iXj: Interactive Source-to-Source Transformations for Java

Marat Boshernitsan, Susan L. Graham
University of California at Berkeley
Computer Science Division, EECS
Berkeley, CA 94720-1776
+1 510 642 4611
{maratb,graham}@cs.berkeley.edu

ABSTRACT

Manual large-scale modification or generation of source code can be tedious and error-prone. Integrating scriptable source-to-source program transformations into development environments will assist developers with this overwhelming task. We discuss various usability issues of bringing such *ad-hoc* transformations to end-users and describe a developer-oriented interactive source code transformation tool for Java that we are building.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments, interactive environments.*

General Terms

Design, Human Factors, Languages.

Keywords

Interactive Environments, Program Transformation Languages.

1. INTRODUCTION

Changing software source code can be tedious and error-prone. The process is complicated because a conceptually simple change may entail pervasive large-scale modifications to a large portion of source code. Examples of such changes abound in the many maintenance tasks faced by developers. For instance, consider a simple task of inserting the name of the enclosing function into the code that prints debugging messages to the console. Unless the language provides programmatic access to "current function name," the implementation of this trivial change might take hours of the developer's time.

Various proposals have been made for automating systematic modification to source code. However, few tools have found their way to the "programming trenches." If a modification is a simple behavior-preserving refactoring and the development environment is sufficiently advanced to include automated support for such transformations, the change process is quick and convenient. However, many modifications simply cannot be broken down into a sequence of well-behaved refactorings. (Several compelling examples may be found in Roberts and Brant [9].) Another option is to employ a general source-to-source transformation engine such as REFINE [3] or TXL [4]. Unfortunately, specifying transformations with these tools requires familiarity with a fairly complex transformation language; so, using such a system for simple changes is overkill. If the modification is not complicated, developers may choose to utilize regular expression-based pattern matching facilities of Perl, SED, or other text-oriented tools. Needless to say, using regular expressions for anything but the most trivial of transformations is usually an exercise in frustration.

To solve these problems, we propose to use *ad-hoc* source-to-source transformations in an interactive mode during authoring and editing phases of software development. Transformations can be construed broadly. In addition to replacing existing code, transformations can also generate new code fragments based on linguistic structure or on meta-information embedded in program source code. In all cases, such tools must meet unique challenges posed by their interactive mode of use. Not only must interactive transformations tools be sufficiently powerful to deal with a broad range of code changing tasks, but also they must address usability issues that arise when attempting to manipulate a non-textual linguistic representation of program source code.

To deal with these challenges, we developed iXj[1] – a language for specifying source-to-source transformations on Java source code. iXj transformations are constructed in an interactive environment that assists the developer with visualizing and directing the transformation process. iXj enables the programmers to utilize high-level linguistic structure and programming language semantics similar to those used when thinking about and discussing software changes. This allows the programmers to express operations on program source code at a level above text-oriented editing, which we believe will improve their efficiency and introduce fewer errors during the modification process.

2. THE HUMAN FACTOR

Many existing tools support specification and execution of transformations on program source code. In addition to aforementioned REFINE and TXL, notable examples include TAWK [7], Inject/J [6], the IP environment [10], and the Refactoring Browser Rewrite Engine [9]. However, these tools are intended for expert use on large and complex tasks. By contrast, our system is oriented toward *end-programmers* – the end-users of traditional development environments. We draw this distinction to differentiate end-programmers from *language tool experts*. Language tool experts are those who understand the structure of program source code from the perspective of compiler-like tools, and may be comfortable thinking about source code in terms of linguistic data structures. We do not expect end-programmers to possess this skill.

OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

[1] iXj – Interactive Xformations for Java

Nevertheless, end-programmers' understanding of program source is based on its structure. This is supported both by our empirical observations of developer expression and by the experimental results in psychology of programming [5]. When describing source code to one another, programmers say things like:

> "Put *p:= link(p)* into the loop of *show_token_list*, so that it doesn't loop forever." [8]
>
> "Change *BI_** macros to *BYTE_** for increased clarity." [11]

Programmers evoke notions such as variables, expressions, statements, loops, and assignments. They directly refer to names found in source code. They use patterns to describe large classes of similar changes. Inspired by these kinds of examples, we can design a formal language for source code transformations.

