Program Manipulation via Interactive Transformations

Marat Boshernitsan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-100
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-100.html

July 25, 2006

Program Manipulation via Interactive Transformations

by

Marat Boshernitsan

B.A. (University of California, Berkeley) 1997
M.S. (University of California, Berkeley) 2001

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Susan L. Graham, Chair
Senior Lecturer Michael Clancy
Professor Marti A. Hearst

Spring 2006

Program Manipulation via Interactive Transformations

Abstract

Program Manipulation via Interactive Transformations

by

Marat Boshernitsan
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Susan L. Graham, Chair

Software systems are evolving artifacts. Keeping up with changing requirements, designs, and specifications requires software developers to continuously modify the existing software code base. Many conceptually simple changes can have far-reaching effects, requiring numerous similar edits that make the modification process tedious and error-prone.

Repetitive and menial tasks demand automation. Given a high-level description of a change, an automated tool can apply that change throughout the source code. We have developed a system that enables developers to automate the editing tasks associated with source code changes through interactive creation and execution of formally-specified source-to-source transformations. We applied a task-centered design process to develop a language for describing program transformations and to design a user-interaction model that assists developers in creating transformations in this language. The transformation language combines textual and graphical elements and is sufficiently expressive to deal with a broad range of code-changing tasks. The transformation environment assists developers in visualizing and directing the transformation process. Its "by-example" interaction model provides scaffolding for constructing and executing transformations on a structure-based representation of program source code. We evaluated our system with Java developers and found that they were able to learn the language quickly and to use the environment effectively to complete a code editing task.

By enabling developers to manipulate source code with lightweight language-based program transformations, our system reduces the effort expended on making certain types of large and sweeping changes. In addition to making developers more efficient, this reduction in effort can lessen developers' resistance to making design-improving changes, ultimately leading to higher quality software.

To Isabelle.

Contents

Acknowledgments

The work presented in this dissertation could not have been completed without continuous help and support from Susan Graham, my research advisor. Other members of my committee also contributed in no small way. Michael Clancy suggested early on to consider the vast body of research in the psychology of programming. Marti Hearst critiqued early prototypes of the user interface and encouraged me to think hard about usability.

Michael Van De Vanter has been a colleague, a mentor, and a friend whose thought-provoking insights into software development influenced much of my work. I am indebted to my employer Agitar Software, Inc. and to its CTO Alberto Savoia for allowing me to complete this research despite my job obligations. Bob Evans, Russ Rufer, and Tracy Bialik all helped tremendously by reviewing drafts of my papers and by making suggestions on how to improve my prototypes.

Andy Begel, a comrade-in-arms through most of my graduate career, helped shape my thinking about programming languages and software engineering. Many aspects of this work emerged through our heated and passionate arguments. Other members of the Harmonia research group worked very hard to keep our system together, especially during the past few years when I had to abandon core Harmonia work and concentrate on my dissertation research. Thank you, guys.

I would like to thank my friends and family for their support and encouragement. My parents deserve special thanks for their unwavering belief that one day my formal education will come to a close. Finally, I would like to thank my wife Milla for her love, patience, and understanding. The weekends are finally yours.

The research described in this dissertation has been supported in part by the NSF Graduate Student Fellowship, Sun Microsystems Graduate Fellowship, NSF Grants CCR-9988531 and CCR-0098314, IBM Eclipse Innovation Grant, and by equipment donations from Sun Microsystems.

Chapter 1

Introduction

Software systems are evolving artifacts. From the day the first line of source code appears on the computer screen, the entire software system undergoes constant modification. Initially, most of the changes to a software system are due to the evolving architecture and to the refinement of the design and of the implementation. Further in the software lifecycle, changes are frequently caused by changing requirements, bug fixes, and the addition of new features.

Many changes are simple and isolated. A significant proportion of changes, however, require large and sweeping modification to source code. Making large and sweeping changes to source code can be tedious and error-prone. A conceptually simple modification may require a significant code editing effort. Examples of such changes abound in the many tasks faced by developers. Consider the following:

- A maintenance update to a software system requiring that the use of one library component is replaced with the use of another that provides equivalent functionality through a different API. This change entails finding all of the uses of the old API and systematically replacing them with the equivalent invocations of the new API.
- A change to a widely used function requiring that all calls to that function are enclosed within a guard clause that checks its return value before continuing execution.
- A code cleanup effort requiring functions that return error codes to throw exceptions with the error codes instead.

Performing any of these changes in a large piece of software may take hours of the developer's time and introduce bugs due to the manual nature of the change process.

Repetitive and menial tasks demand automation. Given a high-level description of a change, an automated tool can apply that change throughout the source code. Creating this description can be viewed as a form of *metaprogramming*. A metaprogram is a computational abstraction that operates on some representation of another (target) program. The computational medium of such a metaprogram is a representation of the target program's

source code. The output is a modified version of the target program's source code with all changes applied.

Metaprograms can be constructed using any programming paradigm, such as procedural or functional programming. Describing systematic source code changes, however, is particularly amenable to *rule-based programming*. In this paradigm, the computation is described using declarative rules that consist of patterns and actions. A pattern describes a part of the computational data structure to which the rule applies. An action describes a transformation of that data structure into a new form. For a rule-based metaprogram this data structure is a representation of the target program's source code.

In the context of this investigation we refer to systematic source code changes as *source-to-source program transformations*. Transformations can be construed broadly. In addition to systematically modifying existing code, transformations can also generate and insert new code fragments based on linguistic structure or on meta-information embedded in program source code. The tools developed in this dissertation can support these types of transformations. In the scope of this work, however, we explore the application of program transformations only to systematic source code editing.

1.1 Source Code Manipulation via Program Transformations

Most of the large-scale systematic changes needed for source code evolution require transformations that can be described with a relatively small set of rules. In the first example above, the set of transformation rules consists of a mapping of all uses of the old API to the uses of the new API. While the mapping must be devised through the intelligence of a human developer, a transformation tool can automate the process of editing the program.

The use of program transformations for source code modification is not a novel idea *per se*. A number of source-to-source transformation systems have been publicly available for at least two decades. We present a detailed analysis of the related work in Chapter 2, but two broad categories of the existing tools are worth mentioning here. In the first category are the general-purpose transformation tools that expose their users to a data structure representing the target program. This data structure is often called a *program model*. The users of these tools create transformations that explicitly manipulate this program model with operations that change its structure. In the second category are the automated refactoring tools that provide a small set of pre-defined transformations that are guaranteed to be *behavior-preserving*. Internally, these tools also use a structural program model, but this model is never visible to the user.

Automated refactoring tools have been successfully integrated with modern interactive development environments. These tools, however, are limited in the types of transformations they support because not all refactoring transformations are amenable to automation and because not all useful transformations constitute behavior-preserving refactoring. General-purpose program transformation systems are more flexible. Yet, these systems are used only for highly specialized tasks such as fixing the Y2K bug [22] or porting software [71]. While general-purpose tools can describe a wider variety of program transformations, their use is

sometimes challenging. The refactoring tools are lightweight, simple, and better integrated, making them easy to use for an average programmer. The focus of our research is to bridge that gap.

The main contribution of our work lies in making the concepts behind general-purpose program transformation systems and their versatility accessible to developers for lightweight source code manipulation.

The principal difficulty in using general-purpose program transformation systems stems from these tools exposing a structural model of the target program. Transformation of the program structure is necessary for many source code changes. But understanding a structural model is challenging, because it bears little resemblance to the programmer's intuitive perception of their program's structure. Existing program transformation tools provide no support to the developers for understanding and manipulating that model. Yet, completely hiding this model, as is done in the automated refactoring and restructuring tools, limits the capabilities of those tools. Thus, the developers must rely on some structural representation of source code in order to specify transformations. We posit the following design requirements for any tool that strives to empower developers through program transformations:

- The source code structure that is exposed to the developers for manipulation must correspond closely to the developers' intuitive understanding of source code. This means that parse trees and abstract syntax trees that are used by most language-based tools are not a useful conceptual representation for code editing transformations.
- Regardless of the "naturalness" of the structural representation, we cannot expect developers to possess *a priori* understanding of this structure. Therefore, they must be assisted by a tool that helps them construct and understand transformations.
- This tool must permit code editing by transformation "in-line" with other coding activities. Experience suggests that many developers are hesitant to use tools that require them to step "outside" of their usual program development environment.

Enabling developers to manipulate source code with lightweight language-based program transformations can reduce the effort expended on making certain types of large and sweeping changes. In addition to making developers more efficient, this reduction in effort will lessen developers' resistance to making design-improving changes, ultimately leading to higher quality software.

1.2 Thesis and Scope of This Research

The thesis of this research is that developers can use formal transformations of program source code effectively and that the use of these transformations will reduce the effort expended on mundane and time-consuming code editing tasks.

In the scope of this dissertation we develop a solution based on *interactive program transformations*. Our solution consists of a new developer-oriented language for describing

program transformations and of an integrated environment for constructing and executing these transformations. The transformation language, called iXj,[1] combines textual and graphical elements and is sufficiently expressive to deal with a broad range of code-changing tasks. We built a transformation environment that assists developers with visualizing and directing the transformation process through a "by-example" interaction model. This interaction model provides scaffolding for constructing and executing transformations on a structure-based representation of program source code. We evaluated iXj with developers and found that they were able to learn iXj quickly and to use it effectively to complete a code editing task.

The research presented in this dissertation makes the following contributions:

1. A new program model for Java source code manipulation designed to be understandable and intuitive to software developers.

2. A new visual language for describing source-to-source program transformations. This language, consisting both of graphical and textual elements, exposes our Java program model through a visual mapping between the structural program model and the program text.

3. A novel user-interaction model that scaffolds creation of transformation patterns through by-example construction and iterative refinement, supported by the immediate feedback to the user.

4. A prototype interactive program transformation tool that implements our visual transformation language and our user-interaction model in the Eclipse IDE.

1.3 Outline of the Dissertation

Chapter 2 sets up the context for this investigation. We examine several existing tools that developers can use for automating systematic code editing tasks. We discuss the capabilities and the shortcomings of these systems. We present a case study that demonstrates how a developer can perform a sample code maintenance task with three of the discussed tools.

Chapter 3 introduces the concept of *interactive source code transformations*. We argue that interactivity is essential for enabling developers to use program transformations effectively. We show how we applied task-centered design to develop a language for describing program transformations and to design a user-interaction model that assists developers in creating transformations in this language. We analyze the expected user population, present several representative tasks, explain influences from the existing systems, and discuss three design mockups that we built prior to implementing iXj. In the context of design analysis, we describe Cognitive Dimensions of Notations, a usability evaluation framework that we used both to guide the initial stages of the design and to evaluate a completed implementation with users.

[1] iXj—Interactive TRANSformations for Java

Chapter 4 presents the iXj transformation language. We describe a program model for Java source code that was designed specifically to help developers understand and manipulate program transformations. We discuss the key concepts in the transformation language, describe syntactic elements of patterns and actions, and explain the semantics of pattern matching and action execution. We conclude with a case study that demonstrates the expressiveness of iXj by implementing several transformations that arise during refactoring of object-oriented programs.

Chapter 5 describes our implementation of a source code transformation tool for Eclipse, a popular Java development environment. We describe how users construct iXj transformations using a special-purpose structure editor and how our interaction model assists developers in learning a new language and understanding their transformations. We conclude this chapter with a brief overview of the architecture of the iXj plug-in for Eclipse.

Chapter 6 presents a formal usability evaluation of our implementation. We describe our evaluation strategy, experimental setup, and the evaluation metrics. We show that the participants in our user study quickly understood the concepts underlying the design of the iXj transformation language and were able to use our tool to complete a sample transformation task.

Finally, in Chapter 7 we summarize the results of our investigation and reflect on the success of our work in meeting its goals. We conclude by presenting a number of promising directions for future exploration.

Chapter 2

Tools and Techniques for Changing Source Code

Various proposals have been made for automating systematic modification to source code. Few tools, however, found their way to the "programming trenches." Existing options range from simple text-based substitution to sophisticated language-aware manipulation of source code with special-purpose tools. In this chapter we discuss these options and present several representative systems from a wide spectrum of tools that can be used to automate repetitive and systematic changes. We discuss the capabilities and the shortcomings of these systems, and use three of these tools in a case study of describing a source code maintenance task with program transformations.

2.1 Text-based Transformations

Text-processing tools range from primitive search-and-replace editor commands to sophisticated regular-expression-based processors. Text-processing tools have always appealed to programmers because of their simplicity and their ability to modify large amounts of source code with little effort. Every source code editor supports search-and-replace commands; many modern editors also support regular-expression-based operations, though historically these were available only through standalone tools, such as Unix's SED and AWK [28]. Scriptable editors, such as Emacs [74, 57], combine text-processing capabilities with a Lisp-like extension language that enables creation of editor macros for manipulating source code.

The major weakness of the text-based source code transformations lies in their lack of awareness of the syntactic and the semantic structure of a programming language. Text-based tools treat source code as flat structureless text. For example, when a programmer uses one of these systems to rename a variable, the tool applies the transformation to all syntactic entities having the same spelling, such as functions and string literals. This can lead to unintended results: renaming `foo` to `ear`, also changes `inner_foo()` to `inner_ear()`, and `print("Go you, call hither my fool!")` to `print("Go you, call hither`

`my earl!")`, clearly not what Shakespeare intended.[1] Care must be taken to work around inconsequential variations in the program text, such as the non-uniform use of whitespace and comments that might otherwise hide transformation sites.

Lexically-aware text processing tools, such as LSME [53], bring some language-awareness to text processing. These tools incorporate a lexer that tokenizes the text stream before processing. Tokenization allows patterns on lexical tokens, rather than on individual characters. It also helps to eliminate some of the linguistic mismatches, such as matching the same pattern in a variable name and within a string constant. But this form of pattern recognition is still limited to regular languages, whereas most programming languages exhibit (mostly) context-free structure. For example, regular matching precludes the ability to match an arithmetic expression at a particular nesting depth in the expression structure—a pattern that might arise in the process of simplifying arithmetic expressions.

Text-processing tools use a simple format for describing transformations. The description consists of two parts: the specification of what to replace (the target) and the replacement text. The target may be specified in a variety of ways, ranging from a simple word sequence to a complex regular expression pattern. Typically, the user provides the specification in a text file or on the command line. This specification mechanism is far from ideal. Research has shown that many users have difficulties creating and understanding regular patterns, once their complexity extends beyond the trivial [4]. This problem is partially alleviated in the "by-example" editing systems that have the ability to learn and generalize editing actions from one or more examples. Blackwell's SWYN [4] and Miller's LAPIS [52] are the two most recently developed systems that fall into this category.

2.2 Language-aware Transformations

Despite plain text being the prevailing on-disk representation for source code, much of the language-oriented structure can be recovered through the traditional source code analysis techniques. Some tools can take advantage of that structural information to enable structure-based modification of source code. Broadly, these tools can be classified along a continuum that represents the degree to which the user is aware of the underlying program structure.

At one end of this continuum are the tools that fully expose program structure to the developer. Traditionally, this structure takes the form of a *syntax tree*, representing either the concrete (parsing) syntax or some representation of the abstract syntax (AST). If the syntax tree is attributed with static semantics information, it is often referred to as an *abstract semantic graph* (ASG). At the other end of the continuum are tools that offer high-level program manipulation without exposing any program structure. Object-oriented refactoring tools, an outgrowth of research on the Refactoring Browser for SmallTalk [58], exemplify this approach. These tools are limited to a small set of behavior-preserving transformations.

[1] William Shakespeare. King Lear, Act 1, Scene IV.

This section explores the capabilities and the limitations of language-aware tools for source code manipulation and provides specific examples that fall at different points along the language tools continuum. The tools that we discuss here are merely representative examples of program transformation systems; a more comprehensive and up-to-date list is being maintained at `http://www.program-transformation.org`.

2.2.1 Structure-based Tools

The range of structure-based tools encompasses tree-based tools that operate at the level of the tree data structures and provide the first step up from a purely text-based representations. A* [46] and TAWK [36] are two of the earlier tools that expose a full syntax tree. These tools permit a syntax-based specification of a tree pattern, augmented with an action that "fires" when the pattern matches somewhere in a tree. As close relatives of the popular text processing tool AWK, both tools employ the pattern/action paradigm familiar to AWK users. The pattern language enables a syntax-based specification of the matching predicate. For example, a TAWK pattern matching calls to the `factorial()` method may look as follows:

```
(MethodCall
    (ObjectAccess
        (Expression:$expr ["factorial"])
        ("(")
        (ArgumentList:$param)
        (")")))
```

The actions are specified in an AWK-like statement language (A*) or in the C programming language (TAWK). While these action languages are not specifically directed at source code manipulation, TAWK includes a C language library that offers a number of tree-manipulation primitives.

As source-to-source transformation tools, both TAWK and A* offer some advantages over their text-based brethren. For instance, the ability to express complex, structure-based patterns is useful for describing many changes. But this flexibility comes at a price. Creating patterns in a tree-centered representation requires a good understanding of syntax-tree data structures. Unlike the pattern languages, the action languages offer very little in the way of expressiveness. For instance, the most commonly used operation in the A* action language is a `print`-statement.

A different class of tools traces its roots to the algebraic data types and pattern-matching facilities found in languages such as Haskell and ML. Stratego/XT [66] is a relatively recent representative example; many others were developed over the years. These tools are similar to A* and TAWK in their tree-based modeling of program source code. These tools offer better primitives for expressing transforming actions. Yet, reliance on a tree-centered representation puts the skill required to use these tools beyond that of an ordinary programmer.

Many tools make an attempt to hide the details of tree construction. While internally these tools still operate on tree-like structures, their pattern-matching facilities are often

designed to eliminate some of the complexity associated with manipulating trees directly. Often, the patterns can be specified using an extended syntax of the underlying programming language, which improves their readability and maintainability. In addition to the purely syntactic information, some tools provide access to the static semantics of the source programs. Representative examples in this category include REFINE [12], TXL [23], and DMS [3]. We have used TXL in our case study of performing a code maintenance task with existing tools (see Section 2.3).

Use of these general-purpose transformation tools requires both a good understanding of the programming language syntactic structure and familiarity with a complex transformation language. JaTS [15] is a simpler tool, providing pattern-matching facilities that can interpret source-code-like patterns. Its pattern language, however, is limited by a small number of pattern variables for matching linguistic structures. Some tools, such as Inject/J [32], abandon a syntax-derived representation in favor of a high-level model centered around concepts in the programming language. This approach makes it difficult to specify transformations at the expression and statement level.

2.2.2 Refactoring Tools

The notion of *refactoring object-oriented code* was conceived by Opdyke [54] as a way of describing transformations that occur during evolution of object-oriented frameworks.[2] A distinguishing feature of refactoring transformations is that they are guaranteed to be meaning-preserving. The goal of a refactoring is to improve some aspect of code's design, rather than change its behavior. A description of a refactoring transformation constitutes a recipe for performing a particular change. Fowler [31] provides a comprehensive catalog of common refactoring transformations.

Many modern development environment include facilities for automated refactoring of source code. Notable examples include Refactoring Browser [58] for SmallTalk, IntelliJ IDEA [39] and Eclipse [29] for Java, ReSharper [40] and JustCode! [60] for C# (both plug-ins for the Visual Studio IDE [24]), and Xrefactory [75] for C++. Unlike most other tools discussed in this chapter, refactoring tools offer no mechanism for specifying transformations. Instead, these tools offer developers a set of restructuring commands, such as "rename this variable," that can be applied to the source code to implement a change. Typical interaction with a refactoring tool consists of selecting a target for the refactoring (such as a class, a method, or a block of code) and invoking a refactoring command from the menu. The tool presents the developer with a sequence of dialogs that ask for additional input (such as a new name for a variable) and that display the preview of the transformation. When satisfied with the result, the developer "accepts" the transformation and returns to coding. A refactoring tool applies a number of program analysis techniques to ensure that a transformation is meaning-preserving. If it is discovered that this property does not hold (usually, in a fairly conservative way), the tool refuses to perform that transformation.

[2] The idea that restructuring transformation can be used to implement meaning-preserving changes to source code predates refactoring. For example, Grisword [37] discusses restructuring transformations in Scheme programs.

Unfortunately, not all refactoring transformations are amenable to automation. Often refactoring requires task-specific knowledge in order to update all source code sites affected by a change. For example, consider a fairly common *Move Instance Method* refactoring (Fowler [31], p. 142). This transformation involves moving a non-static method from `SourceClass` to `TargetClass`. Conceptually, this refactoring consists of moving the source code for the method from `SourceClass` to `TargetClass` and updating every call to this method to provide an object of `TargetClass` as the instance argument. The latter part presents a problem for automated refactoring tools. In general, it is not possible to determine where to obtain the instance argument, unless the method is being moved into one of its parameter types (whose instances are known to be available at each call site).

Even when refactoring tools do provide support for the desired change, partitioning a large refactoring effort into smaller tractable pieces still presents a problem for developers. Lippert [49] proposes that the developers construct explicit refactoring plans that break up large changes into a number of individual refactoring steps. Yet, modern refactoring tools do not support any explicit representation for refactoring plans. Furthermore, devising a plan that fits within the confines of the meaning-preserving refactoring paradigm can be difficult for some transformations. We demonstrate this difficulty in the transformation case study presented in Section 2.3.

Despite its limitations, refactoring has proved to be a popular idea—our recent search of an Internet bookstore revealed 16 in-print books on the subject. Most researchers and practitioners agree that tool support for refactoring is crucial. Tokuda and Batory [62] show that a refactoring tool can dramatically improve developers' productivity when doing software maintenance. Modern IDEs are often compared on the basis of the refactoring functionality that they support [19]. One of the reasons for this popularity is the low entry barrier and high addiction factor: once the developers incorporate refactoring into their workflow, it is difficult to resist the temptation to use it to improve some aspect of the code. We find this trend encouraging: if we succeed in making a transformation tool as accessible as the refactoring tools, the developers will embrace it for their daily tasks.

