Aligning Development Tools with the Way Programmers Think About Code Changes

Marat Boshernitsan*
Agitar Software, Inc.
450 National Ave, Suite A
Mountain View, CA 94043
marat@agitar.com

Susan L. Graham
Computer Science
University of California
Berkeley, CA 94720-1776
graham@cs.berkeley.edu

Marti A. Hearst
School of Information
University of California
Berkeley, CA 94720-4600
hearst@ischool.berkeley.edu

ABSTRACT

Software developers must modify their programs to keep up with changing requirements and designs. Often, a conceptually simple change can require numerous edits that are similar but not identical, leading to errors and omissions. Researchers have designed programming environments to address this problem, but most of these systems are counter-intuitive and difficult to use.

By applying a task-centered design process, we developed a visual tool that allows programmers to make complex code transformations in an intuitive manner. This approach uses a representation that aligns well with programmers' mental models of programming structures. The visual language combines textual and graphical elements and is expressive enough to support a broad range of code-changing tasks. To simplify learning the system, its user interface scaffolds construction and execution of transformations. An evaluation with Java programmers suggests that the interface is intuitive, easy to learn, and effective on a representative editing task.

Author Keywords

Transformations, visual languages, cognitive dimensions.

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces - interaction styles, user-centered design

INTRODUCTION

Software developers often need to make numerous systematic changes throughout the source code. These changes can be motivated by design refinements, bug fixes, and maintenance updates, such as converting library calls using an old API to a new one. Many conceptually simple changes can have far-reaching effects, requiring numerous similar but not identical edits that make the modification process tedious and error-prone. Not surprisingly, there is considerable interest in creating tools to help automate this transformation process.

CHI 2007, April 28-May 3, 2007, San Jose, California, USA.

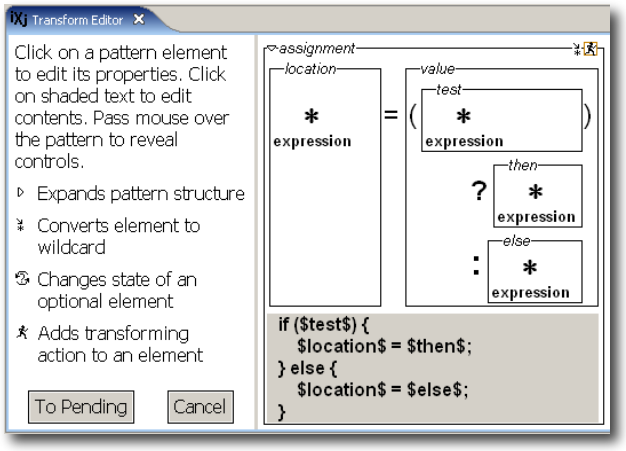


Figure 1. A new tool enables software developers to specify and execute code-editing transformations.

Automated refactoring tools in modern interactive development environments (IDEs) support certain often-used transformations. These tools are lightweight and easy to use, but they offer a limited range of transformations and do not allow programmers to create their own. Powerful and flexible tools that support formally-specified program transformations exist, but are rarely adopted. Using these tools is challenging for the average programmer because they require manipulation of complex representations—such as abstract syntax trees—which bear little resemblance to the programmer's intuitive understanding of programs. Research in the psychology of programming has repeatedly confirmed this observation (see Detienne [12] for a summary).

The key contribution of our work is a novel program manipulation paradigm that enables programmers to change source code with interactively-constructed visual program transformations. Our approach bridges the gap between formal manipulation of source code structure and lightweight refactoring, making transformations accessible to programmers and helping reduce the effort expended on mundane code-editing tasks.

Figure 1 shows a visual program transformation that changes a conditional assignment in a Java program

*The work presented in this paper was done while the first author was at the University of California, Berkeley.

(the pattern at the top of the figure) to an `if`-statement (shown in the shaded part) to improve readability. By employing user-centered design techniques, we have developed a visual language for representing structural transformation patterns. This design, motivated by the *Cognitive Dimensions of Notations* framework [16], allows programmers to perform direct-manipulation of a representation that aligns well with their mental model of source code. To help programmers learn how to apply the visual transformations, we have created a user-interface that scaffolds construction and execution of transformations. The design of the visual transformation language and of the accompanying user-interaction model represents the second contribution of this research. Extending the traditional evaluative use of the Cognitive Dimensions framework to earlier design phases is our third contribution.

Our fourth contribution is the integration of a prototype visual transformation tool into a professional IDE. Tight integration with a traditional development environment allows the use of transformations in-line with other coding activities. Experience suggests that many programmers are hesitant to use tools that require them to "step outside" their usual programming environment.

We evaluated our system through a formal usability study with five proficient Java programmers, finding they were able to learn the language quickly and use the environment effectively to complete a representative code editing task. On average, they responded positively to subjective questions about the intuitiveness, expressiveness, ease of use, and ease of understanding of the system. The participants indicated that while sometimes they made small errors that were frustrating, the system made it easy to stop in the middle of a transformation, check their work so far, and to explore various options before making changes. The results of our study support our hypothesis that an intuitive program transformation tool can reduce the effort expended on making certain types of changes to source code.

In the remainder of this paper, we describe related work, the theoretical framework, and our design process. Then we describe the current design in depth, describe the usability study, and present conclusions.

