, source location information must be updated for every node in the tree, which may be expensive . . . or else a clever implementation (e.g., computation from relative offsets) must be used instead.

—*AST nodes may be removed from the tree, leaving them orphaned.* Suppose a client traverses the AST, storing pointers to some "interesting" nodes in a collection. Then, an ancestor of one of those nodes is removed from the AST. This will leave the client in a difficult-to-debug situation where his collection contains pointers to nodes that no longer exist in the AST.

—*Semantic information may be invalidated.* Refactorings often need to determine, for a particular use of a variable, what declaration that use corresponds to. For example, in Java, if a field and a local variable were both named `x`, then in the expression `this.x = x`, the first `x` would refer to the field, while the second would refer to the local variable. However, if the local variable declaration were removed, the second `x` would refer to the field instead. This can be confusing to clients, since information collected about the program at one point in time may no longer be true after the tree's structure changes.

—*It may be possible to mutate the AST so that it does not represent a legal program.* It might even be possible to mutate the AST so that it is no longer a tree at all. In Java, classes can contain other classes. What happens if a client tries to make a class an inner class of itself? (This could turn the tree into a cyclic graph.) The API for modifying the AST's structure could try to prevent such things, but attempting to detect and prevent every possible mistake would be difficult and potentially expensive.

—*Mutability can be problematic in multithreaded code.*

Mutable ASTs were used in the original Smalltalk Refactoring Browser [Roberts et al. 1997], as well as in CRefactory [Garrido 2005] and Photran [Overbey and Johnson 2009]. Many compilers use mutable ASTs internally. However, these tools have one thing in common: the mutable AST is not accessible via any public API. It is intended for internal use only, by developers who should be better equipped to handle all of the caveats.

Guidelines for API design are discussed in detail in Jaroslav Tulach's *Practical API Design* [Tulach 2008]. A presentation by Joshua Bloch [Bloch 2006] offers excellent advice as well. Both authors suggest minimizing the amount of mutability in an API. Immutable objects tend to have simpler interfaces, and they are thread-safe. Both authors also emphasize that APIs should be easy to learn (even for programmers who do not read all of the API documentation) and difficult for clients to misuse. Given these guidelines, and the list of liabilities for mutable ASTs given above, it seems preferable from the perspective of someone attempting to publish an API to avoid mutable ASTs. But this raises a different question: If ASTs are immutable, what API should be provided for clients to manipulate source code?

4.4 Using Immutable ASTs for Source Code Manipulation

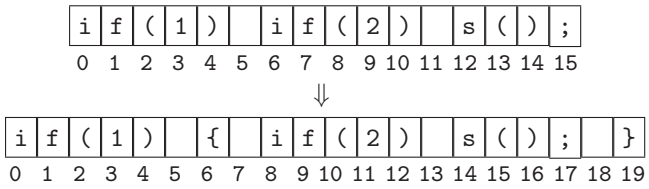
The ultimate goal of a refactoring tool is to modify the user's source code. When ASTs are immutable, this cannot be done by modifying the structure of the AST. Instead, it is done differently: *Treat the source code as a string, and use the source location information in the AST to add, replace, and delete substrings.*

As an example, consider deleting a field declaration. In a mutable AST, you would simply remove the *FieldDeclaration* node from the AST. With an immutable AST, you would retrieve the source location mapping from the *FieldDeclaration* node, and then remove that substring from the source file.

In implementation, the code for removing a field declaration using a mutable AST would probably be something like *field.removeFromTree()*. The code for doing the same with an immutable AST would be something like *sourceCode.deleteSubstring(field.getOffset(), field.getLength())*.

The situation gets a bit worse when changes are made to nested constructs. Recall the example from Section 2, which added braces to two *if* statements, one nested inside the other. Suppose the outer *if* statement were

modified first: Adding braces to it would change the position at which the closing brace for the inner *if* statement would need to be added. Consider a somewhat simplified example. Pictorially:



The body of the inner *if* statement begins at offset 12 in the original code. But after adding braces to the outer *if* statement, the body of the inner *if* statement is shifted to offset 14.

In other words, when refactorings affect nested language constructs, offset/length computations can become complex, since the edits for one construct may interact with the edits for another construct.

So, manipulating source code using a mutable AST is simpler, and it more directly expresses the programmer's intention. Manipulating source code using offset/length information leads to ugly client code. The client code can easily become littered with variables holding character offsets and methods that perform intricate string manipulations.