2.3 Code Maintenance with Existing Tools: A Case Study

Software is rarely developed in isolation. Often, a system relies on a multitude of support libraries from various vendors. As these libraries are developed, some of the library components may become outdated and replaced (or supplemented) with newer and better equivalents. Updating a software system to account for such a change entails finding all of the uses of the old APIs and systematically replacing them with the equivalent invocations of the new APIs. Because implementing this change on a large source code base can be tedious, it presents a good opportunity for use of a transformation tool.

In this case study we examine an update to a Java software system that relies on the `java.io.StreamTokenizer` Java library class. This class can be used to separate an input stream into individual tokens according to their syntactic category such as a word or a number. In a recent version of the Java Development Kit (JDK), Sun Microsystems introduced

```
public class StreamTokenizer {
    // Symbolic constants for the token type
    public static final int TT_NUMBER;
    public static final int TT_WORD;

    public int ttype;   // type of last token
    public String sval; // last token as a string
    public double nval; // last token as a numeral (if possible)

    // Reads next token from the input and returns its type
    public int nextToken();
}
```

(a) Excerpt from the interface of `java.io.StreamTokenizer`. Each subsequent token is accessed via the `nextToken()` method. The value of the last token read from the input is stored in the `sval` field and (possibly) in the `nval` field (only if the last token can be interpreted as a numeric value). The type of the token (`TT_NUMBER` or `TT_WORD`) is stored in the `ttype` field.

```
public class Scanner {
    // Tests if more tokens matching the pattern are available
    public boolean hasNext(Pattern pattern);
    // Gets next token matching the pattern from the input
    public String next(Pattern pattern);

    public boolean hasNext();        // next token is a word?
    public String next();            // gets next word
    public boolean hasNextInt();     // next token is an integer?
    public int nextInt();            // gets next integer
    public boolean hasNextDouble();  // next token is a double?
    public double nextDouble();      // gets next double
}
```

(b) Excerpt from the interface `java.util.Scanner`. Each subsequent token matching a given pattern is accessed via the `next(Pattern)` method. The presence of that token in the input can be tested through the `hasNext(Pattern)` method. Convenience methods `next*()` and `hasNext*()` implement patterns for common token types.

Figure 2.1: Relevant parts of the `java.io.StreamTokenizer` and `java.util.Scanner` interfaces used in the case study.

Before	After
`java.io.StreamTokenizer s;`	`java.util.Scanner s;`
`x = (int) s.nval`	`x = s.nextInt()`
`x = s.nval`	`x = s.nextDouble()`
`x = s.sval`	`x = s.next()`
`if (s.ttype == TT_NUMBER) ...` `if (s.nextToken() == TT_NUMBER) ...`	`if (s.hasNextInt() ||` `  s.hasNextDouble()) ...`

`if (s.ttype == TT_WORD) ...`
`if (s.nextToken() == TT_WORD) ...`
	`if (s.hasNext() &&`
	`  !s.hasNextInt() &&`
	`  !s.hasNextDouble()) ...`

`s.nextToken()`	*`// no longer needed`*

Figure 2.2: Examples of transformations needed in source code to convert the uses of `java.io.StreamTokenizer` to the uses of `java.util.Scanner`.

a new, more capable class that implements an input scanner based on pattern matching (`java.util.Scanner`). Each of these classes takes a different approaches to input tokenization and their interfaces are not completely amenable to automatic translation. In many situations, however, it is possible to translate most uses of `java.io.StreamTokenizer` to the equivalent uses of the `java.util.Scanner`. Figure 2.1 summarizes the parts of both interfaces that are relevant to this case study. Figure 2.2 presents several transformations needed to transition from `java.io.StreamTokenizer` to `java.util.Scanner`.[3]

2.3.1 SED

SED, a *Stream EDitor*, transforms its input stream and produces an output stream. The traditional way to use SED for source code transformation is to employ its '`s`' command that globally replaces each instance of text matching a regular expression. The substitution string may be formed by referencing parts of the matching text with '`\1`'...'`\9`' metavariables that refer to the n^{th} group of characters in the pattern surrounded by `\(` and `\)`. The general format of the '`s`' command is `s/<pattern>/<substitution>/g` ('`g`' indicates that a substitution is to be performed globally, rather than on the first matching instance). SED scripts may be stored in text files, which facilitates reuse of the transformations, provided they are general enough to apply in a broader context.

The transformations listed in the Figure 2.2 may be implemented with the following SED script:

```
s/java.io.StreamTokenizer/java.util.Scanner/g
s/(int)\([_a-zA-Z][_a-zA-Z0-9]+\)\.nval/\1.nextInt()/g
s/\([_a-zA-Z][_a-zA-Z0-9]+\)\.nval/\1.nextDouble()/g
```

[3]We are ignoring the changes required in the setup and the instantiation of each component. Those parts are typically handled manually, because adaptation of the `java.io.StreamTokenizer` setup code requires some creative thinking to match the semantics of `java.util.Scanner`.

```
s/\([_a-zA-Z][_a-zA-Z0-9]+\)\.ttype==TT_NUMBER
 /\1.hasNextInt()||\1.hasNextDouble()/g
s/\([_a-zA-Z][_a-zA-Z0-9]+\)\.nextToken()==TT_NUMBER
 /\1.hasNextInt()||\1.hasNextDouble()/g
s/\([_a-zA-Z][_a-zA-Z0-9]+\)\.ttype==TT_WORD/\1.hasNext()/g
s/\([_a-zA-Z][_a-zA-Z0-9]+\)\.nextToken()==TT_WORD/\1.hasNext/g
s/[_a-zA-Z][_a-zA-Z0-9]+\.nextToken()//g
```

In addition to being virtually unreadable,[4] this script is overly simplistic and does not address the following situations that might arise in the source code:

- **Whitespace and comments in the input.** No provisions are made for syntactically insignificant material that might appear at the match location. For example, `s.nval` gets properly transformed, whereas `s␣.␣nval` does not. Similarly, comments appearing in the match string (and especially multiline `/* */` comments, not easily described by regular expressions) are not handled by this script.
- **Expressions in place of identifiers.** This script matches only a single identifier that represents a `java.io.StreamTokenizer` instance, not allowing arbitrary expressions in its place. This, for example, prevents `getTokenizer().nval` from being properly replaced.
- **Commutativity of the '==' operator.** Additional rules must be added to permit this script to match `TT_NUMBER==s.ttype` in addition to `s.ttype==TT_NUMBER`.

Some of these limitations may be addressed by extending this SED script to match more complicated inputs. Yet, SED's inability to perform any syntactic processing, such as matching arbitrary Java expressions,[5] remains the major limiting factor in using SED-like tools for program transformation.

2.3.2 TXL

TXL was originally conceived as a language for creating extensions for the Turing programming language (in fact, TXL stands for *Turing eXtender Language*). Over time, TXL evolved into a general-purpose language for program transformation. TXL is designed around the paradigm of rule-based structural transformations, and supports unification, implied iteration, and deep pattern matching.

The input to the TXL processor consists of the parsing and unparsing grammars, the transformation specification (TXL script), and the source text. The TXL processor transforms input text into output text according to a TXL program. Processing of input text begins with the parsing phase that constructs the structural representation (parse tree) for

[4] Regular expression have been said to resemble "line-noise" and from this example it should become obvious that there's a good reason for that observation.

[5] Matching nested expressions is outside of the expressive capabilities of regular-pattern matchers.

the input. The subsequent transformation phase rewrites that parse tree according to the transformation rules described in the TXL program. The final processing phase "unparses" the transformed tree into a textual representation. The parsing and unparsing grammars need not represent the same language; cross-language translation is achieved by combining two different grammars within a single description. Because TXL scripts are stored in text files, they can be reused across different target programs, when applicable.

As a rule-based language, TXL permits very expressive specification of transformations. Because a set of transformations is specified in conjunction with a parsing grammar, that grammar needs to express only enough structure necessary for those transformations [25]. This permits manipulation of source text that is incomplete or incorrect with respect to the full grammar for a source language. As long as enough structure is recovered, the transformation engine can apply the rules and produce the desired result.[6] The patterns that constitute the transformation rules are specified textually in a form similar to the source language. This leads to a more natural specification of the transformation rules than those permitted by the tree-based systems such as TAWK and A*.

The TXL program[7] to perform the transformations listed in Table 2.2 begins as follows:

```
include "Java.Grm"
include "JavaCommentOverrides.Grm"

function main
   replace [program]
        P [program]
   by
        P [transformTokenizerToScanner]
          [transformNextInt]
          [transformNextDouble]
          [transformNext]
          [transformNumberTest]
          [transformWordTest]
          [removeNextTokenStatement]
end function
```

The `include` command incorporates the rules for parsing and unparsing Java source code into the current TXL program. The `main` function identifies the transformations that should be applied to the entire program. The `replace` clause applies to a particular structure, in this case, the `program` non-terminal. The job of the `replace` clause is to break down the subtree designated by the non-terminal, according to the pattern specified in the replacement rule. The pattern `P [program]` indicates that it matches the entire program and that the result of the match should be assigned to the pattern variable `P`. The second part of

[6] The TXL distribution includes full language parsing grammars for many popular programming languages. These grammars, however, do not permit any inconsistencies in the input.

[7] This case study was performed using the release 10.4a of the TXL language and tools. TXL can be downloaded from `http://www.txl.ca/`

the `replace` clause (following the `by` keyword) constructs the replacement for the matching non-terminal. This replacement is created by applying several rules to the pattern variable `P` in order. In effect, each rule rewrites `P`, yielding a modified program as a result. As there are seven transformation rules, this transformation requires seven passes over the program to complete. Following is the definition of the first of these rules:

```
rule transformTokenizerToScanner
   replace [qualified_name]
        java.io.StreamTokenizer
   by
        java.util.Scanner
end rule
```

This rule performs the first transformation from Figure 2.2. The pattern and the replacement are specified as fragments of source code that are parsed according the Java language grammar. The transformation writer needs to specify which non-terminal from the grammar is represented by the pattern and by the replacement (in this case, `qualified_name`). This is a simple transformation with no variables in the pattern. Let us consider a more complicated set of rules for performing the second transformation from Figure 2.2 from `(int) s.nval` to `s.nextInt()`:

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

rule transformNextIntInComponent
   replace [repeat component]
        .nval
   by
        .nextInt()
end rule
```

This transformation is somewhat complicated by the shape of the parse tree generated by the Java grammar. This grammar represents qualified field access as an identifier terminal followed by a sequence of "components," terminals preceded by a `.` (dot) that separates them from one another. This representation yields the following parse tree for qualified field access (using brackets `[` and `]` to represent nesting):

```
ID [[. ID] [. ID] [. ID] ... [. ID]]
```

This means that prior to matching the `.nval` component of the qualified fields access, the parse tree for field access needs to be decomposed into the initial identifier that refers to

an instance of `java.io.StreamTokenizer` and the rest of the qualified name. This is the purpose of the `transformNextIntInComponent` rule.

Due to the peculiarities of the TXL Java grammar, the rules above will not transform a qualified field access in which the `java.io.StreamTokenizer` instance is represented as a parenthesized Java expression. For example, the statement `((StreamTokenizer)s).nval` cannot be matched by the pattern of the `transformNextInt` rule. This limitation of the TXL Java grammar can be addressed by an additional rule in the transformation description:

```
rule transformNextInt2
   replace [expression]
        (int) (E [expression]) C [repeat component]
   by
        (E) C [transformNextIntInComponent]
end rule
```

For the sake of brevity we will not present the remaining TXL transformation rules in this case study. In Appendix A we include the full listing of the TXL program implementing the transformations of Figure 2.2. As a source-to-source transformation tool, TXL exhibits the following limitations:

- **Dependence on the parse grammar of the programming language.** TXL uses top-down parsing with full backtracking, permitting a flexible specification of the programming language grammar. Yet, factoring of the grammar can be unnatural and can require additional coding effort on the part of the transformation writer. Relying purely on the syntactic structure can obscure the intent of the transformation, causing the developer to introduce additional rules for matching semantically equivalent constructs (once again, commutativity of the `==` operator serves as a good example).

- **Complexity of the transformation language.** TXL is a rule-based language. While the semantics of rule application are not complicated, the transformation writer needs to be constantly aware of how and in what order the rules are processed. TXL functions are applied in the postfix form, leading to an unnatural specification of the replacement structure.

- **Lack of semantic attribution.** The TXL language operates on tree structures. The mechanism for creating such structures is incorporated into the language. Yet, no support is provided for further analyses of source code, such as static semantics. While it is possible to implement these analyses in TXL, no publicly available implementation exists for the Java programming language. In fact, the TXL transformations described in this case study rely exclusively on the uniqueness of the syntactic form: none of the transformations involving access to an instance of `java.io.StreamTokenizer` ensure that the object being accessed is statically typed as `java.io.StreamTokenizer`.

- **Disregard for the documentary structure of source code.** The final stage of TXL processing involves unparsing the tree structure into text. While the comments

are preserved by TXL as part of that structure, whitespace is not. This affects the visual layout of the source code, making TXL unacceptable as tool for describing source code changes. Van De Vanter [64] emphasizes the significance of the documentary structure and reports that the commercial version of TXL, distributed by the Legasys Corporation, implements a special strategy that incrementally updates the source text to avoid disrupting that structure.

TXL is not well-suited for lightweight transformations. Despite the high-level rule language, the users of TXL are still required to reason about grammars and trees at a level that is beyond what most developers understand. Using the provided language grammars requires understanding of the structure at a finer level of detail than necessary for most transformations. Moreover, TXL is purely structure-based; semantic attribution is not incorporated into the parsing grammars and needs to be specified separately (or computed as part of the transformation). The transformations, especially those involving complicated patterns, are difficult to create. In fact, the TXL tutorial suggests a workflow, whereby a generic transformation is created by iteratively generalizing a rule that applies to one specific instance of the transformation in the source code. Debugging TXL scripts is challenging as no debugger is supplied with the TXL processor. Recently, however, Shimozawa and Cordy demonstrated some advances in this area in their Transformation Engineering Toolkit for Eclipse (TETE) [59].

2.3.3 Refactoring Tools

Using an automated refactoring tool presents an entirely different approach to implementing the source code changes. While it is not possible to automate all steps in our transformation process, a reasonably featureful refactoring tool provides a way to eliminate some of the mundane edits. For our case study, we used the Eclipse refactoring engine that offers developers a set of about twenty automated refactoring transformations. In the following sequence of (mostly) refactoring steps we present a description of each refactoring, followed by a brief explanation of how that refactoring applies to our example.

> *Encapsulate field:* Make a public field private and provide accessors (Fowler [31], p. 206).

We apply the *Encapsulate Field* refactoring to the `nval` and `sval` fields of the `java.io.StreamTokenizer` class to wrap them in accessor methods. This enables a subsequent redirection of accesses to these fields to the appropriate methods in `java.util.Scanner`. This refactoring changes code as follows:

```
...
double x = s.nval;
...
```

⇒

```
double x = s.getNval();
...
double getNval() {
    return nval;
}
```

We do not encapsulate access to `ttype`. This is fully intentional: this field is typically used for testing against one of the pre-defined type constants (`TT_WORD` or `TT_NUMBER`) and rather than hiding just the access to that field, we want to encapsulate the entire test. This is achieved in two subsequent steps:

> *Extract Method:* Turn a code fragment that can be grouped together into a method whose name explains the purpose of the method (Fowler [31], p. 110).

> *Move Method:* Move a method into the class it uses most (Fowler [31], p. 142).

The *Extract Method* refactoring is applied to an expression that is to be isolated into a separate method, in this case `s.ttype==TT_NUMBER` and `s.ttype==TT_WORD`. Most automated refactoring tools (including Eclipse) apply this refactoring to other instances of the same expression occurring elsewhere in the source code.