RELATED WORK

Source Code Transformation Tools

The most ubiquitous and the least sophisticated approach to program transformation is text substitution. Most modern program editors offer this facility, as do several command-line tools, such as `sed` and `awk` [14]. The major weakness of the text-substitution approach is its treatment of source code as flat, structureless text. For example, a structure-based transformation, such as the one shown in Figure 1, cannot be expressed with text-substitution tools. Regular-expression patterns can be difficult to read (see Figure 2a for a `sed` example) and can cause difficulties for the users once their complexity extends beyond the trivial [4].

```
s/(int)([_a-zA-Z][_a-zA-Z0-9]+).nval/1.nextInt()/g
```

(a)

```
rule transformNextInt
    replace [expression]
        (int) E [id] C [repeat component]
    by E C [transformNextIntInComponent]
end rule

rule transformNextIntInComponent ...
```

(b)

Figure 2. Two examples of describing source code transformations using (a) Unix's `sed` utility and (b) TXL.

Several systems provide structure-based source code manipulation primitives. Early tools, such as A* [20] and TAWK [17] expose syntax trees as primitive data structures. Other tools use more abstract representations based on the algebraic data types and term pattern matching. (Stratego/XT [29] is a relatively recent representative example.) While tree-based tools can be powerful, working with them requires understanding low-level program representations. Most ordinary programmers lack the skill needed to use these tools.

Some transformation tools attempt to hide these low-level details by using an extended syntax of the underlying programming language, improving readability and maintainability of transformations. Representative examples in this category include REFINE [8], DMS [3], and TXL [11]. Still, these tools are difficult to use (see Figure 2b for a TXL example) and are utilized only for highly specialized tasks such as fixing the Y2K bug [10] or porting software [30].

By contrast, automated refactoring tools, popularized by the Refactoring Browser [27], offer high-level program manipulation primitives without exposing any program structure. Programmers find refactoring tools indispensable for broadly applicable transformations, such as "rename variable" or "extract method." However, these tools are limited in the types of transformations they support because not all refactoring transformations can be automated and because not all useful transformations constitute behavior-preserving refactoring [26].

Transformations in Structured Documents

Several researchers suggest treating programs as structured text documents by translating them into the XML format [2, 9]. This allows to perform code-changing transformations with an XML transformation tool such as XSLT [1] or XDuce [18]. Unfortunately, these tools again require familiarity with a complex transformation language and with the underlying program's representational structure. Encoding type and scope information in XML can be difficult [7] and can complicate transformation patterns.

Tools such as VXT [25] and Xing [15] offer visual environments for transformation of XML documents based on a treemap representation. However, since these tools

were never intended for programs, their pattern representation is geared toward data sets that impose no spatial structure.

Editing By Example

Programming by demonstration (PBD) is an old and recurring theme in HCI research. Over the years, researchers have developed several PBD tools for automating text editing tasks. These tools, known as *edit-by-example* systems, offer a solution for repetitive editing.

All edit-by-example systems infer abstract editing actions from a few concrete examples and apply these actions throughout the text. Some systems construct an abstract editing model by mapping "before" and "after" blocks of text (EBE [24]) or by inferring text editing operations such as insertions and deletions (TELS [31]). Other systems infer the structure of the text from multiple text regions (LAPIS [23], Visual AWK [21]), permitting subsequent modifications to be made on the common parts across multiple similar regions of text.

Most edit-by-example systems are not aware of the explicit document structure, making them unsuitable for most non-trivial source code transformations. Although some of this structure can be inferred given sufficiently many examples, the need for the programmer to locate these examples obviates much of the appeal of automated editing. LAPIS includes a pattern language that makes the search easier, but having to specify patterns manually still lessens the simplicity of the edit-by-example approach. Moreover, LAPIS's simultaneous editing interface is not sufficiently flexible for complex restructuring common in program source code.

THEORETICAL FRAMEWORK

Our design process was motivated by Green's framework of Cognitive Dimensions of Notations (CDs) [16]. The CDs framework offers techniques for interface evaluation and provides guidance on how the design can be improved. The framework includes fourteen usability dimensions, such as *visibility*, *consistency*, and *error-proneness*. The dimensions are not independent; for instance, an improvement to the consistency of an interface can lead to an increase in error-proneness. Together, the dimensions determine a *cognitive profile* of a system. This profile does not measure the quality of the system; different activities and different systems need different profiles.

Designers usually apply the CDs framework as part of a later-stage usability evaluation using the CDs questionnaire [5]. In our work, we also use it for needs assessment, evaluation of early design mockups, and as part of the final usability evaluation. From our needs assessment (described below), we identified the following cognitive dimensions as essential for the design of an intuitive transformation tool:

- **High visibility.** The interaction with the tool must be visually rich. It should be possible to examine both the pattern and the transformation through a visual representation understandable by the programmer.
- **Moderate diffuseness.** The notation for specifying transformations should not be overly terse so as to make it opaque to programmers new to the tool. At the same time, it should not be overly verbose, requiring long descriptions for simple transformations.
- **Low error-proneness.** The programmer should be prevented from making errors whenever possible; however, error prevention must never happen at the cost of usability. If the programmer happens to introduce an error, it should be easy to detect and fix.
- **Closeness of mapping.** It should be reasonably easy for the programmer to map a structural transformation to his conceptual understanding of the program structure.
- **Progressive evaluation.** The programmer should be able to check the results of a transformation-in-progress by examining its effect on the entire body of source code before "committing" the transformation.
- **No premature commitment.** It should be possible to manipulate patterns and transformations in any order. Any transformation must be easily reversible after it is applied.

DESIGN PROCESS

Task and User Analysis

We conducted two informal studies as the initial step of our task-centered design process. First, we analyzed programmer-provided verbal descriptions of their own changes in several software systems. We studied the development log for TEX [19] (a popular typesetting package), the change log for XEmacs[1] (a text editor for programmers), and a pair-programming transcript recorded by Martin and Koss [22]. Second, we conducted an experiment to help reveal how programmers describe small transformation steps to one another. In that experiment the participants were shown "before" and "after" snapshots of source code and were asked to write down a description of each change. We were particularly interested in how programmers reference code fragments, how they describe the output, and what programming style they use.