In their discussions of API design, Bloch submits that it should be "easy to read and maintain code that uses [the API]" [Bloch 2006], and Tulach notes the advantages of APIs where the clients "don't describe step-by-step what they want your API to do. Rather, they 'declare' what they want to have happen and then rely on your API to do it. . . " [Tulach 2008, p. 225].

So, is it possible to provide a cleaner source manipulation API, while still using an immutable AST? It is, but some additional API is needed.

One option is to provide an API that allows clients to specify what changes they want to make in terms of AST nodes. This API can then translate these into string edits. For example, an API might offer a *Rewriter* class. The client would inform the *Rewriter* that it wants to effectively delete a particular *FieldDeclaration* node from the AST: *rewriter.delete(field)*. When it is ready to manipulate the source code, it delegates this task to the rewriter: *rewriter.rewrite(sourceCode)*. This is illustrated in Figure 2. The Eclipse Java and C/C++ Development Tools [Rüegg 2012] take this approach (with slightly different class and method names).

A second option is to work with ASTs in a functional style, where every edit returns a new AST. So, a call to *field.removeFromTree()* would not change the existing AST, but rather it would return a new AST that has the given field omitted. (Of course, this requires a very careful implementation. Memory consumption and copying are a concern, particularly when ASTs are large. Also, a mechanism must be provided so that clients can easily map nodes in the original tree to the "same" node in the modified tree, and vice versa.)

5. KNOWN USES

Immutable, source-mapped ASTs are used by the Eclipse Java Development Tools (JDT) and C/C++ Development Tools (CDT), as well as the Scala IDE for Eclipse, the Microsoft "Roslyn" Community Technology Preview (CTP), Clang, and the original refactoring engine in Apple Xcode. However, while all use immutable ASTs with source mapping information, the actual rewriting APIs vary substantially.

Both Clang and the original refactoring engine in Xcode 3.0 modify source code using offset-length information directly. Notably, Clang offers a *RewriteRope* class—essentially, a custom string class that allows for efficient mid-string insertions and deletions[1]. (It uses a B-tree structure inspired by, but not identical to, that described in the original paper on Ropes [Boehm et al. 1995].)

[1] `http://clang.llvm.org/doxygen/classclang_1_1RewriteRope.html`

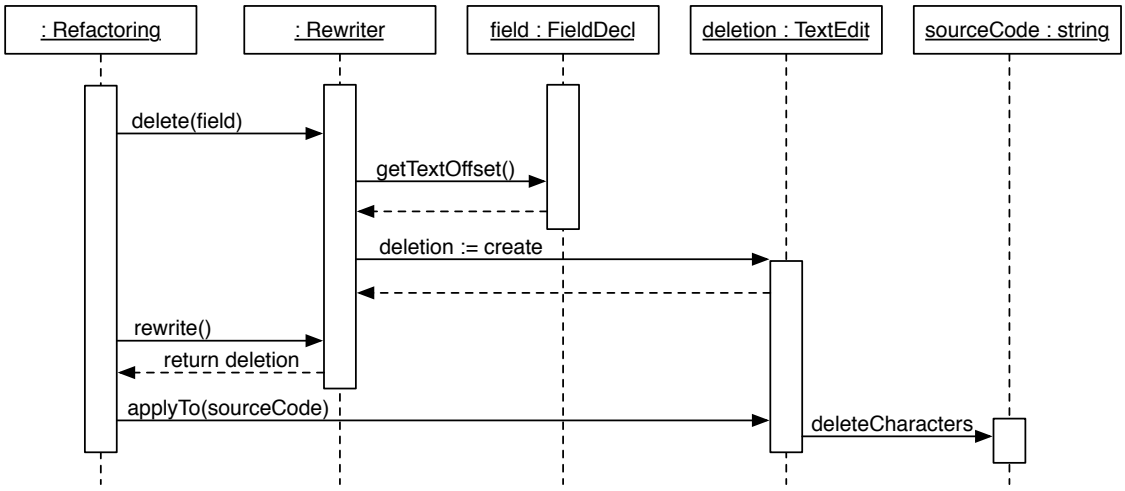


Fig. 2. Sequence diagram illustrating use of a Rewriter to modify source code.