```
...
if (s.ttype == TT_NUMBER) ...
...
```

⇒

```
if (isNum(s)) ...
...
boolean isNum(StreamTokenizer s) {
    return s.ttype == TT_NUMBER;
}
```

Extract Method leaves the extracted method in the class where the body of the method was located prior to extraction. We apply *Move Method* to relocate the extracted method into the `java.io.StreamTokenizer` class. Following similar steps, we encapsulate testing for the `TT_WORD` token type in the `isWord()` method.

At this point, we have isolated all uses of the low-level `java.io.StreamTokenizer` API into high-level methods that implement concepts common to both `java.io.StreamTokenizer` and `java.util.Scanner`. The next step involves rewriting the implementation of these methods inside `java.io.StreamTokenizer` to use `java.util.Scanner`. This can be done as follows:

1. Modify the definition of the `java.io.StreamTokenizer` class to inherit from (extend) `java.util.Scanner`.
2. Modify `getNval()` to `return nextDouble()`.
3. Modify `getSval()` to `return next()`.
4. Modify `isNum()` to `return hasNextInt() || hasNextDouble()`.
5. Modify `isWord()` to `return hasNext() && !isNum()`.
6. Remove `nextToken()`, replacing its body with `return 0`.

Conceptually, this rewrite of the `java.io.StreamTokenizer` class reifies the transformation table of Figure 2.2. This enables application of two more refactoring transformations that complete the upgrade from `java.io.StreamTokenizer` to `java.util.Scanner`:

> *Inline Method:* Put the method's body into the body of its callers and remove the method (Fowler [31], p. 117).

We apply *Inline Method* to each of the methods that we modified in steps 2 through 6. This refactoring removes these methods from `java.io.StreamTokenizer` and inlines their implementation at every call site. At this point, the definition of the `java.io.StreamTokenizer` class contains no useful public methods and no useful fields. This permits us to apply one final refactoring to remove this (now) useless class:

> *Collapse Hierarchy:* Merge a subclass with superclass when they are not very different (Fowler [31], p. 344).

This refactoring removes `java.io.StreamTokenizer` and replaces every reference to it with `java.util.Scanner`. (The *Collapse Hierarchy* refactoring is not implemented in Eclipse. Instead, we can apply *Generalize Type* refactoring to every field, variable, and parameter of type `java.io.StreamTokenizer`.)

Using a sequence of refactoring transformations to implement interface translation is not the most intuitive application of the refactoring methodology. While refactoring is an extremely useful technique for evolving maintainable source code base, it presents a number of challenges as a tool for general purpose source code transformation. In particular, this case study exposes the following issues:

- **Breaking down a transformation into a sequence of primitive refactoring steps is awkward and non-intuitive.** Constructing the entire refactoring recipe presented in this case study can be challenging even to the experienced refactorers. The utility of some of the intermediate steps is unclear, unless the sequence of steps is considered in its entirety. Moreover, this sequence of refactoring steps depends on several factors that are not essential for this transformation. Consider the following issues:

  - In the presented sequence we assumed that we can modify the source code for `java.io.StreamTokenizer`. While in practice this is almost never the case, Eclipse allows us to create a copy of `java.io.StreamTokenizer` in the current project that overrides the library version. This enables us to complete the prescribed steps.
  - This sequence relies on being able to "re-parent" `java.io.StreamTokenizer` to inherit from `java.util.Scanner`. This is only possible because (a) `java.io.StreamTokenizer` has no declared superclass and (b) `java.util.Scanner` is

not marked as `final`, which would prevent it from serving as an inheritance parent to another class.[8]

- **Meaning-preserving refactoring paradigm is too restrictive.** While we do want the API update to preserve the behavior of the original program, it is not necessary for the intermediate steps to preserve that behavior or even to preserve syntactic and semantic correctness of the program. Furthermore, the refactoring tools take a very conservative view of what it means to preserve program's meaning. For example, when applying the *Extract Method* refactoring to `s.ttype == TT_NUMBER`, Eclipse will attempt to locate all instances of that expression and replace them with calls to the extracted method. Eclipse's refactoring engine will not consider `TT_NUMBER == s.ttype` as a candidate for replacement since in Java these expressions are not equivalent.[9] In this case, however, the two equality tests are completely equivalent and interchangeable.

- **Not all transformations can be anticipated in the refactoring tool.** The astute reader will notice that we silently ignored the issue of choosing whether to invoke `nextInt()` or `nextDouble()` when retrieving a numeric value from the token stream. This was due to the need to check the entire expression context (in this case, noticing the cast to `int`) to determine the appropriate method. One obvious solution is to perform a manual "fix-up" of the code following the transformation. In some cases (depending on the program being modified), this issue can be resolved by applying *Extract Method* refactoring to `(int) s.nval`, followed by *Move Method*, instead of *Encapsulate Field* as we did for `s.ttype`.

Notwithstanding the fact that the sequence of refactoring steps is fragile with respect to the target program (which limits its reuse), most automated refactoring tools do not permit creation of reusable transformation specifications. Recent versions of the Eclipse IDE can preserve individual refactoring transformations in an off-line form as *refactoring scripts*. These scripts can be packaged together with code that was subject to refactoring, so that other code that relies on it can be updated accordingly.

2.4 Conclusion

This chapter presents several systems available to developers for automating application of systematic changes to source code. The work presented here is closely related to the notion of interactive program transformation that we present in the subsequent chapter. We also set up the context for broader discussion of transformations in the rest of this dissertation.

[8] In fact, `java.util.Scanner` *is* marked `final`, but we can circumvent that restriction by copying it into the user's project, as we did with `java.io.StreamTokenizer`, and editing the source code to remove the keyword `final`.

[9] Because the equality operator evaluates left-hand side before the right-hand side, replacing the latter test with a call to a method extracted from the former test will change the evaluation order.

We have demonstrated that no existing approach provides an acceptable solution for performing lightweight source-code-changing transformations as part of the natural software development workflow. While general-purpose tools can describe a wider array of program transformations, the refactoring and restructuring tools are lightweight, simple, and better integrated, making them easy to use for an average programmer.

The use of existing general-purpose program transformation tools is difficult because these tools expose structural models of the target program that developers do not understand. These structural models are designed for the benefit of the tool and bear little resemblance to the programmer's intuitive understanding of their program's structure. Yet, transformation of the program structure is necessary for many source code changes. Completely hiding this model, as is done in the automated refactoring and restructuring tools, limits the capabilities of those tools.

Finding a suitable compromise that exposes just the necessary structure and does so in a way that is understandable to developers presents the challenge addressed in this dissertation.

Chapter 3

A Case for Interactive Transformations

In Chapter 1 we argued that transformation tools can be effective in assisting developers with source code changes only if they are integrated into the software development workflow. Yet, of the various transformation tools discussed in the previous chapter, only the refactoring tools have been available as part of development environments. We showed that the refactoring tools provide important, but limited transformation facilities; our goal is to permit more expressive manipulations of source code. In this chapter we introduce the notion of *interactive* source-to-source program transformation and demonstrate how interactive specification and execution of transformations can make the concepts behind general-purpose program transformation systems accessible to developers.

3.1 Interactive Transformation of Program Source Code

One approach to lightweight program transformation is to integrate an existing general-purpose transformation tool (such as TXL) into a modern interactive development environment (such as Eclipse). Much like a compiler that has become an integral component of an IDE, an integrated transformation tool would continue to operate as a standalone component. Integration will eliminate the need for developers to switch tool contexts when they need to use transformations for source code editing. Unfortunately, this approach does not address an important flaw with the transformation process: describing all but the simplest of transformations with general-purpose tools will still be too difficult and too time-consuming.

One of the challenges in designing a language for describing structure-based transformations lies in balancing the ability of the users to reason about program structures with the ability of the transformation engine to interpret transformation descriptions. Research in the psychology of programming (for example, see Detienne [26] for a comprehensive discussion) has repeatedly shown that the mental structures of source code used by programmers bear little resemblance to those used by language-based programming tools.

A solution to this problem is to create a transformation system that scaffolds both the construction of a transformation and the developers' understanding of its effects on source code. Thus, the transformation process becomes inherently interactive. The transformation tool should "hold the developer's hand" from the moment a transformation is conceived, to the moment he or she returns to coding. Design of such a system must take into account the limitations of the human cognitive apparatus.

To design iXj we adapted a user interface design technique called *task-centered design* [47]. Task-centered design focuses on the tasks that the users are expected to perform with the system. The task-centered approach prescribes a sequence of steps necessary to complete the design and the implementation. The rest of this chapter is loosely organized along these steps. Each section begins with a brief summary adapted from Chapter 1 of Lewis and Rieman [47].

3.2 Task and User Analysis

The purpose of task and user analysis is to understand the target audience for a software system and the principal tasks that this audience needs to perform. In addition to addressing the users' needs, a successful system must integrate smoothly with the user's workflow and must communicate with the user via understandable terminology. The system should be optimized for the tasks that the users are trying to perform. It should request only the information that the users are likely to possess, make it easy to correct mistakes, and offer assistance when necessary.

We assessed the developers' needs by analyzing verbal descriptions of changes in several software systems and by performing a series of informal user studies. We studied natural language descriptions of source code changes from three sources. Our first source was the comprehensive development log published by Donald Knuth in *The Errors of TEX* [43]. Figure 3.1a presents several representative log entries. Our second set of examples (Figure 3.1b) came from a `ChangeLog` file included in the distribution of XEmacs [74], a popular text editor. As a third example, we considered a transcript of a dialog between programmers engaged in pair-programming (Figure 3.1c). This self-recorded transcript was presented in Martin and Koss [50].

Finally, we conducted an informal user experiment to understand how developers 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 verbal description of a change. In particular, we were interested in how developers reference code fragments to be transformed, how they describe the output, and what programming style they use.

Our analysis led to several observations:

- The developers use high-level linguistic notions, such as macros, variables, methods, functions, and loops. This means that we can safely expose these notions in the transformation language and expect developers to understand what they mean.

"Rename a few external variables to make their first six letters unique."

"Put `p:= link(p)` into the loop of `show_token_list`, so that it doesn't loop forever."

"Assign a floating-point constant `ignore_depth` to `prev_depth`, instead of assigning the integer constant `flag`."

(a) Excerpts from the TEXdevelopment log [43].

"Change `BI_*` macros to `BYTE_*` for increased clarity; similarly for `bi_*` local vars."

"Change `VOID_TO_LISP` to be a one-argument function. Eliminate no-longer-needed `CVOID_TO_LISP`."

"Rename macros with `XSTRING_*` to `string_*` except for those that reference actual fields in the `Lisp_String` object, following conventions used elsewhere."

(b) Excerpts from XEmacs `ChangeLog` [73]

RCM: (Thinking out loud) "This doesn't compile because we haven't written the `Throw` class."

RSK: "Talk to me, Bob. The test is passing an integer, and the method expects a `Throw` object. You can't have it both ways. Before we go down the `Throw` path again, can you describe its behavior?"

RCM: "Wow! I didn't even notice that I had written `f.add(5)`. I should have written `f.add(new Throw(5))`, but that's ugly as hell. What I really want to write is `f.add(5)`."

RSK: "Ugly or not, let's leave aesthetics out of it for the time being. Can you describe any behavior of a `Throw` object—binary response, Bob?"

RCM: "101101011010100101. I don't know if there is any behavior in `Throw`; I'm beginning to think a `Throw` is just an `int`. However, we don't need to consider that yet, since we can write `Frame.add` to take an `int`."

(c) A transcript of a dialog between two programmers engaged in a pair-programming session [50]

Figure 3.1: Examples of developers describing changes to source code. The examples were used for task and user analysis prior to designing iXj.

- The developers use patterns to describe classes of similar changes, for example `BI_*` and `bi*` in the XEmacs `ChangeLog`. This means that a pattern-based transformation language will naturally fit with their perception of a systematic change.
- To describe a location in the source code, the developers use both the language concepts ("in class `Employee`, method `getName`") and the code fragments in Java ("replace `get(x)` with ..."), switching between these two levels as they feel is appropriate. Consequently, the transformation language should support both of these mechanisms for talking about code.
- Semantic information is implicit in the transformation description. For example, when developers say "rename 'x' to 'y'," the intent, usually, is to rename only those instances of 'x' that are in the same scope. This means that the scoping information should be available as part of the program structure.
- The terminology used by developers was specific to the Java programming language. As a result of this observation, the iXj transformation language is tightly coupled with Java. (We expect, however, that our design methodology can be applied to other programming languages.)

3.3 Representative Tasks

The second step in task-centered design requires the designer to identify several representative tasks that the users will perform with the system. In contrast to the first step, where the designer develops an understanding of the users and the tasks in the abstract, at this point it is essential to incorporate some of the actual tasks proposed by the users. These tasks should cover most of the desired functionality for the system. Typically, a designer augments the initial set of user-identified tasks with additional tasks to achieve the desired coverage of system functionality.

In order to identify representative transformation tasks we again turned our attention to project development logs. In particular, we looked at the `ChangeLog` files for several open-source projects as well as our own development logs for the Harmonia program analysis framework [8]. For consistency of the presentation, the examples below have been translated into Java. Each of these tasks requires different levels of expressiveness in the transformation specification; we outline these differences in the description of each task.

3.3.1 Include the name of the enclosing method in output

When debugging, a common strategy is to insert statements that output the current state of the algorithm and the data structures to the console (so-called, "printf" debugging). In doing so, it is often useful to include the name of the currently executing method as part of that output. We have encountered situations when this was initially overlooked by developers, requiring them to modify the existing tracing code to display the name of the

corresponding enclosing method. For example, the developers might change the code as follows:

```
public void main() {
    ...
    out.println("compute");
    ...
}
```

⇒

```
public void main() {
    ...
    out.println("main(): compute");
    ...
}
```

This transformation operates on the lexical (sub-token) structure of the program. In order to enable this transformation, the tool must support operations that can split and merge existing tokens. In this case, such a token is the string literal in the `print`-statement.

3.3.2 Introduce a symbolic constant

The use of numeric and boolean compile-time values without first assigning them to a symbolic constant is considered a bad software development practice. Not only does it obscure the meaning of a value (compare `42` with `ConstAnswerToLifeUniverseAndEverything`[1]) but also such a coding style removes a level of abstraction that makes it easy to modify that value (in case the answer to life, universe, and everything becomes `54`). Often, this recommendation is ignored by developers. Fixing the code to introduce symbolic constants requires transformation of the following form:

```
...
x = computeTheAnswer(true);
...
y = computeTheAnswer(false);
...
```

⇒

```
...
x = computeTheAnswer(PRECISE);
...
y = computeTheAnswer(APPROXIMATE);
...
```

This is a syntactic transformation that modifies program structure. It is implemented by locating boolean literal expressions that are used as arguments to the `computeTheAnswer()` method and replacing them with references to the corresponding constants.

3.3.3 Replace component implementation

One of the transformations that arise when updating a software system to use the `java.util.Scanner` class in place of the `java.io.StreamTokenizer` class involves changing all source code that reads from the `nval` field in the `java.io.StreamTokenizer` object to call the `nextInt()` method on the `java.util.Scanner` object. (See Section 2.3 for a complete description of this task.) This transformation updates source code as follows:

```
...
int x = (int) s.nval;
...
```

⇒

```
...
int x = s.nextInt();
...
```

[1] Douglas Adams. The Hitchhiker's Guide to the Galaxy.

This transformation must use static type information to ensure that the static type of `s` is `java.io.StreamTokenizer`. Otherwise, it may inadvertently change access to the `nval` field of another class.

3.4 Design Influences from Existing Systems

The purpose of this design step is to check if some of the concepts and paradigms, familiar to the users from their use of other software systems, can be incorporated into the design of the new system. The advantage of this approach (over creating a completely original design) lies in enabling the users to transfer their pre-existing understanding of the interface concepts to the new system. For example, if a user expects to right-click on a misspelled word in a word processor to choose from a list of suggested corrections, that user should be comfortable right-clicking on a mistyped variable name in a source code editor to choose a correct spelling.

The design of iXj incorporates several influences from existing systems. In designing the transformation description language we adapted some common features from other pattern languages. For instance, a wildcard is denoted with a * symbol, similarly to the *glob*-style file name patterns. The user interface and the user-interaction model of the transformation editor also borrow elements from existing tools. For example, the transformation editor includes a modeless transformation assistant that is implemented as a side pane in the editor window. The assistant enables various structural manipulations of the transformation description and explains the current state of the pattern. The contents of that pane changes as the user moves around the transformation description, providing context-sensitive information on what can be done next. This idea was borrowed from the user interface of Microsoft Word, where a similar context-sensitive pane is used to organize paragraph styles, perform searches, and browse clip art.

3.5 Rough Design

Committing a design to a written form constitutes an essential step in task-centered design. The mere act of writing things down causes the designer to consider the multitude of issues that are not usually anticipated in the initial discussions. This step should not involve any coding or prototyping, as doing so will unnecessarily constrain the designer and require commitment to certain decisions too early in the process.

In this stage of the iXj development we made several important design decisions. First, we decided that we would use a rule-based transformation description language, consisting of patterns and actions. This decision was a direct result of our task and user analysis. Second, we realized that in order to create powerful and expressive transformations, the developers will need to be exposed to some program structure. Our goal was to make that structure as understandable as possible. To this end, we designed a new data model for representing source code in a structured way that is understandable to developers. (This model is covered in more detail in Chapter 4.) Third, it became obvious that developers

need substantial support in constructing transformations, both because they are exposed to the structural information that they do not normally perceive, and because they are working with a new language. Since developers are creating transformations on existing source code, we decided to use a "by-example" approach to developing a transformation.

Programming by example (also known as "programming by demonstration") is an old and recurring theme in computer science (see Lieberman [48] for a summary). We adapted this mode of interaction to transformation development. In this mode of interaction developers starts by selecting a single source code fragment that they want to change. The transformation editor provides assistance by creating an identity transformation that changes nothing. The developers use that transformation as a starting point for adding wildcards, 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.

3.6 Design Analysis

The purpose of design analysis is to uncover shortcomings and limitations of the design prior to its being built. There are several well-known techniques for design analysis, such as *GOMS* and *Cognitive Walkthrough*. GOMS [41] (Goals, Operators, Methods, and Selection rules) is a family of techniques based on counting keystrokes and mental operations. Cognitive Walkthrough [56] is a methodology that locates interactions that cause the users to make mistakes or form misconceptions about the system.

Our design analysis is based on the framework of *Cognitive Dimensions of Notations* [33] (CDs) that both offers the techniques for interface evaluation and provides guidance on how the design can be improved. Traditionally, the CDs framework has been used as part of later-stage usability evaluations. In our work, we have put the CDs framework to another use, employing it in the early design stages to gain additional insight into the problem. This section outlines the CDs framework and discusses its use in design analysis. Chapter 6 presents our use of the CDs framework as part of a usability evaluation of iXj.

3.6.1 Cognitive Dimensions of Notations

The CDs framework is applied to *information artifacts*. Green and Blackwell define information artifacts as the "tools we use to store, manipulate, and display information" [5]. The CDs framework divides information artifacts into two broad classes: *interactive*, such as word processors, graphics packages, radios, telephones, and software environments, and *non-interactive*, such as tables, graphs, music notation, and programming languages. The CDs framework was developed as an informal evaluation method to complement the GOMS and the Cognitive Walkthrough techniques. These techniques can be used in conjunction with CDs to provide additional metrics such as the time to completion of a task (GOMS) or the amount of knowledge that the user possesses (Cognitive Walkthrough).

The Cognitive Dimensions framework provides fourteen usability dimensions of information systems, such as *visibility*, *consistency*, and *error-proneness*. These dimensions are

intended to serve as discussion tools, rather than to provide deeply analytic and quantifiable measures. The dimensions are not independent. For instance, an improvement to the consistency of an interface may lead to an increase in error-proneness. Thus, a design process guided by the CDs framework necessarily entails a series of tradeoffs. The cognitive dimensions provide a common vocabulary for the users and the designers of the system. Together, the dimensions determine a cognitive profile of the system. This profile does not represent the quality of the system; different activities and different systems require different profiles.

The following summary of the twelve (from the total of fourteen) cognitive dimensions that are relevant to our work is adapted from the cognitive dimensions tutorial created by Blackwell and Green [5]. Where appropriate and necessary, we indicate how we want our design to fare along a particular cognitive dimension.

Visibility and Juxtaposability. *Visibility* refers to the ability to view, scan, or skim components of a notation. *Juxtaposability* indicates the ability to place components next to one another for easy examination. Developing transformations with iXj involves operating on complex source code structures. This makes high visibility and juxtaposability essential to our design: the description of a transformation must be easily interpretable by the user.

Viscosity. A *viscous* system is one that is hard to modify. Blackwell and Green distinguish two types of viscosity: *repetition viscosity*, when a "single goal-related operation on the information structure requires an undue number of individual actions" and *knock-on viscosity*, when "one change 'in the head' entails further actions to restore consistency." Viscosity of a system can be quantified as the cost of making small interrelated changes (resistance to change). We consider low viscosity to be one of the most important requirements in our design. By-example construction of transformations necessitates many small incremental changes. These changes must be easy for the user to make.

Diffuseness. The verbosity or terseness of a notation is referred to as the *diffuseness* of that notation. To use Blackwell and Green's example, COBOL is a verbose language (consider "`MULTIPLY A BY B GIVING C`"), while Forth is terse ("."—a period—prints a newline). Diffuseness of the transformation notation must be well-controlled. Low diffuseness can reduce visibility (makes it harder to quickly read a transformation). High diffuseness, however, can also reduce visibility by making a transformation description too large.

Role Expressiveness. When the purpose of a component in a system or a symbol in a notation is easily inferred from its representation, the system (notation) has high role expressiveness. High role expressiveness is important for understandability.

Closeness of Mapping. A system exhibits *closeness of mapping* when the representation provided by the system appears close to the domain of the user's knowledge. For instance, a genealogy system provides close mapping when it displays genealogical data to the users as a familiar tree-shaped structure. Closeness of mapping is important in our design because we

must expose some of the program structure that the users typically perceive only intuitively. The closer their intrinsic representation corresponds to that exposed by the tool, the easier it is for the user to understand a description of the transformation.

Progressive Evaluation. The ability to check one's work in progress prior to completing a task is referred to as *progressive evaluation*. A canonical example of a system permitting progressive evaluation is a spreadsheet: the values in cells are recomputed each time the spreadsheet is modified. iXj must provide feedback to the users as the transformation is developed. When making changes to a transformation, the developers must see how that transformation affects source code. The immediate feedback can make the execution of transformations transparent to the developers and can assist in their learning the transformation language.

Error-proneness. The degree to which a system invites the users to make mistakes is called *error-proneness*. For example, a programming language not requiring variables to be declared and defined makes it impossible to detect when a variable name is mistyped.

Consistency. In order for a system to be *consistent*, similar semantics must be expressed using similar syntactic forms. An example of an inconsistency in a notation is the lack of an "infinite-loop" construct in many programming languages, giving rise to multiple workarounds, such as `for (;;)`, `while (true)`, and `do...while (true)`.

Hard Mental Operations. When a system puts high demand on the user's cognitive resources it fares badly on the dimension of *hard mental operations*. Having to remember how to use an API is a typical hard mental operation faced by software developers.

Hidden Dependencies. A system possesses *hidden dependencies* when individual components of the system depend on one another, but that dependency is not visible to the user. For instance, conventional programming languages exhibit many hidden dependencies, such as the *def-use* dependency between the assignment to a variable in one statement and its use in a different statement.

Premature Commitment. Systems requiring *premature commitment* constrain the order in which certain things can be done. An example offered by Blackwell and Green involves writing a note on a piece of paper: if the writer chooses too large a font, the note will not fit on that piece of paper. Yet, the writer "commits" to a font prior to discovering the lack of space.

Provisionality. When it is possible to make temporary notes or marks in the notation, the system fares well on the *provisionality* dimension. Provisionality reduces premature commitment and allows users to explore various directions when they are not sure what to do next. Because our users work with a new tool and a new language for describing

transformations, they often do not know how to proceed. The ability to experiment and to retract, if necessary, is essential to achieve user acceptance.

3.7 Design Mockup, Evaluation, and Iteration

Creating a mockup (or even a preliminary prototype) permits further evaluation of the design and forms a basis for discussion of the design with the users. A mockup can be as unsophisticated as a drawing on a piece of paper, or could consist of a prototype interface implementation with most of the underlying functionality (and even some non-essential features of the interface) left unimplemented. As subsequent mockups are built, they begin to resemble a complete prototype.

The evaluation of the mockup represents an opportunity to solicit feedback and incorporate it into the design. The design should be tested with people whose background and expectations approximate those of the system's real users. An evaluation usually exposes problems with the design. Following the evaluation, a designer modifies the design to incorporate lessons learned from the evaluation. It is not atypical for the entire design to be discarded and reworked from scratch. Often, more than a single iteration is required. Each subsequent iteration improves the designer's understanding of the issues; each subsequent mockup becomes more sophisticated and approaches a mostly functional prototype. Yet, since the features of the design are likely to be interdependent, it is not uncommon that one aspect of the design may only be improved to the detriment of another. It is the designer's job to determine when the specific usability objectives are met and the iteration may stop.

This section presents several mockups of the iXj design. Some elements of the initial mockups survived the iterations and appear in the final design of iXj. Others were found to be inappropriate or limited and were discarded. The final design of the iXj transformation language and the user-interaction model is presented in Chapters 4 and 5.

Most of our mockups focused on a sample task that involves replacing a use of the `java.util.Vector` class with the `java.util.ArrayList` class. `java.util.Vector` and `java.util.ArrayList` are the Java library classes that implement a dynamically expandable array of objects. Unlike `java.util.ArrayList`, `java.util.Vector` is a `synchronized` class and so accesses to its elements incur a performance cost associated with acquiring a lock. If the users want to avoid this overhead, they must use `java.util.ArrayList` instead of `java.util.Vector`. In this task we implemented a transformation that replaces calls to `L.removeElement(x)` on `java.util.Vector` with the equivalent call sequence to `L.remove(L.indexOf(x))` on `java.util.ArrayList`.

3.7.1 First Mockup

Our first mockup prototyped the interaction model between the developer and the transformation tool. This mockup was presented at the OOPSLA 2003 Doctoral Symposium [9] and to the Harmonia research group at University of California, Berkeley. This mockup was implemented as a slide show using Microsoft PowerPoint.

Figure 3.2 presents several of the key slides from the mockup. In this mockup we experimented with an idea of "by-example" construction of a transformation. At that point, we had not yet formally defined the language for describing patterns and actions. Our first design of the transformation pattern language 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.

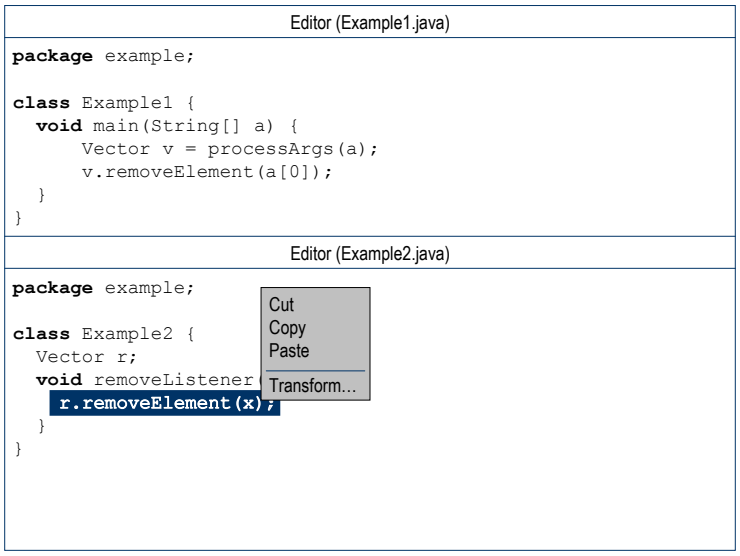
This mockup was well-received by the audiences. It became evident that the "by-example" approach to constructing a transformation was easy to follow. The audience members also found that the ability to interactively manipulate transformation descriptions simplified learning of the transformation language.

As the next step in our design, we formalized the transformation language by creating more complete specifications for the representative tasks described in Section 3.3. This activity uncovered several problems with the SQL-inspired approach. Consider the following complete description of the transformation for converting `L.removeElement(x)` on `java.util.Vector` to `L.remove(L.indexOf(x))` on `java.util.ArrayList`:

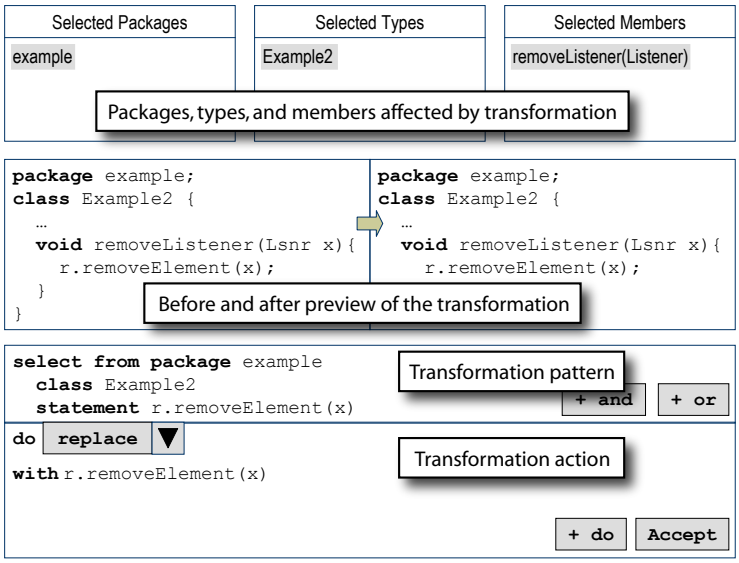
```
collection removeCalls =
select from project MyProject
  package *
  class *
  method *
  code <seq: java.util.Vector expr>.removeElement(<i: expr>)