Our analysis and observations led to four early design decisions. First, we decided to use high-level programming concepts, such as variables, methods, and loops, as the building blocks of our transformation language, because the programmers talked about these concepts in their change descriptions. Second, we noticed that the programmers used patterns to describe classes of similar changes, for example "change `BI_*` macros to `BYTE_*`." This suggested that a rule-based transformation language, consisting of "patterns" and "actions" would fit naturally into the programmers' thinking about systematic changes. Third, when faced with a transformation that incorporates scoping information, the programmers treated it implicitly. For example, the intent of "rename 'x' to 'y'," usually is to rename only those instances of

[1] `http://www.xemacs.org`

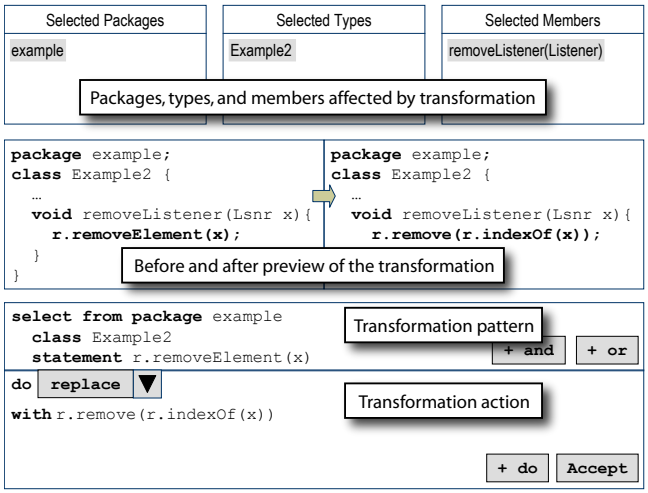


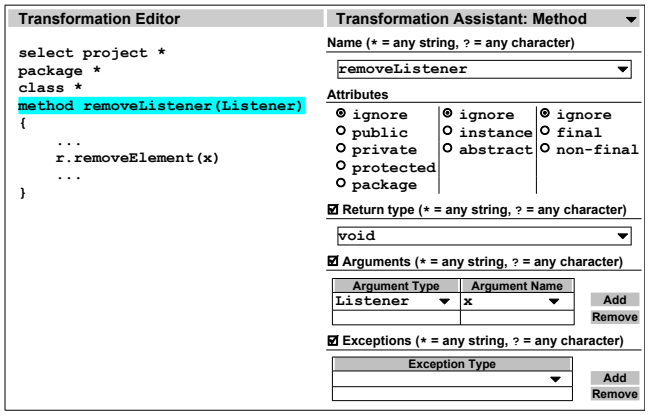
Figure 3. First mockup of the transformation editor. The transformation shown here rewrites calls to the `removeElement()` method in `java.util.Vector` as a combination of calls to `remove()` and `indexOf()`.

'x' that are in the same scope. It should be possible to express this intent in the language that describes transformations. Fourth, we observed that the terminology used by programmers was specific to their programming language. As our goal was to develop a transformation tool for Java programmers, our design is tightly coupled with Java. We expect that our approach can be applied to other programming languages.

Early Designs

We created four mockups of the transformation tool prior to finalizing the design and building the first prototype. The first design attempt was inspired by SQL, a database query language. We borrowed the notion of a "selection," described by a `select`-statement that selects statements for transformation. Classes, packages, and methods are selected using name-based patterns. Statements and expressions are selected using code patterns. An editing action describes how to transform code fragments that the patterns match. In this design, programmers start by selecting a single code fragment that they want to change. The transformation editor provides assistance by creating an identity transformation that changes nothing. The programmers use that transformation as a starting point for making it more general. The transformation editor assists with this process by offering context sensitive help and showing the effects of the transformation after each modification step. See Figure 3 for one of the screens in this mockup.

We evaluated this first mockup with a group of researchers. They applauded the from-example interaction model, but disliked the design of the transformation language. We then evaluated the transformation description language using the Cognitive Dimensions framework. First, we observed that the transformation language exhibits high *diffuseness*. It takes too much code to describe a simple transformation. Second, the notation introduces *hidden dependencies*—the name of a pattern variable defined in the pattern must correspond to its use in the action. This increases the *viscosity* of the description: a change to a name in the pattern must be propagated to all places in the action where that name is used. Third, the description language is *error prone*—it is possible to introduce errors by mistyping a pattern or an action. Fourth, *closeness of mapping* at the level of package, class, and method patterns is not very good because the structure of the `select`-statement does not follow the structure of declarations in Java programs.



(a) Pattern specification in an SQL-like notation. This example includes a mockup of a context-sensitive interface that assists the user with transformation patterns.

```
package edu.[[*cs]].berkeley.example;

class [<Name>] {
    void removeListener(Listenter x) {
        [<Statement>*]
        [<Expression>:Vector].removeElement([<Args>]);
        [<Statement>*]
    }
}
```

(b) Pattern specification based on the Java syntax. Text patterns are enclosed between double square brackets (`[[` and `]]`); structural patterns are bracketed by `[<` and `>]`. Patterns with `*` indicate repetition.

Figure 4. Two early examples of the pattern language design. These patterns match calls to the `removeElement()` method for the transformation described in Figure 3

In the second iteration we attempted to resolve the problems with the first design by implementing a user-interaction model that provides better scaffolding for building transformation patterns (see Figure 4a). We introduced the notion of a context-sensitive transformation assistant that appears as a separate pane to the right of the free-form text editor. The assistant lists the actions that can be performed in the context of the cursor's location in the editor. The transformation editor maintains the consistency between the transformation description and the selected options in the assistant; changes to one update the other.

