

Program Manipulation via Interactive Transformations

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

ABSTRACT

Systematic large-scale modification of source code is tedious and error-prone, because developers use authoring and editing tools poorly suited to the program maintenance task. We combine the results from psychology of programming, software visualization, program analysis, and program transformation fields to create a novel environment that lets the programmers express operations on program source code at a level above text-oriented editing.

Categories and Subject Descriptors

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

General Terms

Design, Human Factors, Languages.

Keywords

Interactive Environments, Programming Psychology.

1. INTRODUCTION

Software artifacts are bound to change. Be it due to design changes, bug fixes, or addition of new features, the process of modifying software source code is often tedious and error-prone.

The change process is complicated because many changes entail pervasive large-scale modifications to an existing body of source code. Examples abound in the many maintenance tasks faced by developers during the lifetime of a typical software project. For instance, simply adding an argument to a procedure requires visiting every invocation site and supplying the missing value. Another example, observed by the aspect-oriented programming researchers, involves delocalized design abstractions such as exception handling, logging, synchronization, and others. Capturing these design abstractions at the source code level is difficult due to limited data and procedural abstractions provided by most programming languages. As trivial a modification as changing which exceptions are handled following the call to a library routine requires finding all such calls and modifying the exception handlers. Yet another flavor of high-level operations is the generative operations that produce chunks of boilerplate code. Outputting fields of a data structure is an example of such an operation. If the list of fields is large and may change over the lifetime of the program, maintaining the output routine “manually” is a tedious and boring task.

Copyright is held by the author/owner(s).
OOPSLA '03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

Various proposals have been made for systematic modification to existing source code. However, few tools have found their way to the “programming trenches.” Our research attacks several major issues with prior approaches: generality, acceptance by the user community, improved abstraction management, and proliferation of proprietary programming language extensions (such as those used in aspect-oriented programming).

We believe that the problem stems from the fact that developers use authoring and editing tools poorly suited to the program maintenance task. Most such tools provide text-based interaction with minimal syntax knowledge and very little structural or high-level language awareness. The use of such tools has significant limitations. On the one hand, developers utilize high-level linguistic structure and programming language semantics when thinking about and discussing software artifacts. On the other hand, the developers are forced to interact with computing systems to create and modify software artifacts using low-level text editors and representations designed for compiler input. Our hypothesis is that enabling the programmers to express operations on program source code at a level above text-oriented editing will improve programmer's efficiency and result in fewer errors.

2. INTERACTIVE TRANSFORMATIONS

When describing their changes to one another, programmers evoke notions such as variables, expressions, statements, loops, and assignments. They also directly refer to names found in source code. These concepts represent the common terminology understood by programmers, making the use of those terms natural for describing actions at a high level. Such descriptions can provide a user-friendly notation for expressing source code manipulations using high-level “update scripts”. The scripts might be executed interactively; they might be stored in a library or a catalog, or they might serve as update agents bound to program components. A sample update script might look like:

foreach statement like `f(@args@) replace with f($args, true)`

This script describes an update following the addition of a new argument to function `f`. Another update script of a more generative flavor might be something like:

```
generate method Node.mem_size() as
int mem_size() {
  int result = 0;
  foreach field f of class Node emit result += $f.mem_size();
  return result;
}
```

This script acts as an update agent that describes how to generate a method. The generated method will be updated each time a field is added or removed in the class `Node`.

The examples use a scripting language that embeds both instances and abstractions from the user's source code. This scripting language is merely a sample; the actual language to be used must be carefully designed and is part of our research. The scripting language must be expressive; that is, it must provide access to the syntactic and semantic structure of source code. At the same time, the underlying program representation must not be exposed to end-users of the transformation tool, as it will hamper their ability to formulate transformations.

To design an appropriate notation, we turn to the field of the psychology of programming, which includes various hypotheses on how programmers construct mental models of programs that they manipulate [3]. However, the existing studies are largely concerned with acquiring higher-level schematic knowledge and constructing domain models. We are conducting a series of experiments to learn how expert programmers formulate update plans for Java programs and what forms these plans take. These studies will both contribute to the existing body of research on psychology of programming and allow us to develop a methodology for working with languages other than Java.