action convertRemoveCalls foreach result in removeCalls =
replace with <seq>.remove(<seq>.indexOf(<i>))
```

This description introduces the notion of a `collection` that represents the set of code fragments matching a pattern. Collections are named and may be referenced as part of the `action`. Wildcards are specified between the angle brackets (`<` and `>`) and are also explicitly named. In the above example, `seq` and `i` are the names that refer to the two expression wildcards, denoted by `expr`. We incorporated type constraints into the wildcard notation: the `seq` expression above matches only when the (statically computed) type of the Java expression is `java.util.Vector`.

We evaluated our transformation description language using the Cognitive Dimensions framework. First, we observed that the transformation language exhibits high diffuseness. It takes eight lines of code to describe a simple transformation. Second, this 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.

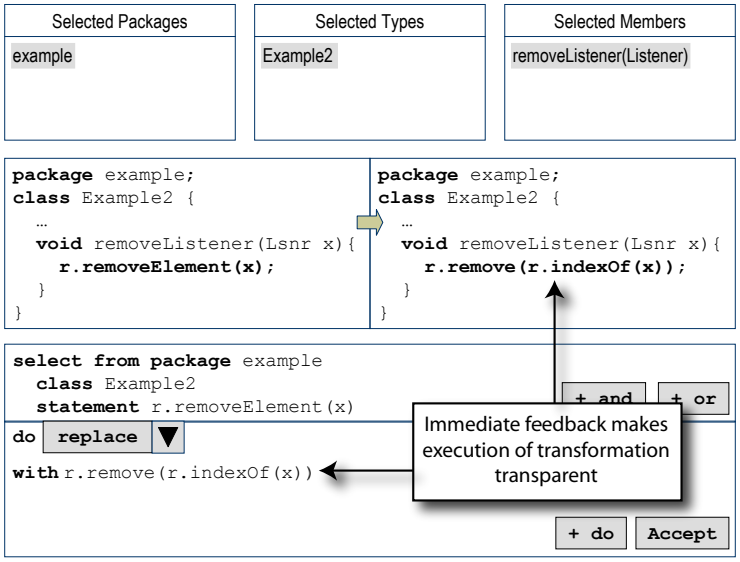


(a) The developer starts with sample code that they want to transform by selecting a code fragment and invoking "Transform..." from a context menu

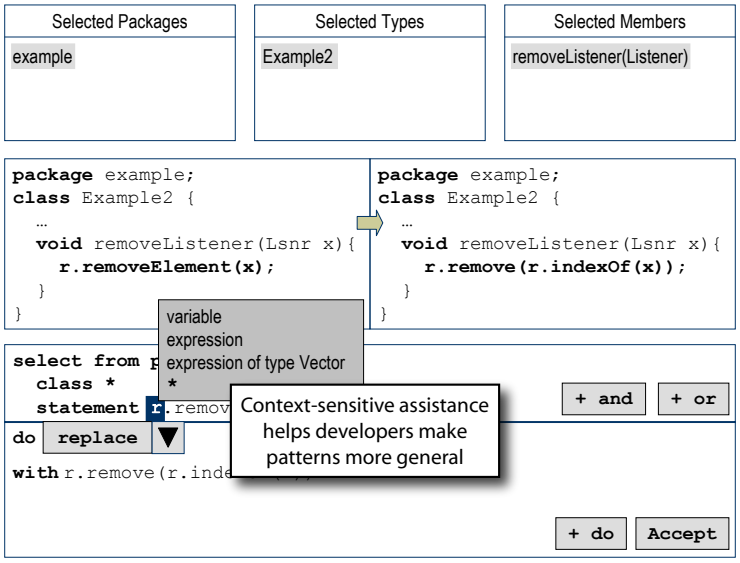


(b) Starting a transformation opens a transformation editor that shows the selected source code, a preview of the transformed source code, a transformation pattern, and an automatically-generated transforming action that implements the identity (no-op) transformation.

Figure 3.2: First mockup of the transformation tool (continues on next page).

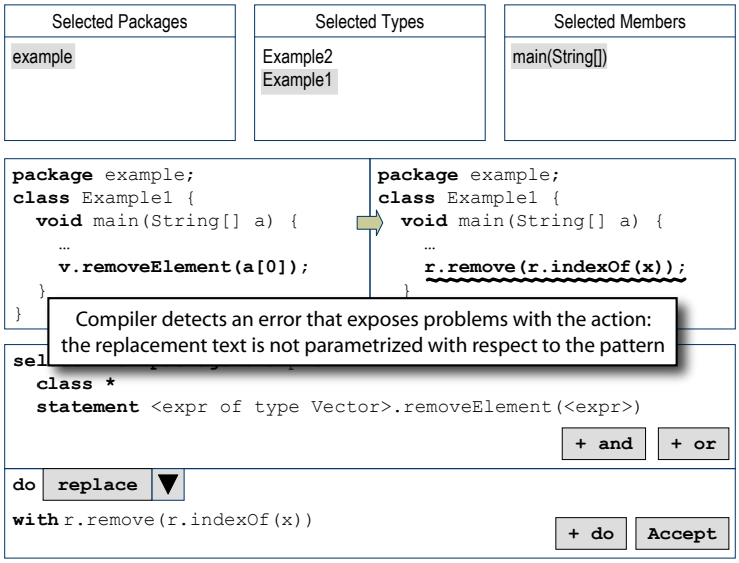


(c) The developer begins by editing the action to indicate how the transformation tool should transform the text matched by a pattern.

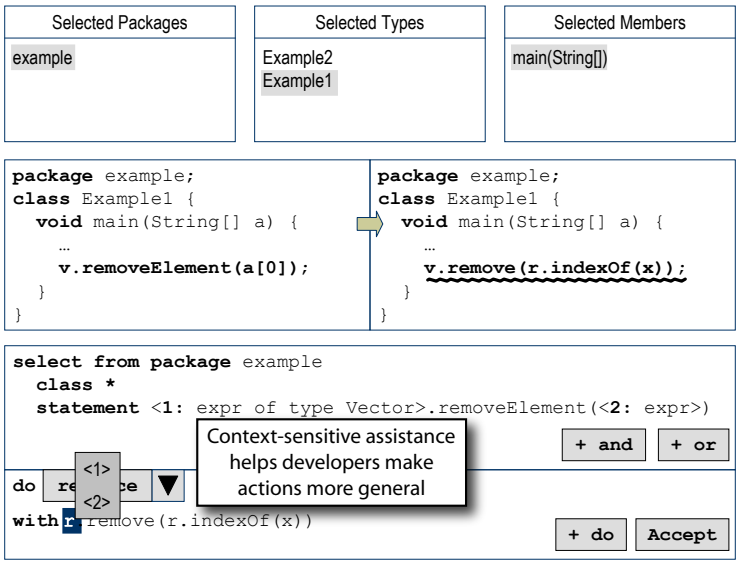


(d) The developer generalizes a transformation by adding wildcards. The transformation editor scaffolds wildcarding by offering multiple options through a context menu. The options present several successively more general possibilities.

Figure 3.2: First mockup of the transformation tool (continues on next page).

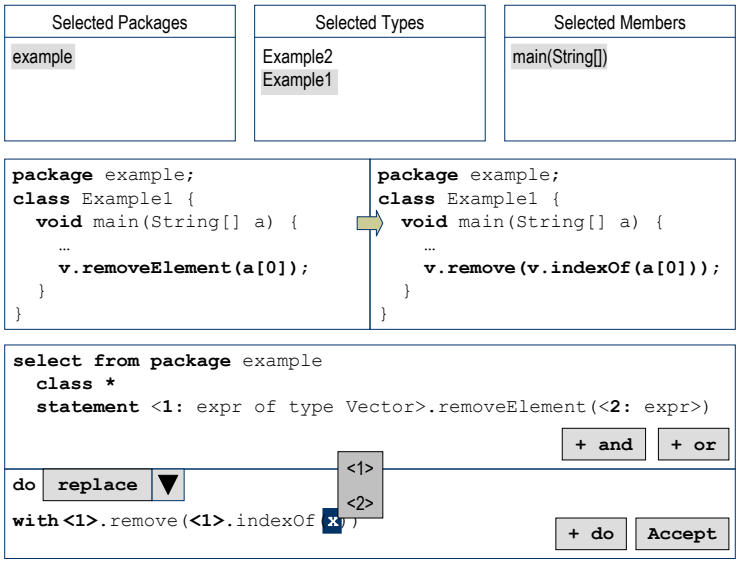


(e) The developer examines a second match of transformation pattern that appears when the wildcards are introduced. Examining this match uncovers a problem—the modification is incorrect in the context of the second match because the replacement text is not yet parametrized with respect to the pattern.

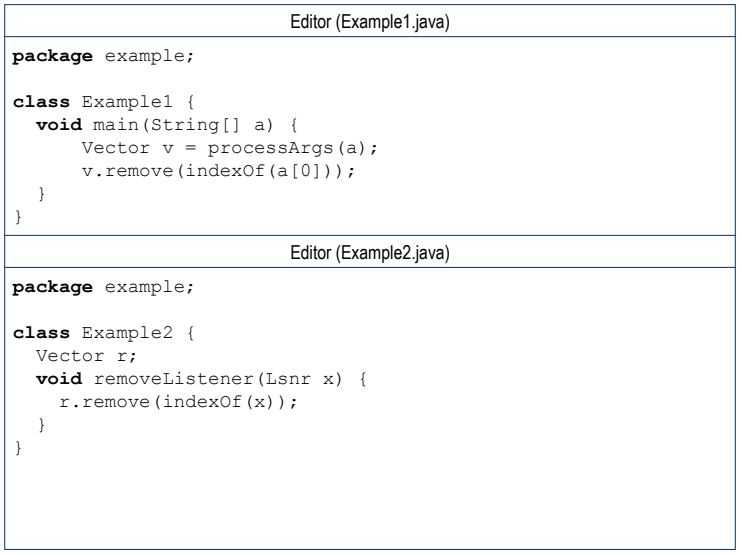


(f) The developer makes the action more general by introducing references to the pattern variables

Figure 3.2: First mockup of the transformation tool (continues on next page).



(g) The developer generalizes the rest of the action.



(h) The developer completes the transformation and applies it to the entire project.

Figure 3.2: First mockup of the transformation tool.

3.7.2 Second Mockup

In the second mockup we attempted to resolve the problems with our first design by implementing a user-interaction model that provides better scaffolding than our first prototype. We did not change the transformation description language. We evaluated this mockup by presenting it to a group of researchers at the University of California, Berkeley. This mockup was implemented as a slide-show using Microsoft Word.

Figure 3.3 shows three representative slides from the second mockup. We designed this mockup around a text-based transformation editor that permits free-form editing of the transformation description. A context-sensitive transformation assistant appears as a separate pane to the right of the editor. The transformation 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 transformation assistant; changes to one, update another.

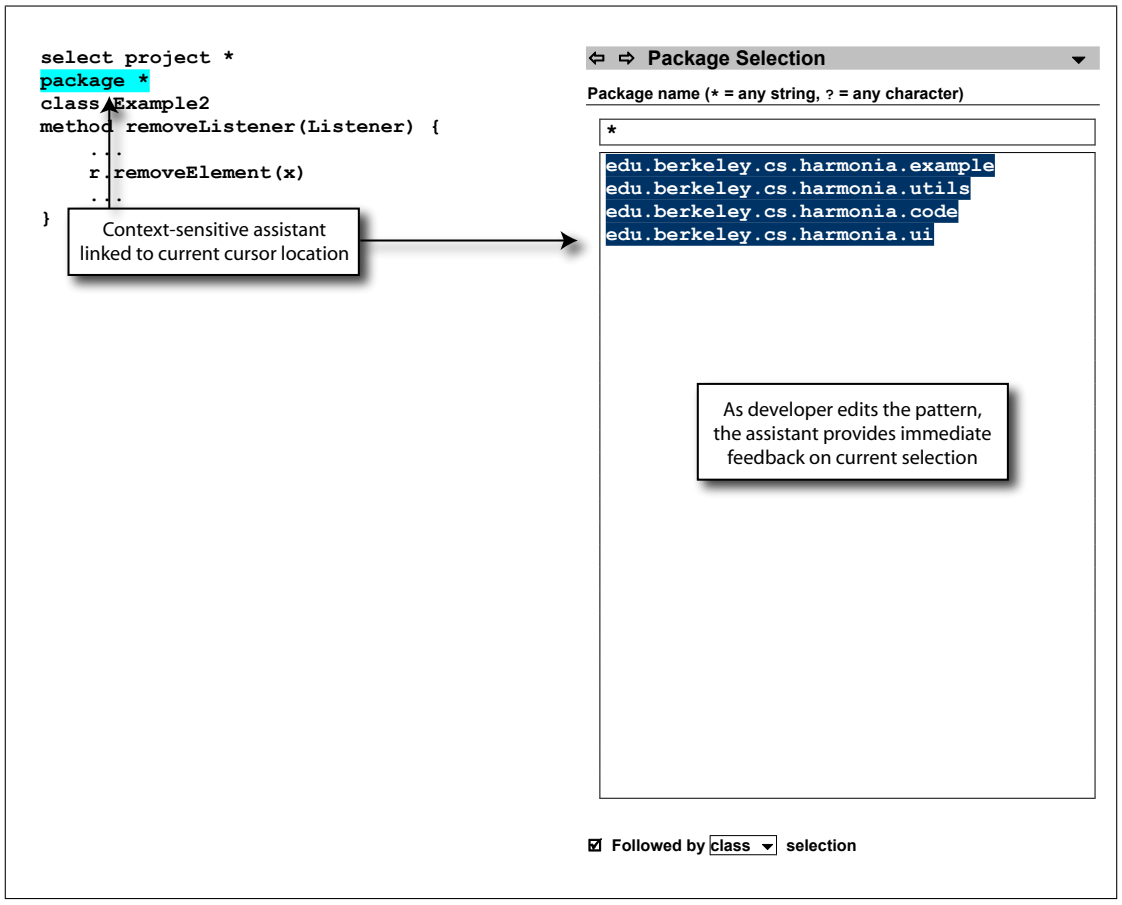
This mockup received mixed feedback. Our audience liked the transformation assistant and appreciated how it helps the user to learn the transformation language. But the transformation language still presented a problem. It was not clear how the developers would know the sequence of the steps necessary to take them from a generated identity transformation to the transformation that they want to implement. We also uncovered a technical problem: free-form editing of the transformation description can lead to its being inconsistent, erroneous, or malformed. This prevents the necessary analyses required to offer context-sensitive assistance.

3.7.3 Third Mockup

The third mockup focused on the design of the transformation pattern language. 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. The following is one of the early examples of this design:

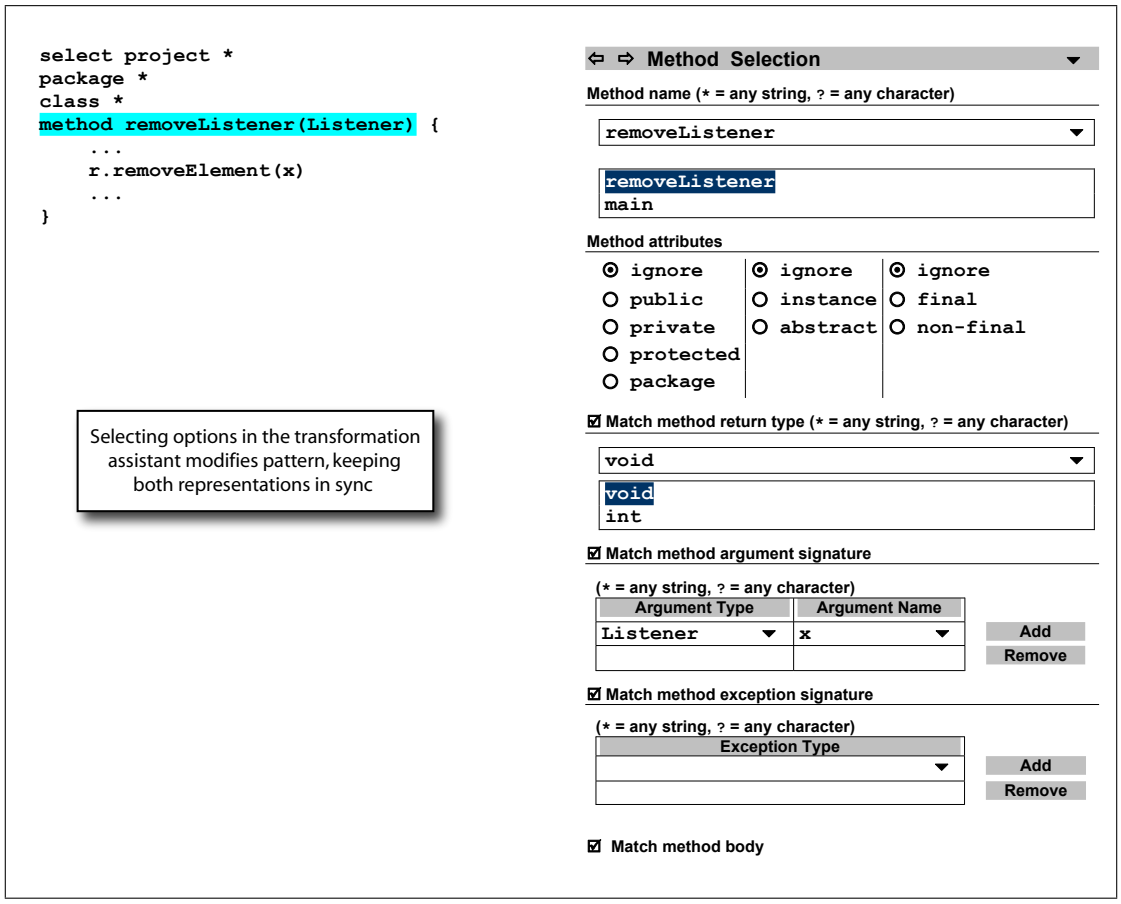
```
package edu.[[*cs]].berkeley.harmonia.examples;