This design received mixed feedback. The audience liked the transformation assistant and appreciated how it helps the programmer learn the transformation lan-

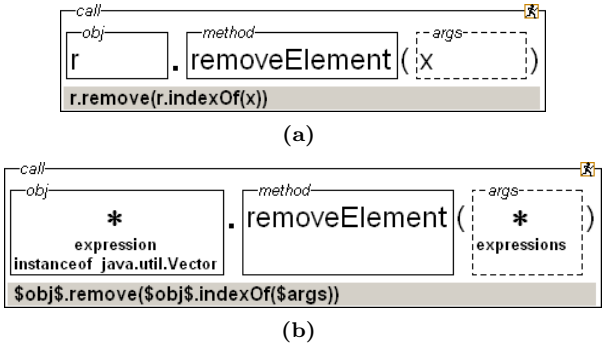


Figure 5. Two transformations that rewrite calls to the `removeElement()` method in `java.util.Vector` as a combination of method calls to `remove()` and `indexOf()`.

guage. But the patterns were still too difficult to understand. It was not clear how the programmers would know the sequence of the steps necessary create a generalized pattern that matches all code fragments they want to change.

As a result of the second evaluation, we abandoned the SQL-inspired format in favor of a language that resembles Java source code. We augmented Java syntax with syntactic escapes to the pattern language for describing wildcards, pattern variables, and matching conditions. Figure 4b shows an example of this design.

After writing down several transformations in this language, it became clear that representing structural patterns in a text-based form is awkward. The transformation patterns became difficult to read, defeating the reason for basing the pattern language on the Java syntax. We solved this problem by moving to a hybrid visual transformation language that combines textual and graphical elements. This language became the foundation of the current design, which we call iXj.[2]

CURRENT DESIGN

Visual Language for Program Transformations

Figure 5a shows a simple pattern that matches calls to the `removeElement()` method in `java.util.Vector`. This method takes an "element to remove" as an argument; our transformation will replace calls to this method with calls to the `remove()` method whose argument is the *index* of an element to remove. (This is the same transformation as in Figure 3.) A pattern in iXj is represented as a code fragment surrounded with a labeled graphical box identifying that fragment's programming language structure. The boxes are spatially arranged in such a way that the pattern appears as a Java code fragment surrounded with rectangles that demarcate structural entities. In addition to specific code fragments, patterns can contain wildcards (see Figure 5b). Wildcards are denoted by a wildcard box containing the `*` symbol. Each wildcard box is annotated with the type of the structural element that it can match (here, "expression"). Optionally, a wildcard box can include additional matching constraints. For example, the first wildcard in Figure 5b matches those expressions whose type is a subtype of `java.util.Vector`. Figure 1 shows another example of a pattern that matches all assignments with a conditional expression (`?:`) as its right-hand-side.

Any pattern element can be associated with a transforming action, which is displayed in a shaded input field at the bottom of the pattern box. In contrast to the mostly structure-based patterns, an action specifies the replacement text, including, if needed, references to fragments of the matched pattern. For example, the action in Figure 5a simply rewrites calls to `r.removeElement(x)` with `r.remove(indexOf(x))`. A more complicated action in Figure 5b substitutes the instance object in the method call (`$obj$`) and the arguments to the `removeElement()` method (`$args$`) by referring to the labels of the corresponding pattern boxes. The transformation in Figure 1 replaces a conditional assignment with an equivalent `if`-statement.

User Interface for Program Transformations

Figure 6 shows a screenshot of the iXj prototype implemented inside Eclipse, a popular professional IDE.[3] Programmers interact with the iXj transformation tool through a view below the standard Eclipse editor (see Figure 6b). The visual transformation editor (6b-center) represents the main part of the transformation tool; two other panes, the transformation assistant and the list of "pending" transformations, are described subsequently.

Creating a graphical representation for a transformation from scratch can be challenging even in an interactive environment. Our user-interaction model scaffolds this process through from-example construction and iterative refinement. From-example construction enables programmers to start with a simple transformation that applies to just one location in the source code and to generalize that transformation to apply to other similar code fragments. The immediate feedback provided in the user interface guides iterative refinement: at any point in the transformation construction process the programmer knows what source code is affected and how it will be modified (see Figure 6a).

To illustrate the process of building a transformation, we will use a simple transformation that applies de Morgan's law to a bitwise-`and` operation. For example, `x = a & b` is replaced with `x = ~(~a | ~b)`.

The programmer initiates the transformation by selecting a sample code fragment to change. The selection is unconstrained; however, the transformation editor activates only when the programmer selects a structurally complete code fragment, such as an expression. When this happens, the system automatically generates an initial pattern from the code selection (see Figure 7a).

The initial pattern matches the exact code fragment that the programmer selected and all the other code fragments that are textually similar. Nested-patterns