Along with the scripting language we are designing an interactive environment for manipulating and executing update scripts. This environment will enable the programmer to visualize execution of the script, examine each transformation site, selectively undo or modify individual transformations, etc. The environment will also capture the source code change history in terms of high-level manipulations. Such a capability will help to document important aspects of program evolution, as well as support selective rollback of high-level changes long after they had been performed.

The environment will augment the scripting language with direct manipulation. We believe that the scripting language 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, we intend to provide a "by-example" pattern matching mechanism, whereby the user selects language constructs that "look like" the intended match. The pattern can be subsequently abstracted to match a larger class of code fragments.

An important advantage of using an integrated environment for transforming source code is the ability to treat the update scripts as abstractions. This permits naming scripts, storing them in a library for reuse, and treating scripts as update agents. An update agent is a metaprogram bound to both the source and the target (generated) program elements. An integrated environment can track dependencies between the two sections of source code and act appropriately if the developer makes changes to either.

3. RELATED WORK

A number of key issues differentiate existing systems for source-to-source transformations, metaprogramming, and source code refactoring from the kind of support for interactive program manipulation that we envision.

A broad class of tools allows the developer to specify transformations in a general-purpose notation. Such tools range from the primitive text processing tools, such as the Unix *sed*, to the tools that operate on the lexical structure of the program (for example, LSME [7]), to the full featured program transformation tools that operate on annotated syntax trees (for example, ASTLog [2]). However, text- and lexeme-oriented tools are not

sufficiently expressive, whereas tree-based tools are not useful to anyone unfamiliar with a tree-based program representation. Furthermore, such tools do not include a facility for evaluating or visualizing pattern matching or selectively reversing or modifying transformations, thereby impeding high-level interactive operations.

Metaprogramming tools form another class of transformation systems (AspectJ [6] is a good representative). Such tools are designed for compiler-like use, inhibiting further manipulation of the result of the transformation. Moreover, these tools work by extending the underlying programming language, limiting their applicability and deterring their acceptance by the industry.

Many interactive development environments offer facilities for performing refactoring transformations, such as those catalogued by Fowler [5]. However, many refactoring transformations cannot be generalized to arbitrary contexts, and, therefore, cannot be automated (for example, moving a non-static method between two unrelated classes).

4. EVALUATION

Our interactive transformation environment is being prototyped on the Eclipse platform [4], an open-source framework for building interactive development tools. Implementing general source-to-source transformation facilities requires a solid program analysis infrastructure. Such infrastructure is not available in the Eclipse platform and will be contributed to Eclipse through integration with the Harmonia program analysis framework [1].

Our prototype will be evaluated against several criteria, including current mechanisms for manipulating source code in an interactive setting, usability studies of the resulting prototype, and the outcome of deploying our Eclipse-based implementation. The system will be instrumented to collect information about how it is used. We will study users carrying out a fixed set of modification tasks with and without our tools, and will compare such factors as modification time, user-perceived ease or difficulty of the task, and quality of the resulting transformations.

5. REFERENCES

- [1] Boshernitsan, M. *HARMONIA: A Flexible Framework for Constructing Interactive Language-Based Programming Tools*. Technical Report. University of California, Berkeley. UCB/CSD-01-1149, 2001.
- [2] Crew, R.F. *ASTLOG: A language for examining abstract syntax trees*. In proceedings of the First Conference on Domain Specific Languages, 1997, p. 229—242.
- [3] Detienne, F. *Software Design: Cognitive Aspects*, Springer Verlag, 2001.
- [4] Eclipse Consortium. <http://eclipse.org/>
- [5] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.
- [6] Kiczales, Gregor, et. al., *Aspect-Oriented Programming*. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag LNCS 1241. 1997.
- [7] Murphy, G.C. and Notkin D. *Lightweight source model extraction*. Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering, 1995.