class HelloWorld extends HelloUniverse {
    public static void main(String[] args) {
        Vector v = new Vector([<Expression>]);
        [<Statement>*]
        for (int i = 0; i < 10; i++) {
            [~ v.add("Hello World"); ~]
        }
    }
}
```

In this example, the lexical patterns are enclosed between double square brackets (`[[` and `]]`) and are represented as Unix *glob* patterns. For example, `[[*cs]]` matches all package



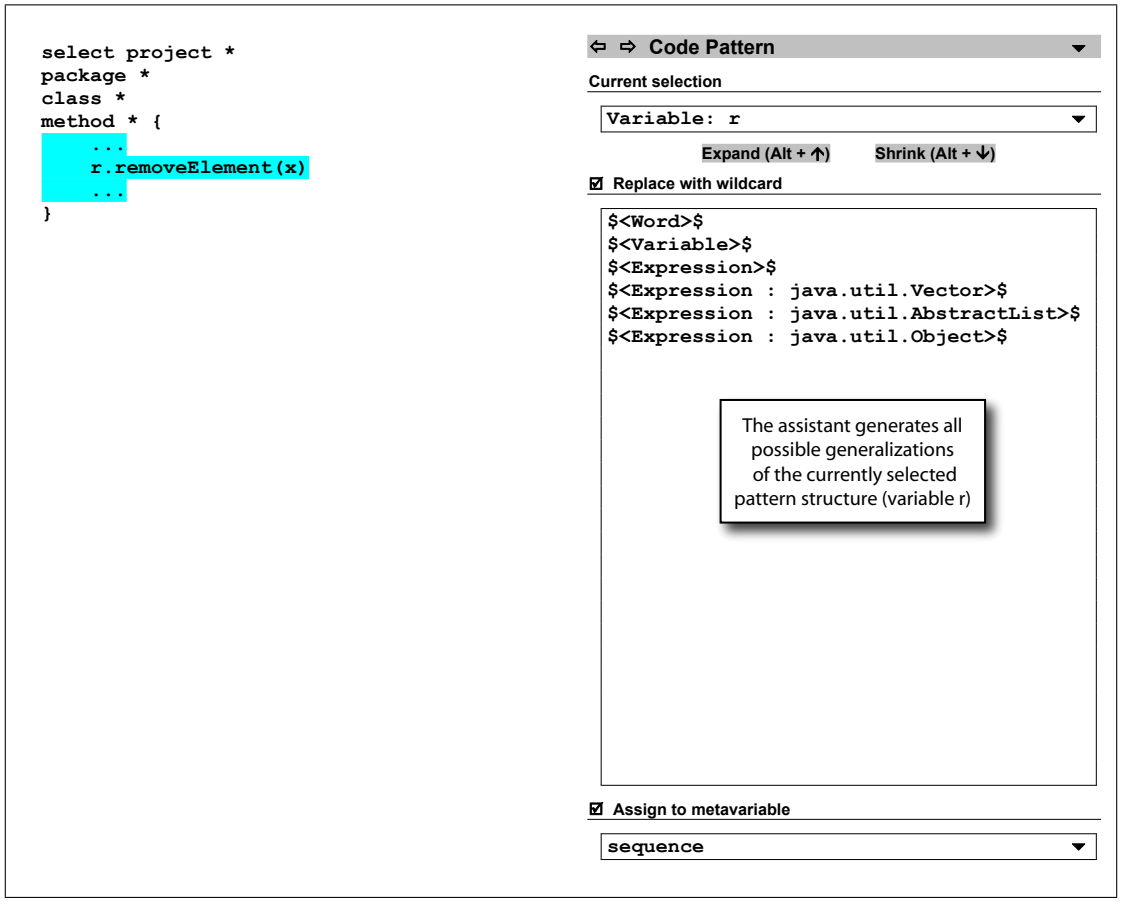
(a) Context-sensitive assistance in the transformation editor. As the developer moves the cursor around the transformation pattern, a context-sensitive transformation assistant (right) shows available operations. The developer can modify the package pattern directly, or through an input field in the assistant's panel. The list of packages shows all known packages in the system; those matching the pattern are highlighted.

Figure 3.3: Second mockup of the transformation tool (continues on next page).



(b) When the developer proceeds to modify the pattern for the method pattern, the transformation assistant displays all possible options for changing that pattern. As the developer selects different options, the transformation pattern is updated to reflect the currently selected set. This mechanism assists the developer in learning the transformation language.

Figure 3.3: Second mockup of the transformation tool (continues on next page).



(c) When the developer works with code, the transformation assistant displays different ways in which a selected code fragment can be replaced by a wildcard.

Figure 3.3: Second mockup of the transformation tool.

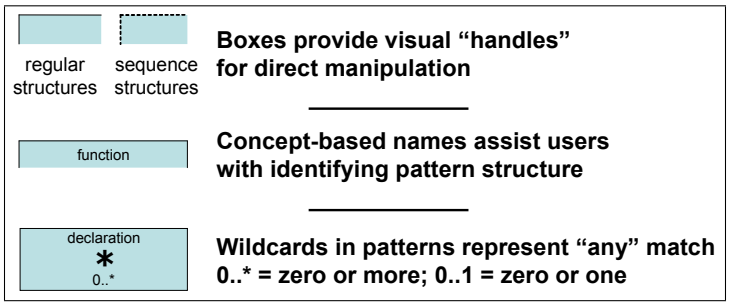


Figure 3.4: This figure shows how the boxes that surround pattern structure annotate the pattern without disturbing its source-code-like structure.

names ending with "cs", such as `cs` and `eecs`. Syntactic (structural) patterns are bracketed by `[<` and `>]`, as in `[<Expression>]`. Patterns with `*` indicate iteration. For example, `[<Statement>*]` matches a sequence of Java statements.

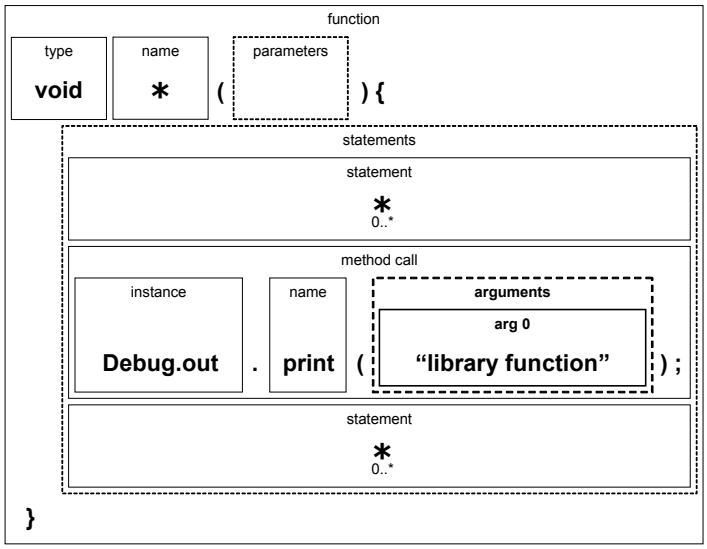
It quickly became clear that the addition of syntactic escapes makes the transformation pattern difficult to read, defeating the reason for basing the pattern language on the Java syntax. This led us to abandon a text-based notation altogether. Instead, we decided to augment Java source code with graphical boxes that demarcate structure. These boxes enable us to add annotations that are visually independent of the source code text, thereby substantially improving readability of the transformation patterns.

Figure 3.4 summarizes some of the visual elements used in this format; Figure 3.5 shows two example patterns. These examples were presented at the OOPSLA 2004 Poster Session [11]. This format was also evaluated with the members of the Harmonia research group. The key idea behind the graphical notation is to arrange information visually in such a way that the structure of the original source code fragment is undisturbed. The information about the pattern, such as names of the pattern elements and multiplicity of the wildcards, is separated from the program text that the pattern is intended to match. This design improves the visibility of the pattern notation and improves closeness of mapping. The developer no longer needs to name pattern elements explicitly—each pattern box is titled with a concept name dictated by its syntactic context. This eliminates hidden dependency between names and reduces the viscosity. The diffuseness of the format is high due to the need for structural boxes. We solved this problem by designing a user-interaction model that hides most of the structure when the developers do not require it. We based the iXj transformation language on this format.

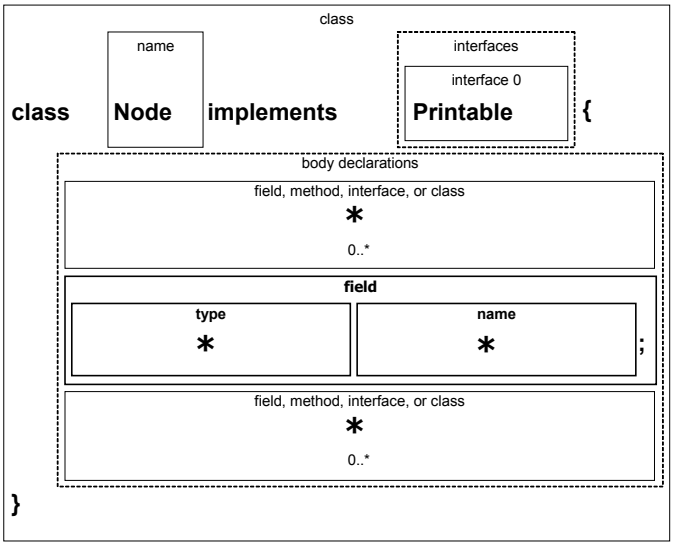
We discuss the iXj transformation language in more detail in Chapter 4. We present a user-interaction model for creating and manipulating transformations in Chapter 5.

3.8 Implementing, Tracking, and Improving the Design

The last stages of the task-centered design consist of implementing, tracking and improving the design. Because some aspects of the design only become finalized in the implementa-



(a) This pattern matches any function that contains calls to debugging `print`-statements. This pattern can be used to transform the argument to the `print()` method to include the name of the enclosing function. The name can be accessed in the transformation action by referring to the contents of the pattern box titled with 'name'.



(b) This pattern matches all fields in a class that implements a `Printable` interface. This pattern can be used to automatically generate the `print()` method specified in the interface to print all fields of the class.

Figure 3.5: Third mockup of the transformation description language.

tion phase, the implementation strategy needs to anticipate both minor and major design changes. We implemented an iXj-based transformation tool as a plug-in for Eclipse. This allowed us to rely on many existing features of the Eclipse platform, limiting the implementation effort to the core technologies required for transformation of program structure. We present our implementation in detail in Chapter 5.

Tracking and improving the design requires an evaluation of the implementation with users. We evaluated iXj with several professional Java developers. Chapter 6 presents the results of the evaluation. The evaluation revealed several areas where the design can be improved. We discuss some of these improvements in Chapter 7. These improvements, however, are beyond the scope of this dissertation.

3.9 Discussion: Task-centered Design for Development Tools

This chapter presents an atypical use of task-centered design. Traditionally, task-centered design has not been used for creating software development tools. One possible exception is end-user programming tools, such as spreadsheet applications. These tools strive to make programming accessible to non-programmers and are often designed by HCI researchers who are well-versed in human-centered design techniques.

We were able to apply task-centered design to development of a program transformation tool. We adapted the traditional task-centered design workflow to our purposes and were able to create a transformation description language and a transformation environment that was well-received by professional developers. We benefited greatly from the iterative evaluation prescribed by task-centered design. While many of our initial design decisions were correct, some had to be rethought thanks to the feedback we received during iterative evaluations. We introduced a new informal evaluation strategy into the methodology of task-centered design. This strategy, based on the Cognitive Dimensions framework, helped us refine the design and also contributed to the user evaluation of our implementation. To the best of our knowledge, the CDs framework has not been previously used in the context of task-centered design.

We conclude, from our experience, that task-centered design can and should be used when developing novel tools for software developers.

Chapter 4

iXj: A Language for Interactive Transformation of Java Programs

Similarly to the tools described in Chapter 2, iXj exposes a language-based structural model of source code. Unlike those tools, iXj provides substantial support to developers in learning, understanding, and manipulating this model. To enable this support, we designed a novel language for specifying transformations and constructed a user-interaction model that allows the developers to build and manipulate these transformations. In this chapter we focus on the design of the iXj transformation language and on the underlying program model that this language exposes to the developers.

4.1 Program Model For Transformations

One of the key tenets of our design (as discussed in the beginning of Chapter 3) is the balance between the ability of the users to reason about program structures and the ability of the transformation engine to interpret transformation descriptions. This dictates that the transformation language must only include those Java syntactic structures that "make sense" to a developer, and that structure-based representation must support only what is needed to specify transformations.

We address this requirement by considering it as a *data modeling problem*. Data modeling was originally proposed by Chen [17] as a way of organizing data in an application. A data model consists of *entities* that represent "things," *relationships* between these things, and *attributes* that represent properties of entities and relationships. Since transformations operate on program source code, the data model in question is the data model that manifests structural relationships in source code. Such a data model is often called a *program model*. Database researchers have long known that the choice of a data model has cognitive consequences for user performance [16]; our goal is to design a program model for Java source code understandable to a typical software developer.

There are two traditional approaches to program modeling. The first approach is to use an *abstract syntax tree* (AST) that captures syntactic relationships between various source

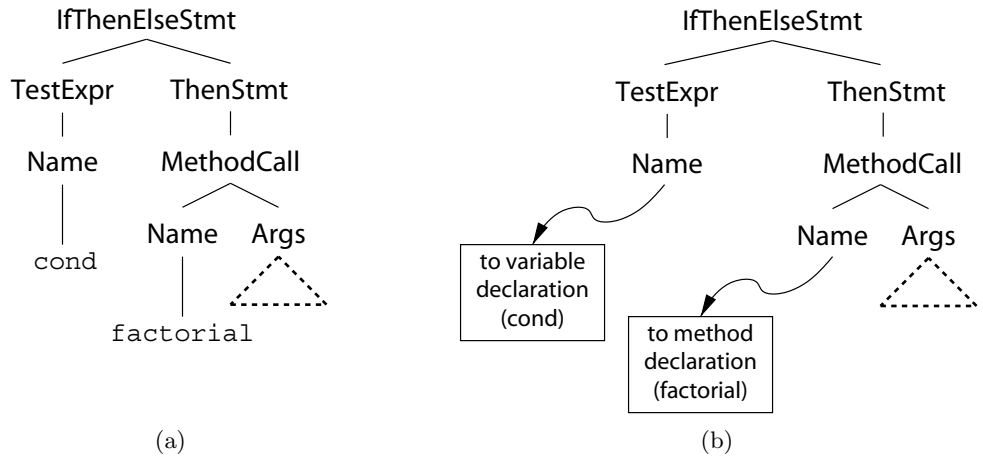


Figure 4.1: Example of an abstract syntax tree (a) and of an abstract semantic graph (b).

code elements. Each tree node (entity) represents a source code element, such as a class, a method, a statement, or an expression. The edges (relationships) between nodes reflect their syntactic nesting. A syntax tree model corresponds to the structure of source code represented in program text. This representation is often used in early processing stages of compilers and other language-based tools. We show an example of an abstract syntax tree in Figure 4.1a.

The second approach reflects the semantic composition of program elements. This model often takes the shape of a graph data structure, where each node represents a program element and the edges represent the semantic relationships between these elements. Those edges that represent the containment relationship coincide with the edges in a tree-based model. Other edges represent relationships such as class inheritance or name scoping. This representation, sometimes called an *abstract semantic graph* (ASG), is used in the later stages of language-based tools, following a static semantic analysis of source code. Figure 4.1b presents an example of an abstract semantic graph.

Designers of language-based tools normally use a program model that is conducive to the analyses and the manipulations that are performed by a tool. In contrast, we have designed a new tree-based model that is appropriate for presentation to a human developer.

4.1.1 Model Relationships

The primary relationship in the iXj program model is syntactic containment. Nested source code elements appear nested in the program model. For example, a method declaration is linked to its containing class declaration, a method body is linked to the containing method and so on. Each relationship between two entities is labeled according to the syntactic position that a nested entity occupies in its containing entity. For instance, an `if`-statement is linked to its constituents via three relationships—*condition*, *then*, and *else* (Figure 4.2).

All relationships are *one-to-one*—any entity can be nested within exactly one other en-

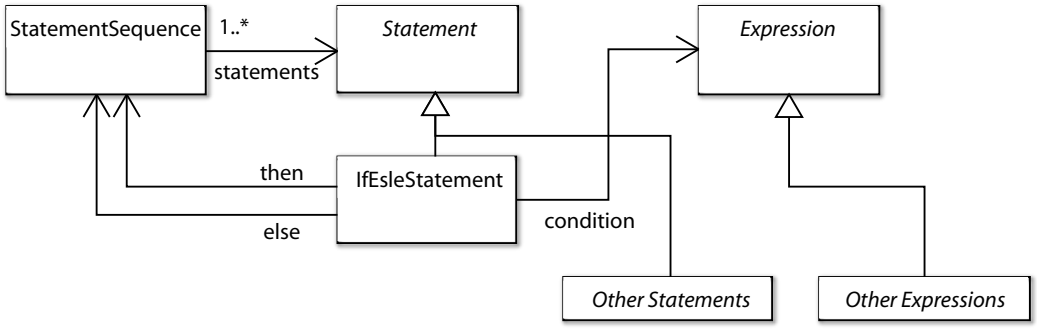


Figure 4.2: iXj program model for the `if`-statement entity presented in UML [7]. IfElseStatement is a class representing the `if`-statement. It extends an abstract Statement class and is linked to the components of an `if`-statement via three relationships (associations in UML terminology)—*condition*, *then*, and *else*.

tity. Our model makes a single exception to the syntactic nesting rule for entities that represent Java names, such as class, method, and variable names. Every Java name is treated as if it were uniquely qualified from the top level of the Java namespace. A name for a method, for example, always includes its containing class and package, as in `java.util.Vector.toString()`. A name for a local variable includes its containing method's name together with that method's argument type signature, as in `java.util.Vector.add(Object).obj`. This approach to names represents a point of departure from a purely syntax-oriented program model. We explain the significance of this scheme when we present the iXj pattern language.

4.1.2 Model Entities

The iXj program model supports two classes of entities. *Fixed-length* entities represent Java language features with fixed structure. For example, a method definition contains the modifiers, the return type, the method name, the argument list, the thrown exception list, and the method body. *Sequence* entities represent homogeneous sequences of other entities. For example, a class definition contains an unordered sequence of methods and fields.

Every entity in the iXj program model represents a construct in the Java programming language. For example, the body of a method is treated as a sequence of statements, each statement being either a Java control statement (`if`-statement, `for`-loop, etc.), an expression with a side-effect, or an assignment. Expressions are represented using a traditional syntactic nested structure. Superficially, this makes our model similar to a typical AST-based representation. But we make a number of important deviations that make understanding and manipulating the model easier for ordinary developers. For instance, a common "if-else-if" construct is "flattened" in our representation and appears to the developer similar to a `switch`-statement with branches (Figure 4.3). A multi-variable declaration statement, such as `int i, j, k` is "teased apart" and appears as a sequence of single-variable declarations, such as `int i; int j; int k`.

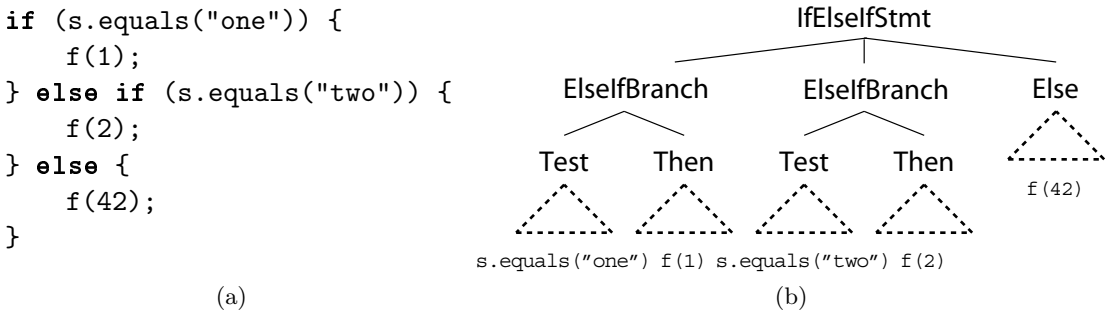


Figure 4.3: iXj model representation (b) for the “if-else-if” construct (a). The syntactic nesting is “flattened” and appears to the developer similarly to a `switch`-statement with branches.

Because non-syntactic material, such as whitespace and comments, is not supported as a target for iXj transformations, the iXj program model makes no provisions for its representation. As we show in Chapter 5, the iXj program model is never transformed *per se*. Rather, it represents a read-only data structure that is used for pattern matching. In order to preserve the documentary structure of source code, the iXj transformations are applied as text modifications to the text-based representation of source code. This approach was motivated by Van De Vanter's insights into the difficulties of maintaining association between non-syntactic material and a structural representation of source code during transformation [64].