[2] iXj – Interactive TRANSformations for Java

[3] `http://www.eclipse.org/`

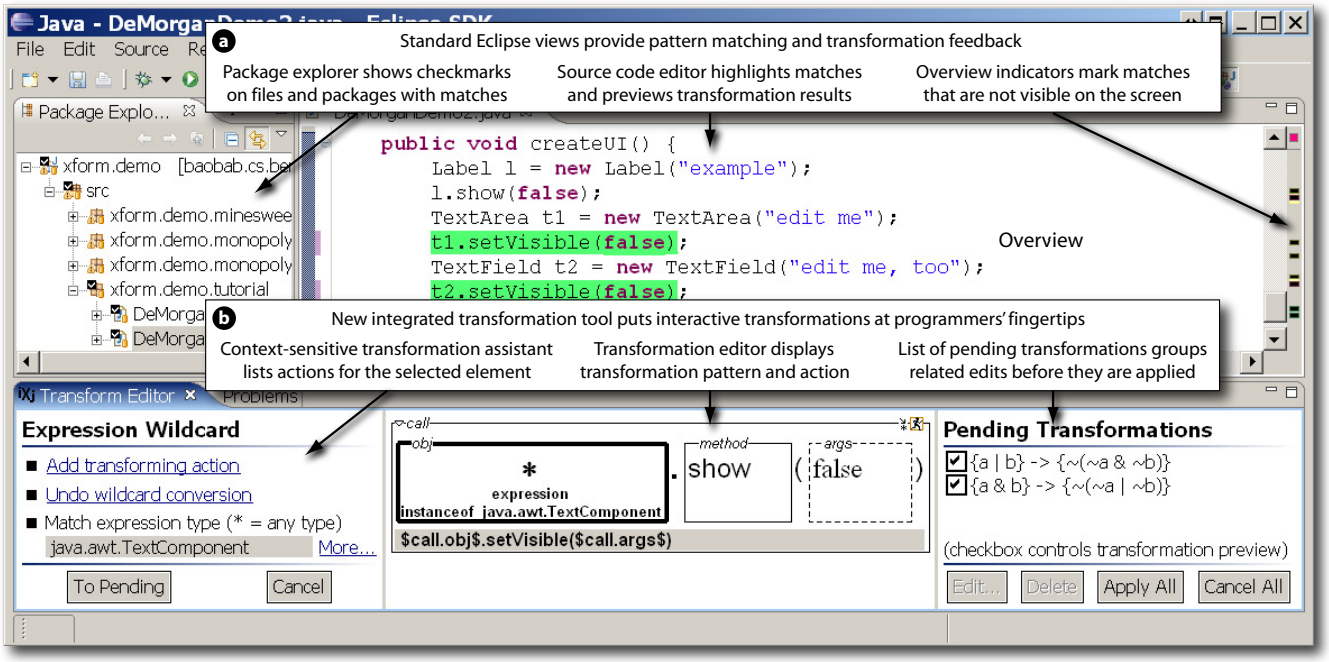


Figure 6. Interactive transformation tool as a fully-integrated extension for Eclipse IDE. The transformation in this figure replaces calls to a deprecated Java library method called `show()` with an equivalent method called `setVisible()`.

are not shown initially to reduce visual clutter. The programmer can manipulate the transformations directly by invoking controls that modify pattern structure.

The controls that appear in the top of each pattern box permit expansion and collapse of the pattern structure (see Figure 7b), conversion of a pattern element to (and from) a wildcard (7c), and addition (and removal) of a transforming action (7d). Any operation can be reversed by re-activating a control. To reduce visual clutter, the controls appear on the screen only when the programmer passes a mouse pointer over the box.

The transformation editor permits free-form text editing of the transforming action and textual parts of the transformation pattern—the programmer can edit the action simply by clicking on its text (see Figure 7e).

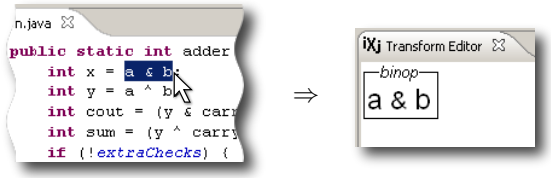
In addition to the transformation editor, Figure 6b shows two other elements of the iXj user interface: the transformation assistant (6b-left) and the list of pending transformations (6b-right). When the programmer selects a pattern element (by clicking), the context-sensitive transformation assistant describes that element and lists various actions, including those not available through direct manipulation. The list of pending transformations helps to avoid intermediate inconsistent states of the source code by grouping related transformations together prior to application. The effects of pending transformations can be previewed, but they are not applied until the programmer decides to do so. Often, when working on a related transformation, the programmer realizes that one of the pending transformations is incomplete and continues to modify that transformation until it behaves as expected.

Design and Implementation Influences

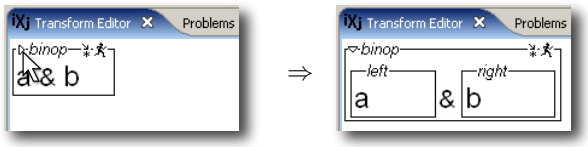
The design of iXj incorporates several influences from existing systems. The transformation language includes elements from other pattern languages, such as the * symbol for a wildcard, because these elements are likely to be familiar to users. Our interface design embodies some of the same principles as the edit-by-example systems: iterative refinement and immediate feedback guide the user in creating the transformation pattern.

iXj's graphical notation was inspired in part by diSessa's Boxer [13], a Logo-like visual programming language and a computational environment. Boxer's computational element is a box that comprises a visual presentation and computational semantics. Unlike Boxer, iXj descriptions do not represent computations; they merely reflect static structure of Java programs. In departure from Boxer's free-form spatial metaphor, the transformation language enforces strict layout rules on the position of pattern boxes on the screen. This is necessary to align pattern boxes so that the pattern looks like source code. And while iXj's transforming actions are similar to Boxer's computation semantics of boxes, Boxer is a complete visual programming language, whereas iXj only allows simple rewrites of source code.

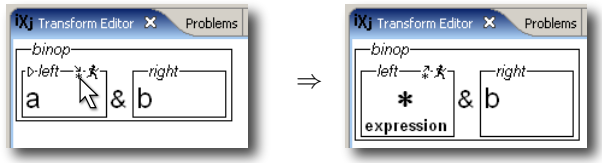
The availability of a free and open development platform, such as Eclipse, was instrumental to this research. Several elements of the Eclipse user interface influenced the implementation of the iXj interaction model. For example, the checkmarks on matching files and packages (see Figure 6a-left) and the overview indicators (6a-right) are both features of the Eclipse workbench. Eclipse also provided a familiar development environment for the participants in our evaluation.