The Eclipse Platform is designed to be language-independent; individual plug-ins like JDT and CDT determine what languages it supports. The Eclipse Platform includes a component called the Eclipse Language Toolkit (LTK) [LTK 2006]. which provides (among other things) a language-independent API for manipulating text files using offset/length information. Specifically, *TextEdit* objects describe individual changes (like "delete three characters starting at offset 462"), and many such changes are aggregated into a single *Change* object that represents the net effect of a refactoring. Of course, the LTK provides the infrastructure necessary to apply *Change* objects, writing the modifications to disk, as well as to undo those changes at a later point in time.

Eclipse JDT provides a class (*ASTRewrite*[2]) that allows the programmer to specify modifications in terms of Java AST nodes, and then translates those changes into an LTK *TextEdit* object. Eclipse CDT's refactoring engine is based on JDT's and follows a similar design; it offers its own version of *ASTRewrite* [Rüegg 2012].

Refactoring support in the Scala IDE for Eclipse[3] takes a functional approach, implementing refactorings as tree transformations. Source generation is handled separately: one printer uses position information to retain the formatting of the original code, while another pretty-prints new nodes that were added to the AST [Stocker 2010].

Microsoft's "Roslyn" CTP takes a functional approach, where imperative methods on AST nodes do not modify the AST but rather return a new AST [Vogel 2012]. Since a naïve implementation could make this prohibitively expensive, the Roslyn team devised an implementation that they call *red-green trees* (named after the colors of the whiteboard markers that were used during the design meeting) [Lipper 2012]. Essentially, the AST exposed via the API (the "red tree") is a façade for a different, internal tree (the "green tree"); the implementation attempts to reuse existing AST nodes whenever possible (which is possible since all nodes are immutable).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1217271. The author would like to thank Peter Sommerlad, who shepherded this pattern, as well as the members of Workshop 000001 at PLoP 2013: Filipe Correia, Dibyendu Goswami, Joshua Kerievsky, Juan Reza, and Russ Rubis. Their feedback was immensely helpful.

[2] `http://help.eclipse.org/kepler/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/rewrite/ASTRewrite.html`

[3] `http://scala-refactoring.org`, `http://scala-ide.org/`

REFERENCES

AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BLOCH, J. 2006. How to design a good API and why it matters. `http://www.infoq.com/presentations/effective-api-design`.

BOEHM, H.-J., ATKINSON, R. R., AND PLASS, M. F. 1995. Ropes: An alternative to strings. *Software: Practice and Experience 25*, 1315–1330.

GARRIDO, A. 2005. Program refactoring in the presence of preprocessor directives. Ph.D. thesis, Champaign, IL, USA.

LIPPER, E. 2012. Persistence, facades and Roslyn's red-green trees. `http://blogs.msdn.com/b/ericlippert/archive/2012/06/08/persistence-facades-and-roslyn-s-red-green-trees.aspx`.

LTK 2006. The Language Toolkit: An API for automated refactorings in Eclipse-based IDEs. `http://www.eclipse.org/articles/Article-LTK/ltk.html`.

OVERBEY, J. L. AND JOHNSON, R. E. 2009. Generating rewritable abstract syntax trees. In *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*. Lecture Notes in Computer Science Series, vol. 5452. Springer-Verlag, Berlin, Heidelberg, 114–133.

ROBERTS, D., BRANT, J., AND JOHNSON, R. 1997. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems 3,* 4, 253–263.

RÜEGG, M. 2012. Eclipse CDT refactoring overview and internals. `http://wiki.eclipse.org/images/b/be/PTPUserDev2012_Ruegg_Refactoring.pdf`.

SOMMERLAD, P., ZGRAGGEN, G., CORBAT, T., AND FELBER, L. 2008. Retaining comments when refactoring code. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA Companion '08. ACM, New York, NY, USA, 653–662.

STOCKER, M. 2010. Scala refactoring. M.S. thesis, HSR Hochschule für Technik Rapperswil.

TORCZON, L. AND COOPER, K. 2011. *Engineering a Compiler* 2nd Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

TULACH, J. 2008. *Practical API Design: Confessions of a Java Framework Architect*. Apress/Springer-Verlag, New York, NY, USA.

VOGEL, E. 2012. Roslyn CTP custom refactoring. `http://visualstudiomagazine.com/Articles/2012/03/15/Roslyn-CTP-Custom-Refactoring_1.aspx`.