3. INTERACTIVE TRANSFORMATIONS

Guided by the above principles, we are building an end-programmer-oriented interactive tool for source code transformation. In order to enable developers to describe transformations using familiar concepts, we targeted our notation toward the Java programming language. While iXj is a language tightly coupled with Java, we expect that our design methodology will be applicable for other programming languages.

Prior to designing iXj, we conducted an informal user experiment to understand what programming paradigm is most "natural" for expressing transformations. In this experiment the participants were shown "before" and "after" snapshots of a piece of source code and were asked to write down the transformation that was used to perform the change. In particular, we were interested how developers reference code fragments to be transformed, how they describe the output, and what programming style they use. We learned that to describe a location in the source code developers use language concepts ("in class *Employee*, method *getName*…") interspersed with code fragments in Java ("replace *System.out.println(x)* with…"). We also discovered that imperative programming style ("first do this, then do that") is most natural for describing modifications.

Armed with this knowledge, we based the first version of iXj on the selection/action programming model. A *selection* is a pattern that describes a set of Java source code fragments. One or more *actions* describe a transforming operation for each selection.

In order to provide scaffolding to help developers learn and use an unfamiliar notation, we are also building an integrated transformation environment for creating and executing iXj programs. This environment is being prototyped on the Eclipse platform, augmented with the Harmonia framework [1] to provide advanced program analysis infrastructure.

In addition to offering context-sensitive assistance during creation of iXj programs, the transformation environment enables the programmers to visualize execution of iXj selections and actions as well as to view partial results of the transformation. Additionally, the developer can examine each transformation site, selectively undo or modify individual transformations, etc. The transformation environment can also capture the source code change history in terms of high-level transforming operations. Such a capability helps to document important aspects of program evolution, as well as supports selective rollback of high-level changes days, months, and even years later.

An important advantage of using an integrated environment for transforming source code is the ability to treat the iXj programs as abstractions. Not only does this permit naming transformations and storing them in a library for reuse, but also it allows treating transformations as *update agents*. An update agent is a metaprogram bound to both the source and the target (generated) program elements. An integrated transformation environment can track dependencies between the two sections of source code and act appropriately if the developer makes changes to either.

We believe that iXj will provide the right high-level vernacular for describing code, and we expect professional developers to have no trouble specifying the control structure of pattern matching and transformations in a textual notation. At the same time, the transformation environment augments iXj with direct manipulation. Selection patterns can be created "by-example," whereby the user selects a source fragment that represented a single matching instance and then abstracts the generated pattern to match a larger class of code fragments.

4. CONCLUSION

The presented work draws on our earlier proposal for *ad-hoc* manipulation of source code with interactive transformations [2]. Since then, we have designed the iXj language and partially completed the implementation. This experience has informed our understanding of the issues involved in designing a human-oriented source code transformation tool.

5. REFERENCES

[1] M. Boshernitsan. HARMONIA: A Flexible Framework for Constructing Interactive Language-Based Programming Tools. Technical Report. UC Berkeley. UCB/CSD-01-1149, 2001.

[2] M. Boshernitsan. Program Manipulation via Interactive Transformations. In *Companion of the 18th Conference on Object-oriented programming, systems, languages, and applications.* 2003.

[3] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software reengineering. In *Proceedings of the 14th International Computer Software and Applications Conference*. IEEE Computer Society Press, 1990.

[4] J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceeding of the International Conference of Computer Languages*, pp. 280-285, Miami, FL, Oct. 1988.

[5] F. Detienne. *Software Design – Cognitive Aspects.* Springer-Verlag, New York, NY, 2001.

[6] T. Genssler and V. Kuttruff. Source-to-source transformations in the large. In *Proceedings of Joint Modular Language Conference (JMLC) 2003*.

[7] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the 4th Workshop on Program Comprehension.* 1996.

[8] D. E. Knuth. The errors of TeX. *Software – Practice and Experience*, 19(7):607-685. July 1989.

[9] D. Roberts and J. Brant. Tools for making impossible changes - experiences with a tool for transforming large Smalltalk programs. In *IEE Proceedings Software*, 151(2):49-56. April 2004.

[10] C. Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research (MSR), Sept. 1995

[11] B. Wing. XEmacs ChangeLog entry for 2002-05-05. http://cvs.xemacs.org/viewcvs.cgi/XEmacs/xemacs-20/src/ChangeLog