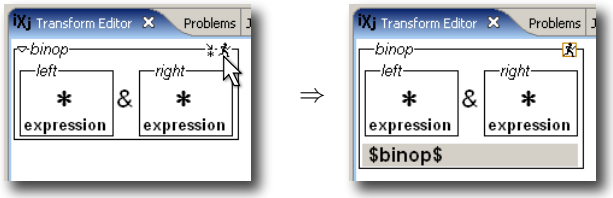
(a) iXj generates the initial pattern from selection.



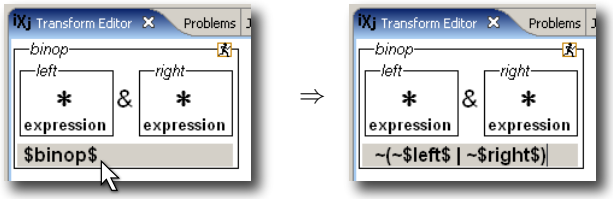
(b) "Triangle" control expands the pattern structure.



(c) "To-∗" converts pattern elements to wildcards.



(d) "Running-man" creates the initial do-nothing action.



(e) Users edit the action as a text-input field.

Figure 7. Key elements in the iXj's interaction model.

EVALUATION

In order to assess the visual transformation language and the from-example interaction model, we conducted an evaluation of the Eclipse-based transformation environment through a usability study with five Java programmers. We trained the participants to understand, construct, and evaluate transformations. The participants completed a short code editing task and filled out an evaluation questionnaire.

Experimental Setup

Each evaluation session consisted of four major components: (1) a 20-minute pre-study interview to assess the participants' familiarity with major concepts in source code maintenance, (2) a 20-minute training session in which the participants learned the transformation language and the user-interaction model of the transformation environment, (3) a 30-minute block of time allotted for the participants to complete a code-editing task, and (4) a 20-minute post-study interview, ending with the participants completing a questionnaire. The entire session was recorded for later analysis.

Participants

The participants in our study were proficient Java programmers with various levels of Eclipse experience. Our main selection criterion was Java proficiency—we specifically wanted to avoid novices who may not have enough experience with code maintenance tasks. Three participants were professional Java programmers employed in the software industry. Two were students (one graduate and one undergraduate) in the Computer Science Department at the University of California, Berkeley. Four participants were male and one was a female.

All participants considered themselves expert Java programmers, with an average of eight years of Java experience. Two participants were self-reported Eclipse novices having little familiarity with its automated refactoring facilities. Three participants use Eclipse for their day-to-day Java programming.

The pre-study interview was aimed at establishing common terminology and understanding of source code maintenance. We defined maintenance as any programming activity that does not involve adding new code to a software system (authoring). We distinguished three forms of maintenance: adaptive (adding new features), corrective (fixing defects), and perfective (anticipating future changes). These definitions coincide with those established in software engineering literature [28].

During the pre-study interview all participants reported regularly performing adaptive and perfective source code maintenance activities. Three participants estimated that they spend 20% of their coding time on source code maintenance, two participants reported that fraction to approach 40%-50%, and one participant estimated that 80% of her time is spent performing some form of maintenance of existing code. All participants reported using some tools to assist them with these tasks. Of these tools, the Java compiler was considered the most ubiquitous for its ability to locate places in source code that are semantically or syntactically inconsistent after a change. The participants indicated that they often structure their maintenance activities to intentionally cause compilation errors by starting with the most disruptive change. This practice enables them to use the resulting compiler error messages as a "to-do" list. (One participant referred to this as a "chasing the errors" approach.) Three of the participants reported routinely using refactoring tools in Eclipse to assist them with code maintenance tasks. Only one of the participants was comfortable using command-line tools (such as the `sed` and awk utilities), although all participants were aware of these tools.

We trained participants via a detailed walkthrough of several transformation examples using the prototype.

Transformation Task

We asked participants to perform a code maintenance task on a console-based implementation of the MineSweeper game. The existing implementation of MineSweeper relied on the `java.io.StreamTokenizer` class to process console input. We asked the participants to convert the uses of `StreamTokenizer` to the uses of `java.util.Scanner`.

Because we did not expect the participants to be familiar with the `StreamTokenizer` and `Scanner` APIs, we gave them a listing of both APIs and an example of transforming the uses of one API into another. We told participants that they should not interpret the transformations in the example literally and that transformations in the MineSweeper source code will have slightly different shapes than those appearing in the example.

Metrics

During evaluation we measured time to completion for each transformation. We also recorded total time to completion of the entire task, but we did not find that metric useful—two participants decided to perform several transformations that were not on the list, because "they seemed appropriate."

Following completion of the sample task, the participants were asked to evaluate the transformation tool by completing a twelve-item questionnaire consisting of both qualitative and quantitative questions. During our analysis of the results we were trying to determine (1) the understandability of the transformation language vocabulary, (2) the intuitiveness of the pattern structure, and (3) the ease of developing transformations. We were also interested in classifying the most common mistakes that the participants made while attempting to complete the code editing task.

The questionnaire was constructed using the Cognitive Dimensions framework and was adapted from Blackwell and Green's questionnaire optimized for users [5]. We augmented the traditional qualitative CDs questionnaire with a seven-point semantic differential scale. Because we also applied the CDs framework in the early stages of our design, the CDs questionnaire provided feedback on our design targets, as well as the overall usability picture.

Hypothesis

Our hypothesis was that the participants would find the transformation tool intuitive and easy to use. We expected them to perform well on the sample transformation task and to become reasonably proficient with the tool. After a brief exposure to the transformation tool the participants should understand how to build a pattern, how to create an action, and how to evaluate correctness and completeness of a transformation.

Performance Results

Figure 8 lists time (in seconds) spent by each of the participants on each of the transformations in our task. We recorded both the time spent on the first attempt to construct a transformation ("Init") and the time spent on any subsequent modification ("Fix"). Subsequent modifications were necessary because some of the participants made mistakes on the first attempt without realizing it. Later, they went back to an earlier transformation from the pending transformation list to correct their mistakes. We also computed total transformation time, though there was a great amount of variation among the participants depending on the order in which the participants attempted the transformations.

	Participants				
Transformation	1	2	3	4	5
T1 Init	135	88	157	112	91
Fix	28	37			41
Total	**163**	**125**	**157**	**112**	**132**
T2 Init	84	174	93	136	219
Total	**84**	**174**	**93**	**136**	**219**
T3 Init	75	80	43	91	44
Fix		8			
Total	**75**	**88**	**43**	**91**	**44**
T4 Init	46	48	63	–	–
Fix	22				
Total	**68**	**48**	**63**	–	–
T5 Init	131	118	–	55	102
Total	**131**	**118**	–	**55**	**102**
T6 Init	16	138	94	47	39
Fix	166			20	60
Total	**182**	**138**	**94**	**67**	**99**

Figure 8. Time in seconds spent by each of the participants on each of the transformation tasks. "Init" is the time spent on the initial attempt. "Fix" is the time spent on a subsequent correction, if any. "Total" is total time spent for a transformation. Some times are absent because not all participants attempted all transformations.

While drawing general conclusions from these times is dangerous because not all transformations are of the same difficulty, we observed that participants' fluency increased with experience with the tool. This can be seen from the times for transformations T2 and T3: every participant attempted these transformations in sequence because the targets of the transformations occur close together in the source code. These transformations are comparable in pattern complexity, and the second transformation in the sequence was completed more quickly than the first by every participant.

Cognitive Dimensions Evaluation

We evaluated the transformation tool along twelve cognitive dimensions, but for brevity report on only six here. The participants were given an opportunity to include a verbal description of the issues related to each dimension. Boshernitsan [6] provides a detailed discussion of the questionnaire results.

The participants felt that *visibility* of the transformation description language was good and that it was easy to see and find various parts of the transformation description as they were working with it. One participant expressed concern about the long name references in the transformation action (and two others noted this problem in response to a different dimension). The participants noted that *viscosity* (resistance to change) was low, with one participant observing that making

changes was "much easier than [he] expected." Another participant appreciated the ability to make changes to the completed transformations by taking them out of the pending transformation list. The participants reported moderate *diffuseness*, indicating that the transformation descriptions were reasonably concise, while remaining easily readable.

iXj received high marks on *role expressiveness*—when looking at the transformations it was easy to tell the purpose of each element in the overall scheme. The participants stated that "it is obvious where each part comes from," and thanked us for "not showing these as a tree." The participants reported that we achieved reasonable *closeness of mapping* between the transformation description and their understanding of the program structure. One participant noted: "The pattern looks, visually, like source code. It makes sense to edit it as an example of the change you want to make and convert things to wildcards where they are unnecessarily specific." Another participant commended our design for "great indication of wildcarding."

Most of the participants were satisfied with iXj's *progressive evaluation* capabilities. One participant noted that he had trouble "mak[ing] sure [he] had grabbed all matches that [he] intended." This problem was also mentioned by other participants in the post-study interview. The participants felt comfortable with exploring various directions because it was "fast to make changes" and they could "see the code change right away." One participant particularly liked "going back and forth from wildcard to the original part [to see] the effect of converting to a wildcard."

Participants' comments on the *error-proneness* dimension indicated that this aspect of our design needs more work. One participant indicated that "[he] was often not sure that [he] made the pattern sufficiently generic." Two other participants noted that it is easy to mistype a variable name in the transformation action. Another participant disliked small icons for pattern box controls.

Most participants felt that they could work on a transformation in any order, avoiding *premature commitment*. One participant indicated that the ordering of several related transformations can be important for organizing one's work, although the tool does not enforce any restrictions.

Common Mistakes, Errors, and Misconceptions

We analyzed screen and audio recording of each of the evaluation sessions to classify participants' mistakes, errors, and misconceptions. Mistakes are the slips that the participants made when constructing transformations. Mistakes were usually corrected at some point during the transformation session, if not immediately. Errors are more fundamental. Often, the participants did not realize that they introduced an error and that their final transformation was incorrect. Misconceptions caused the participants to pause the transformation process and to consult the interviewer.

We traced many mistakes to small issues with iXj's interaction model. For example, forgetting to click outside of the editable text field to cause the input to be accepted is easily remedied. Some mistakes, such as clicking on the wrong icon, were due to the participants' unfamiliarity with the tool. Since the participants were given no time to practice with the user interface, we expect these mistakes to subside with practice.

By contrast, insufficient wildcarding of the transformation pattern emerged as a common problem. Several participants indicated that they had trouble deciding when the pattern has enough wildcards to match all places in the source code needing transformation. This result contrasts with our initial design intuition that the programmers will be able to "reason" about a transformation and will add all of the necessary wildcards based on the intent of the transformation. We plan to fix this by implementing an approximate pattern-matching algorithm. The approximate matches will indicate to the user which variations of the pattern structure exist in the program and which parts of the transformation can be wildcarded to affect more code fragments. This approach will reduce the need for the upfront wildcarding in anticipation of possible matches.

We traced all of the transformation errors to the participants' unfamiliarity with the subject source code and with the `StreamTokenizer` and `Scanner` APIs. These errors were not related to their use of iXj. We observed several misconceptions about some of the structural elements in the transformation description language. Boshernitsan [6] discusses these misconceptions in detail; we intend to address them in the future versions of our design.

Discussion and Observations

The evaluation confirmed our hypothesis: the participants learned the transformation language and the transformation editor quickly and completed the transformation task successfully. During the post-study interview all participants reported that they found the transformation description language intuitive. They confirmed comprehensibility of the exposed source code structure, and indicated that they had no trouble understanding and manipulating the transformation descriptions. Several participants expressed interest in using the transformation tool in their daily source code maintenance activities.

The evaluation strategy was limited in several respects. First, the transformation task presented to the participants was simple and was executed on a small source code base. These restrictions were needed to ensure that the evaluation session could be completed within reasonable time without tiring the participants. Second, the participants only worked on a few hand-selected transformation tasks. Because of this they were only exposed to some aspects of the transformation tool. Given more time with the tool, it is possible that the participants would have discovered additional issues not observed in our study. Third, the number of participants was small,

and so may not be fully representative of the user population. We are planning a more thorough evaluation of the transformation tool through study of programmers using iXj on their own code over a period of time. This study will inform us of the scalability of the visual approach for more complex transformation tasks in real-world systems.

CONCLUSION

We present a tool that enables programmers to perform systematic source code transformations easily and with a high degree of confidence in the results. The design consists of a novel visual language for describing and prototyping source code transformations and of a user-interaction model that simplifies learning and use of this language. We used several user-centered techniques to create a design that aligns the representations used by the tool with the programmers' mental model of programming structures. We built and evaluated a prototype implementation that allows programmers to make transformations in an intuitive manner.

We are currently developing a successor to the iXj prototype that will be publicly available. We will use this tool for further evaluation and refinement of our design.

Enabling programmers to manipulate source code with lightweight language-based program transformations reduces the effort expended on making certain types of large and sweeping changes. In addition to the immediate improvements to programmers' efficiency, this reduction in effort has the potential for lessening programmer's long-term resistance to making design-improving changes, ultimately leading to higher quality software.

ACKNOWLEDGMENTS

We would like to thank the participants in the evaluation of iXj for their time and effort. We would also like to thank the anonymous reviewers for their helpful comments and suggestions. The research described in this paper has been supported in part by the NSF Grants CCR-9988531 and CCR-0098314, IBM Eclipse Innovation Grant, and by Agitar Software, Inc.

REFERENCES

1. XSL transformations (XSLT), version 1.0. World Wide Web Consortium, Recommendation, Nov. 1999.
2. G. J. Badros. JavaML: a markup language for Java source code. *Computer Networks*, 33(1–6):159–177, June 2000.
3. I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proc. of ICSE '04*, pages 625–634, 2004.
4. A. F. Blackwell. SWYN: A visual representation for regular expressions. In H. Lieberman, editor, *Your Wish Is My Command*. Morgan Kauffman, 2001.
5. A. F. Blackwell and T. R. G. Green. A cognitive dimensions questionnaire optimised for users. In A. Blackwell and E. Bilotta, editors, *Proc. of PPIG 13*, May 2000.
6. M. Boshernitsan. *Program Manipulation via Interactive Transformations*. PhD thesis, EECS Department, University of California, Berkeley, July 25 2006. Technical Report UCB/EECS-2006-100.
7. M. Boshernitsan and S. L. Graham. Designing an XML-based exchange format for harmonia. In *Proc. of WCRE '00*, pages 287–289, 2000.
8. S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software reengineering. In *Proc. of ICSAC '90*, pages 314–322, 1990.
9. M. L. Collard, J. I. Maletic, and A. Marcus. Supporting document and data views of source code. In *Proc. of DocEng '02*, pages 34–41, 2002.
10. J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Software engineering by source transformation-experience with TXL. In *Proc. of SCAM '01*, pages 170–180, 2001.
11. J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Comput. Lang*, 16(1):97–107, 1991.
12. F. Detienne. *Software Design - Cognitive Aspects*. Springer Verlag, 2001.
13. A. A. diSessa and H. Abelson. Boxer: a reconstructible computational medium. *CACM*, 29(9):859–868, 1986.
14. D. Dougherty. *sed & awk*. O'Reilly & Associates, Inc., 1991.
15. M. Erwig. A visual language for XML. In *Proc. of VL '00*, page 47, 2000.
16. T. R. G. Green. Cognitive dimensions of notations. In *Proc. of HCI'89*, Cognitive Ergonomics, pages 443–460, 1989.
17. W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proc. of WPC '96*, 1996.
18. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.
19. D. E. Knuth. The errors of TEX. *Software– Practice and Experience*, 19(7):607–681, July 1989.
20. D. A. Ladd and J. C. Ramming. A∗: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, Nov. 1995.
21. J. Landauer and M. Hirakawa. Visual AWK: a model for text processing by demonstration. In *Proc. of VL '95*, 1995.
22. R. C. Martin and R. S. Koss. Engineer notebook: An extreme programming episode. In R. C. Martin, editor, *Advanced Principles, Patterns and Process of Software Development*. Prentice Hall, 2001.
23. R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *Proc. of USENIX ATC '01*, pages 161–174, 2001.
24. R. P. Nix. Editing by example. *ACM Trans. Program. Lang. Syst.*, 7(4):600–621, 1985.
25. E. Pietriga, J.-Y. Vion-Dury, and V. Quint. VXT: A visual approach to XML transformations. In *Proc. of DocEng '01*, pages 1–10, 2001.
26. D. Roberts and J. Brant. Tools for making impossible changes. *IEE Proceedings Software*, 151(2):49–56, 2004.
27. D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.
28. E. B. Swanson. The dimensions of maintenance. In *Proc. of ICSE '76*, pages 492–497, 1976.
29. E. Visser. Program transformation with Stratego/XT. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.
30. R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, Aug. 1988.
31. I. H. Witten and D. Mo. TELS: learning text editing tasks from examples. pages 183–203, 1993.