4.1.3 Summary

The iXj program model is reflected in the design of the pattern language for transformations. No *a priori* knowledge of the program model is expected of a programmer. The understanding of our model is scaffolded by the transformation editor, making learning the new model relatively transparent. We discuss this scaffolding as part of the iXj user interaction model in Chapter 5.

4.2 iXj Source Code Patterns

Representing structural patterns in a text-based form can be awkward. After experimenting with several structural pattern languages based on text (see Section 3.7), we discovered that any moderately sophisticated pattern quickly becomes difficult to understand. We solved this problem by moving to a hybrid transformation language that combines textual and graphical elements.

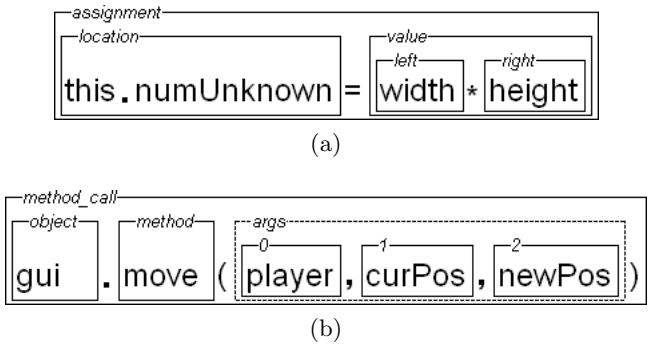


Figure 4.4: Two basic iXj patterns: solid lines indicate fixed-length structures (a); dashed lines represent variable-length sequence structures (b).

4.2.1 Basic Patterns

A pattern is represented as a source code fragment surrounded with a graphical box that identifies the programming language structure corresponding to that fragment. This graphical box maps directly to an entity in the iXj program model. Each box is labeled with a concept name on top of the box that identifies pattern structure and corresponds to an entity in the program model. The concept name is the same as the label on the program model relationship between the outer box (enclosing entity) and the inner box (nested entity). The boxes are arranged in such a way that the pattern appears as a Java source code fragment surrounded with rectangles that demarcate structural entities (Figure 4.4a).

We distinguish two types of pattern elements. Fixed-length elements are denoted by solid-line boxes; variable-length sequence elements, such as parameter lists, are denoted by dashed-line boxes (Figure 4.4b). Graphical representation of pattern elements admits pattern features not easily expressible in a Java-based textual notation. For instance, a sequence element can be visually annotated with a constraint that instructs the pattern matcher to treat that sequence as unordered.

4.2.2 Wildcard Patterns

Free variables (wildcards) in a pattern are denoted by a wildcard box (Figure 4.5a). Each wildcard box is annotated with a syntactic type that corresponds to types of entities in the program model that it can match. For example, the wildcard for a boolean condition in the `if`-statement (Figure 4.5a) indicates that it only matches expressions.

There are three kinds of wildcard patterns. The usual wildcard element represents a "match-anything" pattern, denoted by "`*`". Optional parts of language syntax admit two additional forms: "match-anything-or-nothing", denoted by "`?`", and "match-nothing", denoted by "`x`". As an example, consider matching a Java `if`-statement, when the developer does not care about the `else`-clause. In Java, the `else`-clause of an `if`-statement is optional. Yet, the wildcard pattern of Figure 4.5a only matches `if`-statements that contain some `else`-clause. In contrast, the fragment of a pattern in Figure 4.5b indicates to the pattern matcher

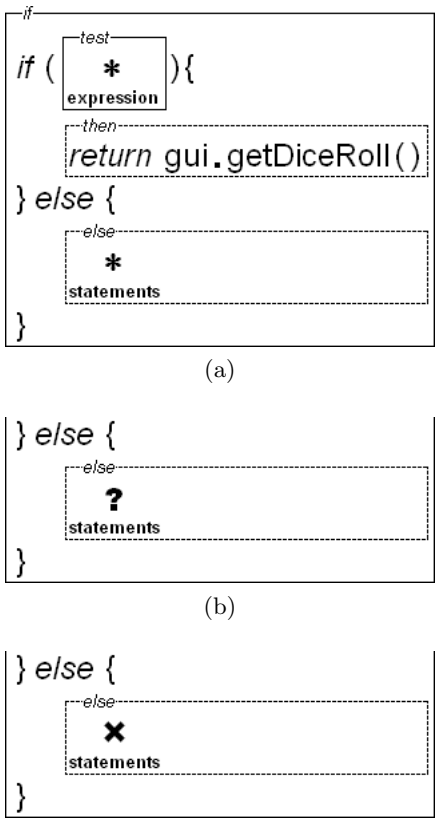


Figure 4.5: Patterns with wildcards (a), optional (b), and explicitly prohibited (c) elements.

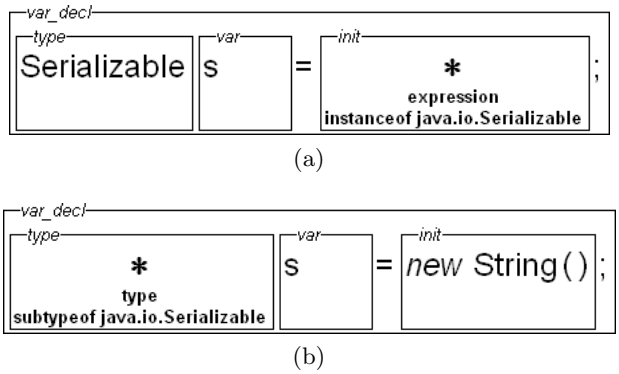


Figure 4.6: Patterns with type-constrained wildcards. The expression wildcard (a) constrains the type of the matching expression to be an instance of `java.io.Serializable`. The type reference wildcard (b) constrains the matching type reference to be a subtype of `java.io.Serializable`.

that it does not matter whether an `else`-clause exists. The developers who specifically want to prevent matching `if`-statements with an `else`-clause, could use the pattern in Figure 4.5c.

4.2.3 Semantic Patterns

The expressive power of iXj patterns extends beyond describing simple syntactic structures. The pattern language exposes the Java type system as well as the Java name-scoping rules. The type information can be used in a pattern in two ways. First, an expression wildcard can be annotated with a *type constraint*, restricting the Java type that an expression wildcard may match. For instance, a developer can instruct the pattern matcher to consider only those expressions whose (statically computed) type is a subtype of `java.io.Serializable`. This constraint is displayed as an annotation on the expression wildcard preceded by the `instanceof` keyword that relates this constraint to a Java concept familiar to developers (Figure 4.6a). Similarly, a type reference wildcard can be annotated with a type constraint that restricts matching to those type references that satisfy that constraint (Figure 4.6b). (A type reference is any mention of a type name in the program.) Type constraints are specified in a *type pattern language*. The type pattern language is presented subsequently, when we discuss other non-structural elements of the iXj patterns.

Scoping information for names is modeled in a structural fashion by treating each name as uniquely qualified from the top level (package level) of the Java namespace. Java names include class and interface names, field and method names, and local names (variables and parameters). Class and interface names are qualified with the name of their containing package, class, or method. Field and method names are qualified with the full name of their containing class. (To obey Java method resolution rules, method names are also considered together with their type signatures.) A local name is qualified with the full name of its containing method. Figure 4.7 shows some examples of the name structures. When working with patterns, the internal structure of a name is not initially shown to the

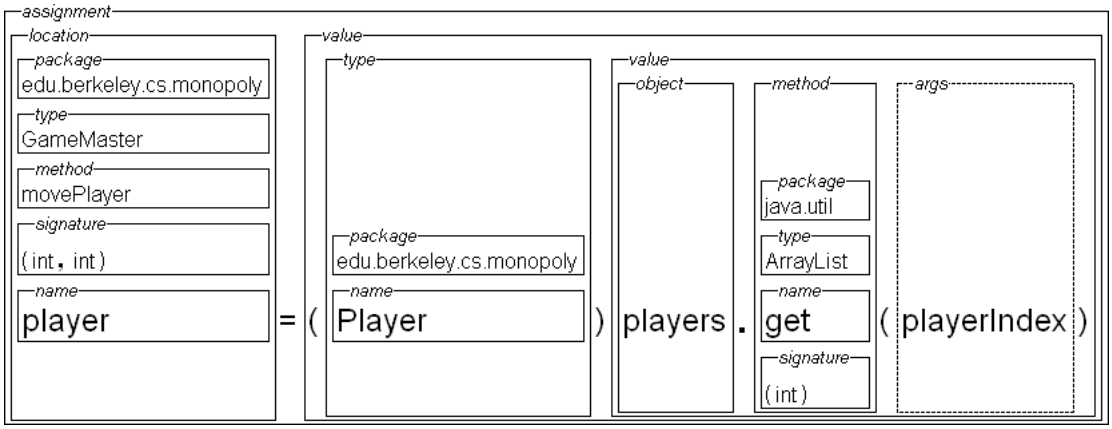


Figure 4.7: A pattern with exposed name scoping information. This pattern illustrates how type names (`Player`), method names (`get`), and local variable names (`player`) are represented in the pattern language. Our pattern editor hides this structure, unless a developer specifically wants to use scoping rules in a transformation.

developer. The pattern editor hides this structure, unless the developer specifically wants to write transformations that require the scoping rules.

4.2.4 Non-structural Patterns

Not all features of the transformation pattern (and the underlying program model) are structural. String and numeric literals, Java identifiers, modifier lists, and type constraints are matched using simple text-based pattern notation.

String and Identifier Patterns

String literals and identifiers are matched using regular expressions over strings. For example: `get.*` or `[gs]etName`. Regular patterns can include *capturing groups* demarcated by `\(` and `\)`. Capturing groups store the result of the pattern matched in pattern variables that can be referenced in the transformation action (see Section 4.3).

In the current implementation these patterns are implemented using a Java regular expression matcher (`java.regex.Pattern`), whose expressiveness is roughly equivalent to PERL5 regular expressions [70].

Numeric Patterns

Numeric literals are matched using simple patterns that express numeric relationships. When necessary, the developers can use familiar logical operators. For example: `42` or `<42` or `>=42 && <=54`. Numeric patterns that begin with a logical operator compare the value being matched against the numeric literal in the pattern using that operator. Numeric patterns are constructed according to the following grammar:

NUMBER	→	*a Java numeric literal*
NUMERICPATTERN	→	NUMBER
	|	`!=` NUMBER
	|	`<` NUMBER
	|	`>` NUMBER
	|	`<=` NUMBER
	|	`>=` NUMBER
	|	`(` NUMERICPATTERN `)`
	|	`!` NUMERICPATTERN
	|	NUMERICPATTERN `&&` NUMERICPATTERN
	|	NUMERICPATTERN `||` NUMERICPATTERN

The precedence of operators in the numeric patterns corresponds to their precedence in the Java programming language.

Modifier Patterns

Java modifiers are treated as boolean annotations that can be tested for their presence or absence. Modifier patterns can be combined with conjunctions and disjunctions. For example: `public || protected` or `public && static && !final`. The following grammar formally defines construction rules for the modifier patterns:

MODIFIER	→	*one of Java type, field, method, or variable modifiers*
MODIFIERPATTERN	→	MODIFIER
	|	`(` MODIFIERPATTERN `)`
	|	`!` MODIFIERPATTERN
	|	MODIFIERPATTERN MODIFIERPATTERN
	|	MODIFIERPATTERN `&&` MODIFIERPATTERN
	|	MODIFIERPATTERN `||` MODIFIERPATTERN

The rule "MODIFIERPATTERN MODIFIERPATTERN" is equivalent to "MODIFIERPATTERN `&&` MODIFIERPATTERN", but permits a more familiar specification of modifier conjunction, such as `public static final`.

Type Constraint Patterns

The developers can use type constraint patterns in two contexts. First, a type constraint can be specified on expression wildcards. This constraint restricts the match of the wildcard to those expressions in the target program whose statically computed type satisfies the constraint. Second, a type constraint can be specified on a type reference wildcard. This constraint restricts the match to the type references in the target program that satisfy the constraint.

Type patterns are constructed using `subtypeof` and `supertypeof` operators that can

be combined with conjunctions and disjunctions, just as other patterns. For example: `subtypeof Operator && subtypeof java.lang.Cloneable.`

QualifiedTypeName	→	*fully qualified Java name for the type*
TypePattern	→	QualifiedTypeName
	|	`subtypeof` QualifiedTypeName
	|	`supertypeof` QualifiedTypeName
	|	`(` TypePattern `)`
	|	`!` TypePattern
	|	TypePattern `&&` TypePattern
	|	TypePattern `||` TypePattern

The `subtypeof` operator ensures that the type being matched is the same as the type specified or a subtype of it. The `supertypeof` operator is a reverse of `subtypeof`. This operator checks that the type being matched is the same as the given qualified type name or one of its base types.

4.3 iXj Transformation Actions

Each pattern element can be associated with a transforming behavior. This behavior, triggered when that pattern element matches, specifies the result of the transformation in the iXj action language. In contrast to the mostly structure-based pattern language, developers specify actions as replacement source code text, augmented with a special syntax for referencing fragments of the matched pattern. When a transformation pattern matches, the action text is pre-processed to substitute referenced fragments from the match. Post-processed action text for a pattern element replaces the section of source code corresponding to the match of that element.

Figure 4.8 shows two complete transformations consisting of patterns and actions. The developers reference fragments of the matching text by specifying a path to the pattern element using label names, separated with dots. The matching fragment is referenced in an action by enclosing its path between `$` characters. For example `$if.test$`, `$if.then$`, and `$if.else$` refer to the boolean test and the two branches of an `if`-statement. Because the names are automatically assigned to boxes, the transformation writers do not need to worry about explicitly naming relevant parts of the pattern. If the last element of a name refers to a pattern element containing a string pattern with a capturing group, the developer can access the captured text by appending $\backslash n$ to the last element of a name. We borrowed this notation from the regular-expression-based text processing tools—n refers to the n^{th} capturing group. (Figure 4.9b contains an example of such an action.)

The transforming behavior need not apply to the top-level pattern element. An action can be associated with any of the sub-patterns, in which case only that part of the matching source code fragment is affected by the transformation (Figure 4.8b). Nested actions are prohibited—the outer action always affects a larger source fragment, erasing the effects of the inner action.

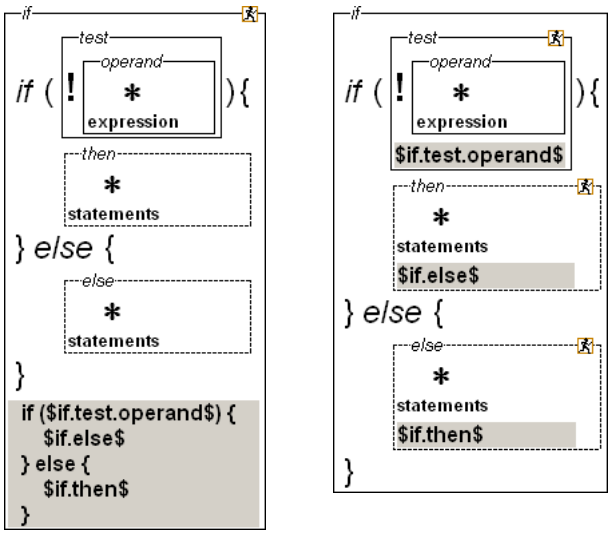


Figure 4.8: Two complete transformations that remove a boolean negation from an `if`-statement and reverse its branches. This example illustrates how an action can be specified either at the top level pattern element (a) or at one of the sub-patterns (b).

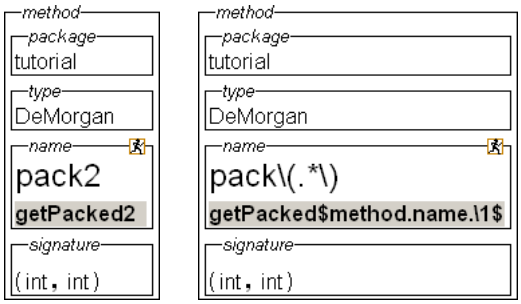
4.4 Refactoring With iXj: A Case Study

iXj is not a replacement for automated refactoring tools. It does, however, enable transformations that are less broadly applicable, not necessarily behavior preserving, and project- and task-specific. The availability of iXj-based transformations can complement the refactoring facilities found in modern development environments. In this section we illustrate how iXj transformations can be used to implement several common refactoring transformations. We concentrate only on those parts of refactoring transformations that require many similar and repetitive edits. While iXj can be used to specify local one-time modifications, those are typically performed by hand.

4.4.1 Rename Method

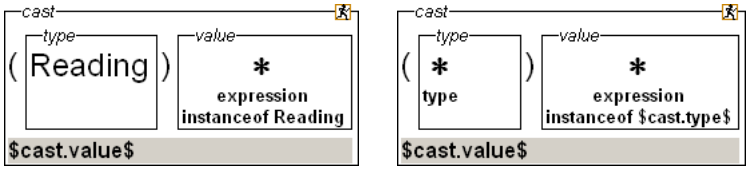
The *Rename Method* refactoring is applicable when "the name of a method does not reveal its purpose" (Fowler [31], p. 273). The transformation presented in Figure 4.9a implements this refactoring for method `tutorial.DeMorgan.pack2(int, int)`. The transforming action attached to the method's name replaces it with `getPacked2`. This transformation applies to references to the method's name at each call site and in the method's declaration. Similar transformations can be created to rename fields, variables, or classes.

Existing automated refactoring tools do not support renaming many similar entities in a single step. In contrast, the iXj transformation language permits such an operation. For example, the transformation in Figure 4.9b renames all methods in `tutorial.DeMorgan`



(a) (b)

Figure 4.9: Example of the *rename method* refactoring transformations in iXj.



(a) (b)

Figure 4.10: Example of the *encapsulate downcast* refactoring transformations in iXj.

whose name starts with `pack`. The suffix of the matching method's name is stored in a pattern variable by the `\(` ... `\)` capturing group, which is referenced in the transforming action as `\1`. This transformation can be generalized to rename all methods whose name starts with `pack` by wildcarding the package, the type, and the signature in the pattern.

4.4.2 Encapsulate Downcast

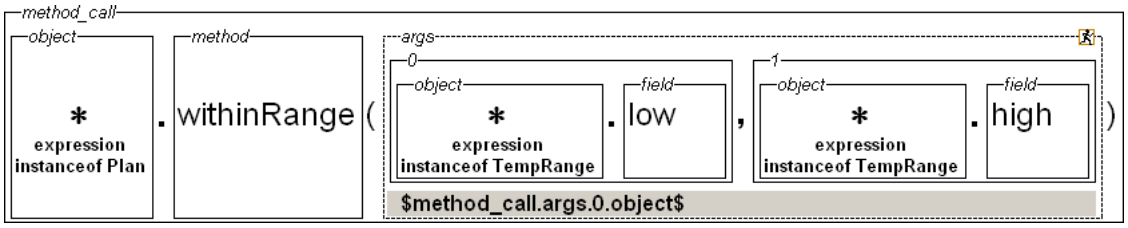
When "a method returns an object that needs to be downcasted by its callers" (Fowler [31], p. 308), it is usually advisable to move the downcast into the method's body. This results in many unnecessary casts in the source code. For example:

```
Object lastReading() {
    return readings.last();
}
...
Reading r = (Reading) lastReading();
```

⇒

```
Reading lastReading() {
    return (Reading) readings.last();
}
...
Reading r = (Reading) lastReading();
```

After this refactoring takes place every call to `lastReading()` contains the leftover casts to `Reading`. The developers can use iXj to clean up these casts. The transformation in Figure 4.10a implements this change by matching all casts to `Reading` whose casted expression's type is `Reading`. This transformation replaces the entire cast with the casted expression, thereby dropping the unnecessary type. A more general transformation shown in Figure 4.10b can be used to remove all unnecessary casts for all possible types.



(a)

Figure 4.11: Example of the *preserve whole object* refactoring transformations in iXj.

4.4.3 Preserve Whole Object

The *Preserve Whole Object* refactoring is useful when a program is "getting several values from an object and passing these values as parameters in a method call" (Fowler [31], p. 288). For example:

```
plan.withinRange(getRange().low,
                 getRange().high);
```

⇒

```
plan.withinRange(getRange());
```

By applying this transformation, the developer can use other features of the range object in the `withinRange()` method. Unfortunately, when this method is called in many places in the program, modifying every call site can be time consuming. The transformation for implementing this change is shown in Figure 4.11. This transformation looks for all calls to the `withinRange()` method and modifies its argument list to contain just the expression that returns an instance of `TempRange`. The body of the `withinRange()` method needs to be modified manually to account for its new argument signature. In a typical development environment, the need for this change would be indicated by a compilation error.

4.5 Program Models for Source Code Manipulation

As language-aware source code manipulation tools become mainstream, there is a growing interest among tool builders in using language-based program models. These program models can take various shapes and forms, representing a point of departure from a traditional approach to modeling source code as ASTs and ASGs. Not surprisingly tool builders are increasingly discovering that different applications are best supported by different program models.

For instance, when studying the implementation of Java Development Tools in the Eclipse IDE [29], we counted seven distinct Java parsers. Each of these parsers caters to the needs of the specific client. All parsers share some common code through object-oriented factoring of parser classes. Each parser, however, supplies its own program model to represent the results of the parse. For example, the source element parser provides an outline view of the Java source code to the level of field and method declarations. A completion parser is used by the code-completion feature to parse the context surrounding the

location at which the developer requests a completion. A code snippet parser is used by the debugger for parsing small fragments of Java source code. Other parsers are used for compilation, hyperlink navigation, language-aware source code searching, and so on.

The need for different program models is motivated by different contexts in which these models are used. A code formatting tool needs to maintain code comments in its model; a compiler does not. Some models need to be constructed from ill-formed, incomplete, or inconsistent states of source code and represent only partial information about the program. (Van De Vanter [63] describes this as the "I3" condition.) Even when the information that needs to be represented is fixed, there is a considerable degree of variation in the way it can be modeled. We explored some of these issues in our earlier work on an exchange format for the Harmonia framework [10]. In our present work, the design of the program model for program transformations was driven by the need to expose that model to the users of our tool.

There has been considerable interest in finding the holy grail of program representations—a canonical, universal all-purpose program model. Recent examples include the work on JavaML [1], on srcML [20], and on the standardization of tool exchange formats [30, 38]. Yet, time and again we see the designers struggling with the decision of which features must be included in the model and how to bring existing tools into compliance with the design.

Our work demonstrates that an application-centric program model can be more appropriate. This notion was introduced in the early work on TXL [23] and REFINE [12]. Because these tools include a parser as part of the transformation engine, for any given transformation the developers can use a grammar that derives the structure needed for that transformation. More recently, TXL designers published a report on *agile parsing*—a technique for adapting the language parsing grammar to the needs of a particular application [25]. Our experience suggests that designing custom models on a case-by-case basis is the right approach for building language-based tools. However, extracting model instances from source code text and maintaining the correspondence between various models is a topic for another dissertation.

4.6 Visual Languages and Program Transformations

iXj is a visual language. Our graphical notation for the transformation language was inspired in part by diSessa's Boxer programming language [27]. Boxer's computational element is a box that comprises a visual presentation and computational semantics. As in iXj, Boxer's boxes can be nested and their spatial arrangement reflects the semantics of the computation. Boxer's boxes can be collapsed and expanded to hide the details of the computation. This mechanism is similar to the folding and unfolding of the iXj's patterns that is implemented in the iXj pattern editor. iXj's transforming actions are similar to Boxer's computation semantics of boxes. Unlike Boxer, iXj enforces strict layout rules on the position of pattern boxes on the screen. This is necessary to align pattern boxes in such a way that the pattern looks visually like source code.

iXj's representation of the hierarchical program model as nested boxes is related to

treemaps, an approach to visualization of hierarchical data [42]. The treemaps, however, enforce no spatial arrangement of boxes, whereas in iXj the box layout is driven by the traditional visual layout of source code structures. VXT [55] is a visual language for transformation of XML documents based on their treemap representation. VXT transformations describe how to modify a document's structure visually; iXj's transformations are driven by structure, but modify a text-based representation of source code using text substitution.

The design of iXj was strongly influenced by the framework of Cognitive Dimensions of Notations (CDs) presented in Chapter 3. This choice seemed especially appropriate in the face of the CDs framework's first published use—an evaluation of the LabVIEW and Prograph visual programming languages [35]. Our use of the CDs framework to drive the design is unique in many ways. Traditionally, the CDs framework has been used to evaluate existing designs and tools. In addition to Green and Petre's work on LabVIEW and Prograph, some representative examples include evaluation of C# [18], Prolog [34], and UML diagrams [45].

In our case, the CDs framework was instrumental at the early design stages and not just as an evaluation tool. The key benefit of the CDs framework was in stimulating our thinking about the dimensions early in the design process, while there was still room for variation and time for improvement. For this reason, we believe that the CDs framework should be used more pervasively to guide the design process long before the first evaluation.

Chapter 5

Interactive Transformation of Java Programs in Eclipse

This chapter describes the implementation of an interactive environment for transformation of Java programs in Eclipse. First, we present our tool from the user's perspective. We discuss the interaction model that we designed to support effective construction and manipulation of iXj patterns. We show how the immediate feedback and visualization mechanisms work together to provide the necessary scaffolding to the users. Second, we discuss the internal architecture of our tool, focusing on those parts that were necessary to implement our user interaction model.

5.1 iXj User-Interaction Model

Creating a structural pattern representation is challenging even in an interactive environment. We designed a user-interaction model that guides the developer through the construction of a transformation. This section presents an overview of the transformation development workflow and its implementation in Eclipse.

5.1.1 Workflow Overview

At a high level, the developer's workflow to perform a single transformation consists of the following steps:

1. The developer selects a sample source code fragment needing transformation. The system scaffolds the creation of a transformation by automatically generating an initial pattern from the textual selection in source code. The generated pattern is concrete, that is, it contains no wildcards and no matching constraints.

2. Using the generated pattern as a starting point, the developer interactively modifies the transformation by inserting wildcards, matching constraints, and transforming actions. The system displays a preview of the transformation as it is modified.

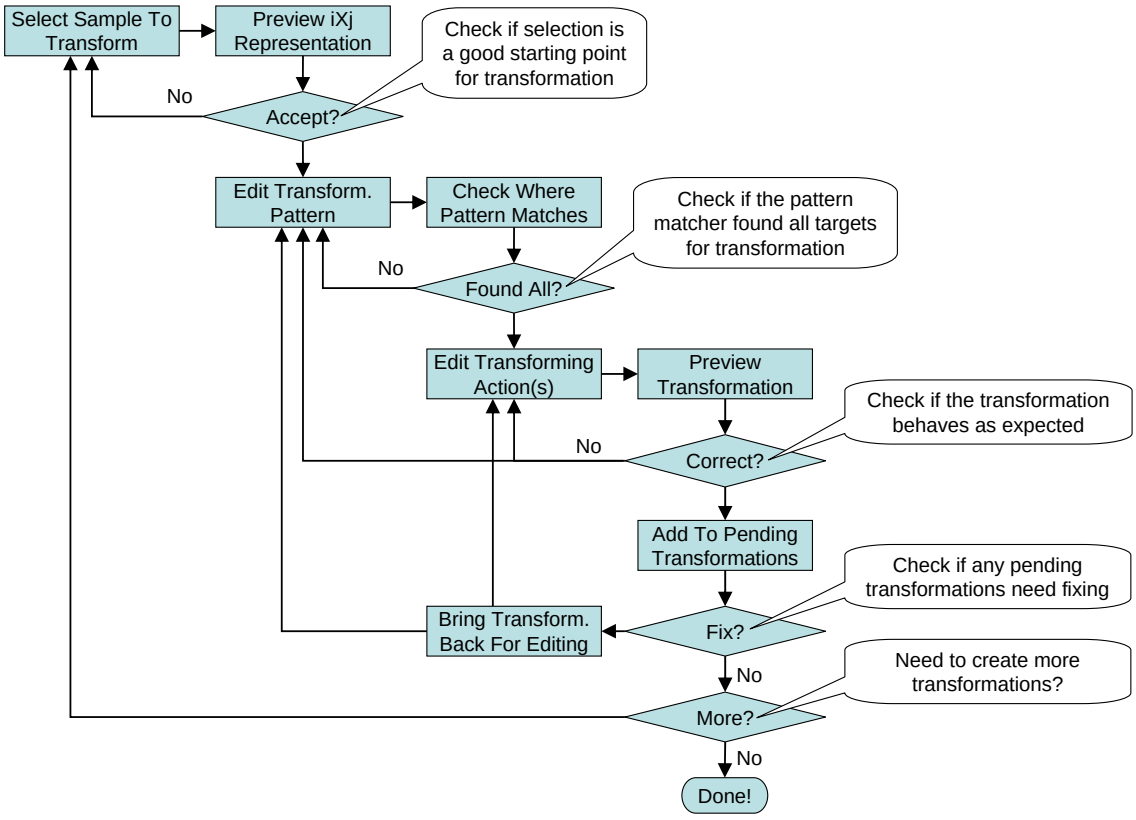


Figure 5.1: The flow of user's interaction with the transformation editor.

3. When the developer is satisfied with the result of the transformation, he or she applies the transformation to source code and returns to other coding tasks.

This workflow embodies two key principles of the interaction model: by-example construction and iterative refinement. By-example construction enables developers to start with a single instance of a transformation and to generalize the description of that transformation to apply to similar source fragments. In practice, a single conceptual change may require several related transformations. We support this through a notion of a *pending transformation set*. This set groups related transformations prior to their application to source code and helps avoid intermediate inconsistent states of source code. Any transformation in the set can be modified further, if the developer discovers that it does not affect the code as expected. When the developer is satisfied with all transformations in the set, the transformation editor can apply all of these transformation at once.

Figure 5.1 presents the developer's workflow at a finer level of detail. This figure illustrates iterative refinement of transformations, guided by the feedback from the transformation editor. The feedback helps the developers to generalize the transformation pattern and

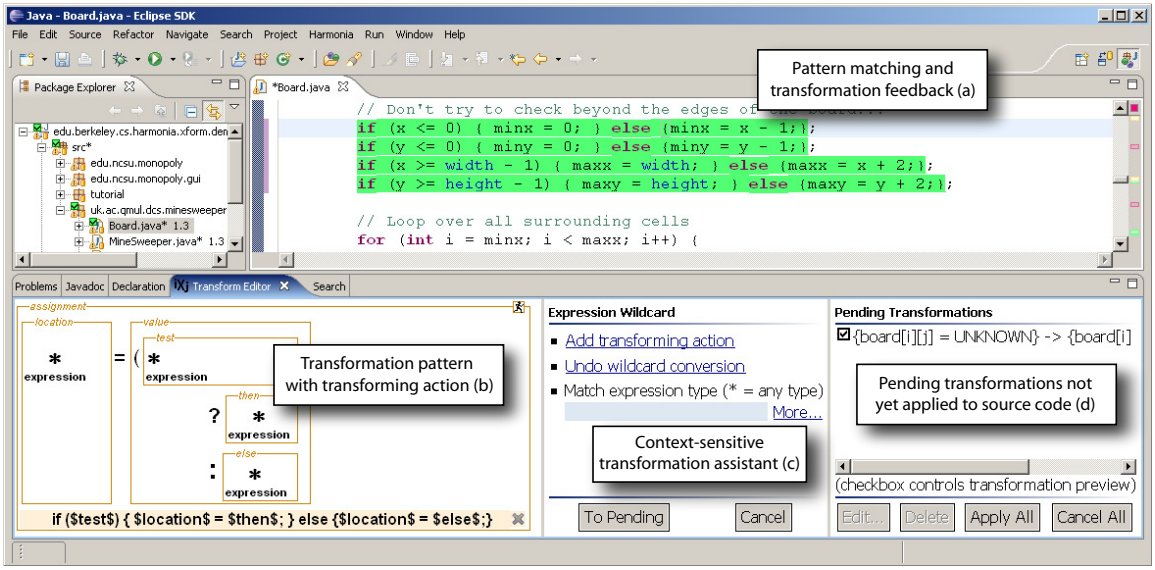


Figure 5.2: iXj plug-in for Eclipse provides an interactive transformation editor.

to specify the transforming action. It also enables developers to evaluate the set of pending transformations and to decide when that set is complete.

5.1.2 Transformation Workflow in Eclipse

We implemented iXj as a fully integrated plug-in for Eclipse. For developers, integration with the code editing workflow in Eclipse affords access to lightweight, as-needed transformations. The transformation capabilities are available at any time when the program is in a well-formed syntactic and semantic state.

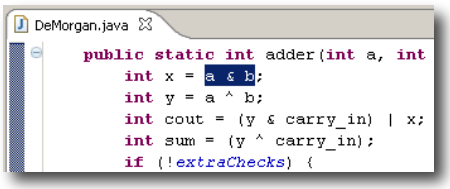
Figure 5.2 presents four major components of the iXj user interface.[1] Developers interact with the iXj transformation editor through a pane (a "view" in the Eclipse terminology) below the editor window (Figure 5.2a). The transformation editor consists of three panes: the pattern editor (5.2b), the transformation assistant (5.2c), and the pending transformation list (5.2d).

To illustrate the process of building a transformation with the iXj plug-in for Eclipse, we will use a simple transformation that applies de Morgan's law to a bitwise-`and` operation on integer expressions as follows:

```
int x = a & b;
```
$\Rightarrow$
```
int x = ~(~a | ~b);
```

The developer initiates the transformation process in the Eclipse source code editor by selecting a sample source code fragment that needs to be changed. The selection is performed using traditional textual selection operations.

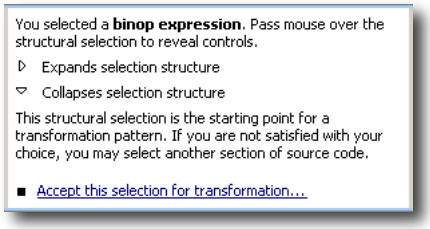
[1] Figures 5.2a-d designate the panes with comments labeled (a)-(d).



The selection is unconstrained; however, the transformation editor activates only when the developer selects a structurally complete source code fragment, such as a statement or an expression. When this happens, the system automatically generates an initial pattern from the selection in source code. This pattern appears in the transformation editor pane.

The initial pattern matches the exact source code fragment selected by the developer and all the other source code fragments that are textually similar to the selection (whitespace and comments are ignored by the pattern matcher). Sub-patterns are not shown initially to provide a less cluttered view.

The transformation assistant describes the selected structure and reminds the developer what can be done next.



The developer can select another source code fragment if the selected structure does not appear to be a good starting point for a transformation. The developer can also choose to "accept this selection for transformation."

Having accepted a selection, the developer can change the structure of the pattern to make it more general. The transformation editor offers manipulation of the transformation structure through direct manipulation and through the transformation assistant. The transformation engine uses annotations in the source code editor to provide feedback to the developers. A complete transformation may be stored in the pending transformation list prior to application. The following sections describe each of these mechanisms.

Direct Manipulation

The developer can manipulate the transformation descriptions directly by invoking visual "handles" to modify pattern structure and by changing pattern and action text.

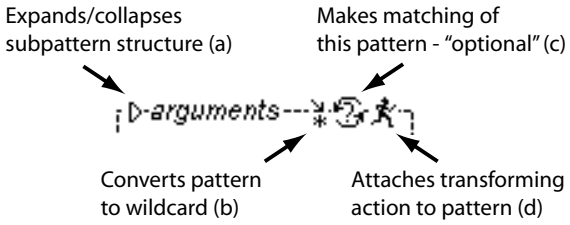
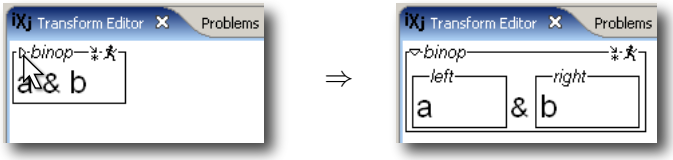


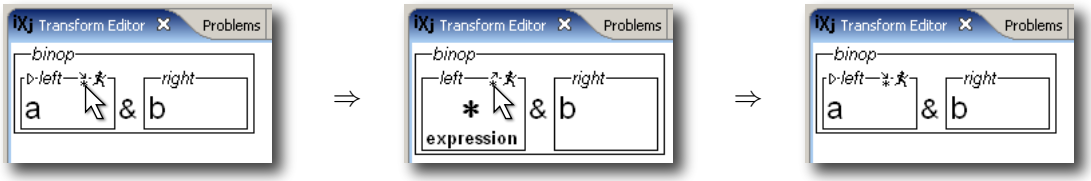
Figure 5.3: iXj patterns structures can be manipulated with "handles" attached to the pattern boxes.

Handles. Each pattern box provides handles that enable the developer to modify pattern structure (Figure 5.3). The handles permit expansion and collapse of the pattern structure (5.3a), conversion of a pattern element to and from a wildcard (5.3b), cycling through the optionality states of a pattern element (5.3c), and addition and removal of a transforming action to a pattern element (5.3d). To reduce visual clutter, the handles appear on the screen only when the developer passes a mouse pointer over the pattern box.

Expansion and collapse of the pattern structure. This handle controls the visual appearance of a pattern, but does not change its structure. It permits "drilling down" into the pattern, when the developer wants to manipulate a substructure that is not initially visible. For example, the developer can expand the initial pattern to see more of its structure.

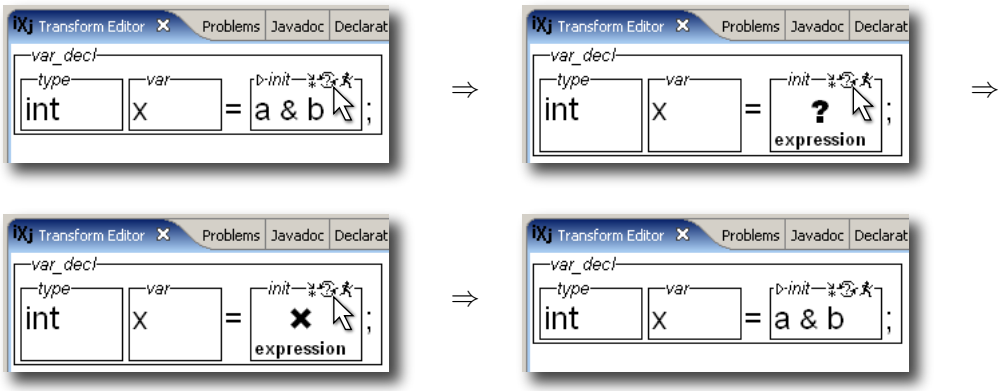


Conversion of a pattern box to and from a wildcard. This handle toggles between a concrete pattern box and a "match-anything-of-this-type" wildcard. The syntactic type of the wildcard is determined by its context in the pattern. For example, a concrete Java expression in the initial pattern can be converted to a wildcard that will match any Java expression at its position.

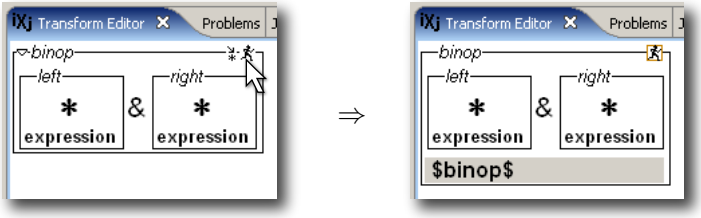


Every handle operation, including conversion to a wildcard, is reversible. The second toggle of the wildcard handle converts the corresponding pattern element back to its unwildcarded form.

Cycling through the optionality states of a pattern element. This handle only appears on pattern boxes that are optional in the Java syntax, such as the `else`-clause in the `if`-statement. Invoking this handle cycles through three optional states: (1) match anything appearing in this pattern position, including nothing (zero-or-one match), (2) match nothing (zero-match), and (3) match current pattern box, which may be a wildcard (one-match). Our running examples does not contain any optional elements. Assuming, however, that the developer selected the entire variable declaration, we can see that a variable declaration in Java can contain an optional variable initialization expression.



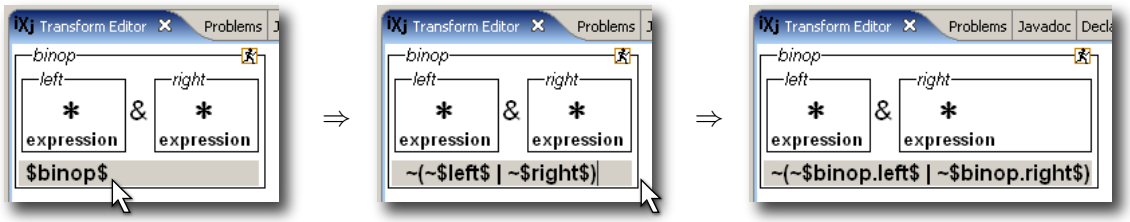
Addition and removal of the transforming action. This handle adds (or removes) the transforming action for the pattern element. The action is initialized to an identity transformation that replaces the source code fragment matching that element with itself. To perform the de Morgan law transformation, the developer adds an action to the bitwise-`and` expression.



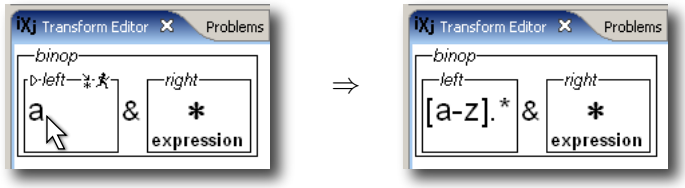
As with other handles, the action may be removed by re-invoking its handle.

Pattern and Action Editing. The transformation editor permits free-form text editing of the transforming action and parts of the transformation pattern. Because the transforming action is specified in a text-based format, the developer can edit the action simply by clicking on its text. Doing so creates an editable input field that accepts all standard text editing commands. In order to reduce typing, long pattern variable names can be abbreviated to their *least unique qualified suffix*. The suffix must contain a complete box

name, but this name need not be qualified beyond what is necessary to unambiguously resolve that name in the currently visible expansion of the pattern. For instance, in our running example it is sufficient to refer to `$left$` and `$right$`, rather than `$binop.left$` and `$binop.right$`. The names are expanded to their long unambiguous form when the developer leaves (clicks outside of) the action editor.



The transformation editor permits free-form editing of the non-structural elements in the transformation pattern. These elements include string and numeric literal patterns, modifier patterns, and Java name patterns. Clicking on such an entity in the transformation editor creates an editable input field. The developer can edit that field to specify a more complex pattern using the text-based pattern language appropriate for that entity (see Section 4.2). As the text is edited, it is verified against the specification of the corresponding pattern language. If the pattern does not conform to the specification, the developer is notified by a change in the pattern's color. For example, suppose the developer wanted to apply the de Morgan transformation to just those bitwise-`and` expressions whose first operand is a variable that starts with lower-case letter (contrived, but possible). In this case, he or she could modify the pattern as follows.

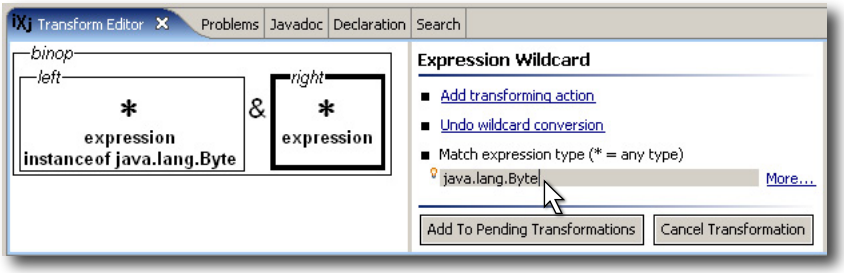


Transformation Assistant

Any pattern element in the transformation editor can be selected by clicking anywhere inside its box. The context-sensitive transformation assistant provides a description of the selected pattern element and lists various actions that apply to it. When no specific pattern element is selected, the transformation assistant describes each of the handles to remind the developer of their purpose.

In addition to enabling all actions accessible via handles, the transformation assistant includes several options that are not available through direct manipulation. These options include specification of various matching constraints, such as a constraint on the Java type of

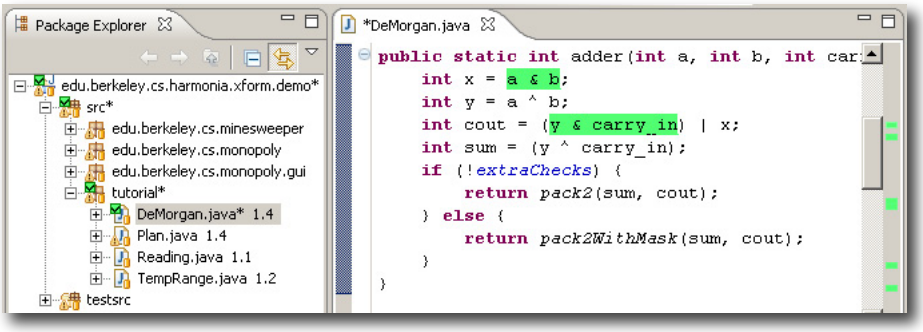
an expression that can be matched by an expression wildcard. For example, if the developer wants to constrain the de Morgan transformation to apply only to the instances of `java.lang.Byte`,[2] he or she can use the transformation assistant to specify this constraint.



The input field for the type constraint in the transformation assistant offers type-name completion; a full list of known type names is accessible by following the "More..." hyperlink.

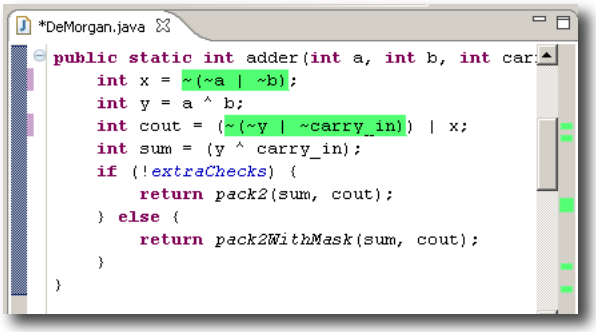
Immediate Feedback

As the pattern is edited, the pattern matcher runs continuously, providing visual feedback to the developer. The pattern matcher highlights all matches in the source code editor. An overview ruler on the right-hand side of the source code editor provides simple navigation to the matches in the same source file that are not immediately visible. Tick-mark annotations on source files and packages in Eclipse's package explorer indicate presence of a match within a source file.



The developers can associate a transforming action with a pattern at any time. Frequently, the developers will experiment by adding an action to a concrete pattern, then making the pattern more general by adding wildcards, and finally changing the action to introduce missing pattern variables. The effect of the transformation is displayed immediately upon specification of an action; however, the results are not yet "committed" to the source code. For example, when the developer edits the action in the de Morgan law transformation, the transformation engine immediately updates the view in the source editor:

[2]This assumes that the program to which the transformation is applied relies on Java 5 automatic unboxing semantics.

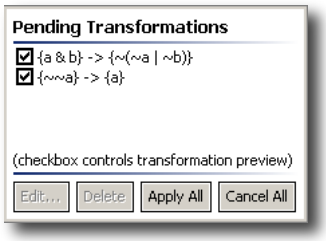


The immediate feedback provided by the transformation engine makes the execution of transformations transparent to the developers and assists in their learning the transformation language. The interface encourages experimentation by enabling the programmers to view partial results and to visualize the effect of the transforming action.

Pending Transformation List

Upon completion of a single transformation the developer adds that transformation to the list of pending transformations. This list groups related transformations together prior to application, helping to avoid intermediate inconsistent states of the source code. The effects of pending transformations can be previewed, but they are not "applied" until the developer decides to do so. Often, when working on a related transformation, the developer realizes that one of the pending transformations is incomplete and continues to modify that transformation until it behaves as expected. The developer can independently toggle the preview of any pending transformation to visualize its effect on source code. The list of pending transformations effectively forms a "transformation plan," making it easier to isolate a large source code change consisting of several related transformations.

Each transformation in the pending list is represented in a text format that describes how that transformation changes the source code fragment originally selected in the by-example construction. For example, when applying the de Morgan law transformations, the pending transformation list might contain the following transformations:



The second transformation in this list cleans-up double-negations left over after the de Morgan law transformation.

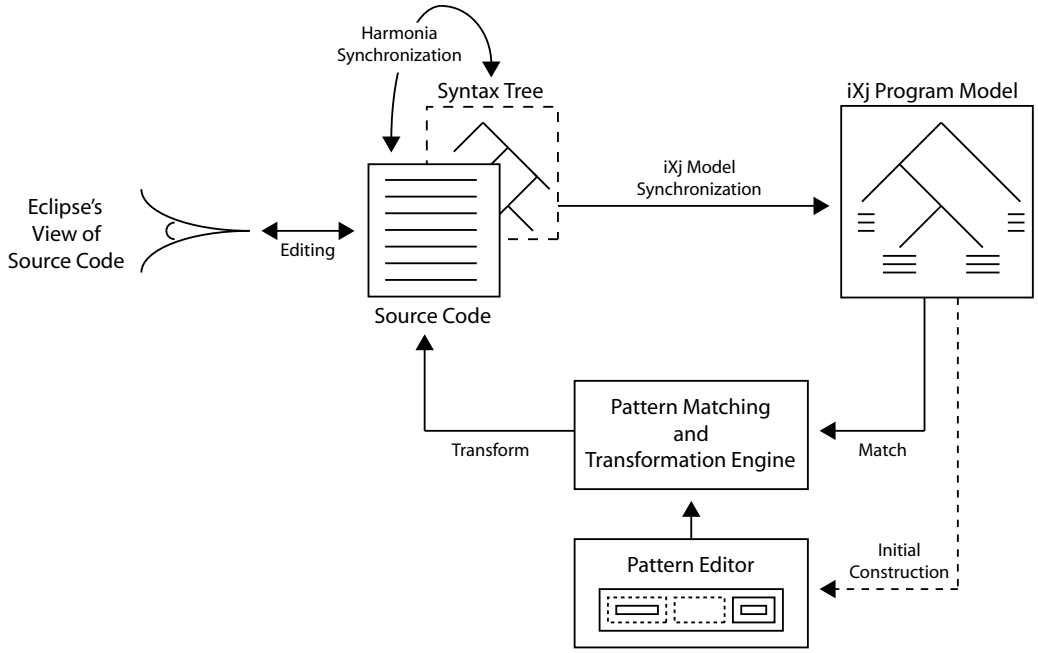


Figure 5.4: The architecture of the iXj plug-in for Eclipse.

5.2 iXj Architecture

The current implementation of the iXj plug-in is built on top of the Eclipse platform [29]. Eclipse provides the infrastructure for building interactive development tools and, together with Java Development Tools (JDT), is considered to be one of the most popular Java IDEs. Eclipse JDT offers some leverage for building the language-specific source code analysis components that are necessary for the advanced language-based tools, such as iXj. Nevertheless, to increase implementation flexibility we decided to build iXj on top of the analysis infrastructure provided by the Harmonia framework described subsequently.

Figure 5.4 presents an architectural overview of the iXj plug-in. Source code editing in Eclipse is facilitated by document objects that provide an in-memory representation of source files. The Harmonia framework maintains a syntax-tree based representation of source code that servers as a backing store for Eclipse's document objects. The syntax trees support free-form editing of the program text; the structural consistency is transparently maintained by the Harmonia analysis engine. The iXj plug-in maintains the correspondence between the Harmonia syntax trees and the iXj program model. The program model serves two purposes. First, it is used to map a text-based selection of the initial source fragment to a transformable structural entity. This mapping is used to generate the initial concrete pattern. Second, the iXj transformation engine uses the program model for pattern matching. Each match of a structural pattern in the model is mapped to the text-based representation of source code, permitting its subsequent modification by the transformation engine.

Figure 5.5: Component-level view of the Harmonia architecture. Language modules extend the analysis kernel with language-specific information. An application, implemented in one of the supported programming languages, uses the services of the analysis kernel through bindings appropriate for that programming language.

The iXj pattern matcher uses a syntax-directed top-down tree traversal of the iXj program model. The transformation engine applies the transformation textually to the part of the source code text that corresponds to the match. Implementation of changes on the text-based representation permits the least disruptive modification of source code with respect to the non-syntactic material such as whitespace and comments. The text changes are subsequently incorporated in the syntax tree-based representation by the Harmonia analysis engine.

The iXj pattern editor uses the iXj program model to construct the initial representation of the pattern structure. The editor uses a "box-and-glue" layout algorithm for presenting pattern structure as nested graphical boxes. Our algorithm uses baseline-alignment constraints to ensure that the source code text contained in graphical boxes looks visually like source code. This algorithm was inspired by the layout mechanisms of TEX [44]. Horizontal layout and spacing are controlled by typesetting rules, similar to those formulated by Baeker and Marcus for C programs [2]. This mechanism was derived from our earlier work on displaying and editing source code in a programming environment [65]. When a pattern spans multiple lines of source code, the algorithm uses a simple line-breaking strategy that terminates the horizontal layout at appropriate points in the structural context (for example, after the opening curly brace in the `if`-statement's body).

5.2.1 The Harmonia Framework

The Harmonia framework resulted from our earlier research in building language-based tools and services [8]. Figure 5.5 presents a high-level view of the Harmonia architecture. The three major components of the Harmonia framework are the language kernel, the language modules, and the application interface layer that exports Harmonia APIs to several programming languages.

The *language kernel* (LK) provides the abstractions for modeling program source code and the language-independent infrastructure for incremental program analyses. Language-specific analysis details are encapsulated by *language modules* that can be demand-loaded into a running Harmonia application. The Harmonia framework supports simultaneous

loading of multiple language modules. Language modules encapsulate language-specific analysis details, such as the lexer definition, the parse tables, and tree node definitions. Language modules parameterize the behavior of the framework for the particular language. To create a language module, the system is given lexical, syntactic and semantic descriptions that are compiled into a dynamically loadable library. The analysis engine can support any textual language that has formal syntactic and semantic specifications. Descriptions exist for Java, C, C++, XML, COBOL, and several other languages used internally by the Harmonia framework.

The ability to analyze Java source code quickly and efficiently was the primary motivating factor in using Harmonia as the back-end analysis infrastructure for iXj. Moreover, the ease with which new languages can be introduced to the Harmonia framework was instrumental in building the iXj transformation editor. In just one day we were able to create a set of language modules for each of the textual pattern languages used in the iXj transformations to describe literal, name, and modifier patterns. This enabled us to reuse syntactic analysis infrastructure implemented in the Harmonia framework, rather than build custom lexers and parsers for each of those languages.

Any application that uses the Harmonia framework can be implemented in the language of the programmer's choice. In addition to the APIs in its implementation language (C++), the Harmonia framework includes bindings for other popular programming languages, including Tcl, Java, and Lisp. Harmonia features are provided to Eclipse by a set of Java plug-ins that access native Harmonia libraries through the Java Native Interface (JNI) [51].

The Harmonia Language Kernel

The Language Analysis Kernel lies at the core of the Harmonia framework. Figure 5.6 presents its architecture in more detail. The central component of the language kernel is the program representation infrastructure that consists of abstractions for modeling program source code and the programming language grammar. The source code abstractions are maintained using language kernel analysis services. The editing and analysis model governs the interface with the Harmonia application.

Language-based program representation. Program source code is represented in Harmonia as syntax trees that are constructed and maintained by the analysis engine. In addition to serving as the data structure for source code analysis, syntax trees provide a document model that can be queried and manipulated by the clients. Harmonia syntax trees are produced by the parser. Yet, they are distinct from parse trees used in traditional language-based tools in several important ways. First, the Harmonia trees are "mostly" abstract, that is, the nodes in a tree correspond to linguistic entities in the high-level abstract grammar for the programming language. This is achieved by using Generalized LR (GLR) parsing technology [69] that permits a flexible and concise specification of the language syntax. Second, the Harmonia trees provide representation for non-linguistic material such as whitespace and comments. This allows a syntax tree to serve as the sole representation of program text, eliminating the need for consistency maintenance among several different

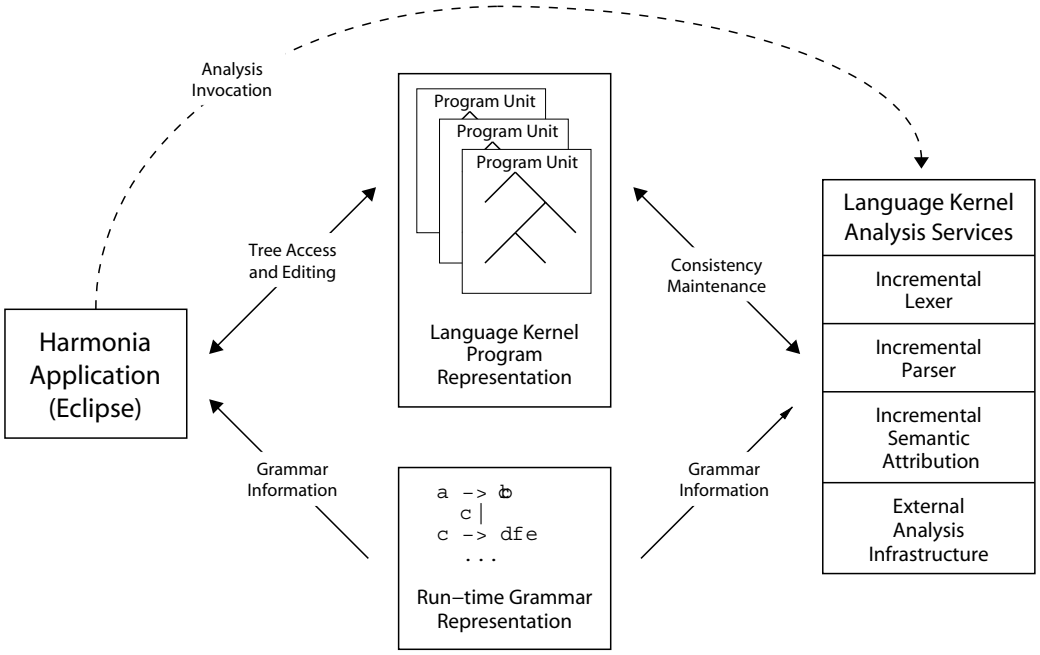


Figure 5.6: The architecture of the Harmonia language kernel.

representations. Third, the Harmonia syntax trees embody a language-specific document object model. This is achieved by automatically generating class definitions for syntax tree nodes from a grammar-based declarative specification. This specification also facilitates translation between the Harmonia document object model and the iXj program model.

Harmonia syntax trees are modeled as self-versioned documents, constructed from versioned primitive data types [68]. The document representation incorporates a fine-grained history of changes, facilitating access to previous versions of the document. This supports powerful forms of undo operations, as well as providing an underlying mechanism for maintaining a development history. Because the versioning is organized structurally rather than temporally, changes to particular portions of the program can be recovered even if other changes were made in between. The ability to "roll-back" any document version is important for iXj's implementation. It enables the preview of the current and of the pending transformations directly in an Eclipse editor by temporarily modifying the syntax tree that provides the backing store for an Eclipse editor.

In Eclipse, a Harmonia syntax tree serves as a backing store for an Eclipse document in place of the traditional character-based data structure. The same tree can be simultaneously accessed directly through the Harmonia APIs. The iXj plug-in uses the Harmonia API to construct its program model from the syntax tree. The transformation engine uses the text-based access to perform text-substitution operations. The correspondence between the two representations is maintained by Harmonia; changes to one are immediately reflected in changes to the other.

Incremental analyses. The analysis engine of the Harmonia framework provides fine-grained incremental lexical and syntactic analyses that can both construct the syntax trees from traditional text files and incorporate changes to the syntax trees incrementally, as they are introduced. The framework includes an incremental GLR parser, an efficient incremental scanner, and an infrastructure for building semantic analyses.

In order to give the developer needed flexibility in modifying programs, the framework continues to provide services when programs are ill-formed, incomplete, or inconsistent. The incrementality of the Harmonia parsing algorithms, together with history-sensitive error recovery [67], naturally incorporates inconsistency into the syntax tree by enabling syntax analysis to continue beyond malformed regions, and by enabling malformed regions to contain well-formed substructure. The iXj transformation engine takes advantage of this feature to enable transformations on all regions in source code that do not contain syntactic or semantic errors.

Editing and analysis model. The Harmonia framework permits unrestricted editing of the syntax tree data structures. The source code representation can be changed without concern for transient ill-formedness introduced during an edit. A change discovery mechanism based on the persistent document representation is used to incrementally restore consistency following a modification. The frequency of reanalysis is under control of the analysis clients; in Eclipse, a reanalysis is invoked after a brief pause in the user's typing.

This analysis model is instrumental for permitting text-based transformation of the program structure. When the iXj transformation engine applies a transformation to source code text, the program structure is recovered by a subsequent analysis.

5.3 Design Retrospective

Building iXj was a significant engineering effort for a one-person project. The iXj plug-in for Eclipse is implemented in approximately 18,600 lines of Java code.[3] The Harmonia plug-in for Eclipse that connects the Harmonia framework to Eclipse takes another 7,000 lines of code. The Harmonia framework, designed by the author of this dissertation, was a multi-person project developed over 10 years and implemented in approximately 450,000 lines of C++, C, Java, and other custom languages. Not surprisingly, the stability and the performance of all this infrastructure was essential to iXj's success.

Creating a research infrastructure is the bane of many software engineering research projects. On the one hand, the infrastructure is necessary to provide many of the support services, such as some basic source code analyses and user interfaces. On the other hand, building such an infrastructure typically involves a significant engineering effort by several researchers, while providing little opportunity for novel research. This observation clearly applies to our work on iXj and Harmonia.

[3] Not counting comments and blank lines of code.

5.3.1 Experience with the Harmonia Framework

Harmonia provided extremely flexible and powerful program analysis services. Unfortunately, it was also plagued by bugs and deficiencies dealing with which, while challenging, did not contribute to the research presented in this dissertation. The complexity of some of the algorithms, most notably incremental GLR parsing and error recovery, made it difficult to diagnose and fix defects in a timely manner. Integration with Eclipse was difficult, primarily due to the multi-threading issues and due to the complexity of the cross-language (Java/C++) implementation.

At the same time, there were many advantages to using the Harmonia analysis infrastructure. When we started the implementation of the iXj plug-in in 2002, the program analyses in Eclipse were too immature for creating the iXj program model. The Harmonia analysis and editing model reconstructs a structural representation quickly after each text edit and fits well in the iXj workflow. The ability to roll back any syntax tree to an arbitrary past version made some aspects of the iXj's implementation simpler.

Yet, to develop iXj further, we feel that we must abandon the Harmonia framework. Over the past few years, the program analysis services available on the Eclipse platform have matured sufficiently to provide a solid base for the iXj's analyses. Because Eclipse is continuously developed and widely supported, its stability and its ability to support the latest features of the Java programming language cannot be matched by a research platform.

5.3.2 Experience with Eclipse

Eclipse was instrumental to iXj's development in many ways. Most importantly it provided a stable and familiar development environment for professional Java developers that took part in our evaluation. This enabled us to concentrate only those aspects of implementation that were central to our research and not to worry about source code editing, project management, and other services expected of an integrated development environment.

At the same time, Eclipse is not without its faults. Not all aspects of its interface and internal representation were conveniently exported as public APIs. On several occasions we were forced to modify Eclipse's source code to expose the necessary functionality. iXj was limited by the user-interface concepts and metaphors available in Eclipse. For instance, we could not easily prototype some of our ideas for visualizing the effects of a transformation in the Eclipse source code editor.

Still, in retrospect, the choice of Eclipse as the implementation platform for iXj was correct. Our only wish is that more tools of the same caliber were available for research in software engineering.

5.3.3 Interactive Transformations in Other Programming Languages

Creating interactive transformation tools for a programming language requires two essential components: the existence of program analysis infrastructure and the existence of an interactive development environment. For Java, these goals were met by Harmonia and Eclipse; other languages may requires a different combination of tools.

Designing an interactive transformation tool for a new programming language comprises the following steps:

1. Understanding code-changing behavior of expert developers, including their language terminology and their perception of the source code structures. For C-like languages (such as Java), much can be borrowed from our work.

2. Designing a program model for interactive transformations and implementing its construction. Our program model can be used as the starting point for object-oriented languages similar to Java. The analysis algorithms for constructing a program model can use a standalone library (like Harmonia) or can rely on the analysis infrastructure of an IDE (like Eclipse).

3. Designing and implementing a visual transformation language. Most visual elements of the iXj design can be used for other programming languages. We developed several graphical components for Eclipse that facilitate rendering and layout of these visual elements. These components implement the "box-and-glue" algorithms and do not depend on any particular program model.

4. Implementing the user-interaction model that embodies by-example construction and immediate transformation feedback. The user-interface components that we developed for Eclipse are independent of any particular programming language and can support any visual transformation language similar to iXj. These components embody the interface concepts and metaphors found in Eclipse; other IDEs may require a redesign that fits into their user interface.

5. Implementing a pattern matching and transformation engine. For the most part, the iXj's pattern matcher and the transformation engine can be reused, though our syntax-directed pattern matching algorithm includes language-specific components that correspond to the iXj program model for Java. These components would have to be reimplemented for a different programming language.

Interactive transformations can have many uses. Most modern programming languages and most interactive development environments will benefit from having a tool for interactive program transformations available for developers' use. We hope that other researchers will follow and bring the concept of interactive program transformations to programming languages other than Java.

Chapter 6

Usability Evaluation of Interactive Transformations

In order to assess the ease of learning the visual transformation language and the usability of the iXj user-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 iXj transformations. The participants completed a short code editing task and filled out an evaluation questionnaire. This chapter presents our methodology and discusses the results of the evaluation.

6.1 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 with audio- and screen-capturing software for later analysis.

The evaluation was conducted on an IBM Thinkpad T42 laptop with 1.8GHz Pentium M processor and 1Gb of RAM. The laptop was equipped with a 15-inch high-resolution display (1400 x 1050 pixels). The participants had a choice of using a trackpoint, a touchpad, or a mouse as their pointing device. We used Camtasia Studio[1] for screen and audio capture.

6.2 Participants

The participants in our study were proficient Java programmers with various levels of experience working with Eclipse. Our main selection criteria was programming proficiency with

[1] Camtasia Studio is a commercial product available from TechSmith, Inc.

Java—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. All participants considered themselves expert Java programmers, with an average of eight years of Java programming experience. Two participants reported being novices to Eclipse and 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, such as in Swanson [61].

During the pre-study interview all participants reported regularly performing adaptive and perfective source code maintenance. 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 the 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 text-processing tools (such as the SED utility or PERL scripts), although all participants indicated that they were aware of these tools.

6.3 Training

The training session consisted of a walkthrough of the Eclipse user interface, emphasizing the interaction with our transformation environment. We demonstrated iXj on a simple set of de Morgan's law transformations, similar to those presented in Section 5.1.2. During the evaluation session the participants learned the following features of iXj and of our interaction model:

- How to construct the initial transformation pattern "by-example" from the selection in the source code editor.
- How to read pattern representation based on nested structure demarcated with graphical boxes.

- How to expand and collapse pattern structure and how the visual alignment of the pattern elements helps to see the relationship between the pattern and the source code.
- How to convert a pattern element to a wildcard and how to undo the conversion by re-invoking the wildcard handle.
- How to cycle through the optionality states of a pattern element.
- How to add a transforming action to a pattern element, both at the top-level of a pattern and at any of its sub-patterns.
- How to refer to the parts of the matched pattern using box names as pattern variables.
- How to evaluate the transformation based on the feedback shown in the source code editor and the package explorer.
- How to add a transformation to the pending list, how to individually toggle the preview of transformations on that list, and how to bring a transformation on the pending list back into the transformation editor.

6.4 Transformation Task

We asked participants to perform a code maintenance task on a console-based implementation of the MineSweeper game. The task was similar to the one used in the transformation tools case study (see Section 2.3). The existing implementation of MineSweeper relied on the `java.io.StreamTokenizer` class to process console input. The participants were asked to convert the uses of `java.io.StreamTokenizer` to the uses of `java.util.Scanner`.

The MineSweeper game consists of approximately 600 lines of Java source code divided among three classes.[2] We modified the source code to include more references to the `java.io.StreamTokenizer` objects and refactored the implementation so that the accesses to `java.io.StreamTokenizer` are divided among several methods that appear in different files. We also varied the way `java.io.StreamTokenizer` objects are referenced. For example, in one place the code calls `s.nextToken()` and in a different place it calls `st.nextToken()`. This variation forces the participants to introduce a wildcard into the pattern for locating these calls. The participants were instructed not to replace the call to the `java.io.StreamTokenizer` constructor and not to fix the declaration of a field containing a reference to `java.io.StreamTokenizer`. Typically, these tasks would be performed "by hand." The participants needed to change a total of 26 lines of code to complete the transformation. We present a listing of the relevant parts of the MineSweeper source code in Appendix B.

Because we did not expect the participants to be acquainted with the `java.io.StreamTokenizer` and `java.util.Scanner` APIs, we presented them with a listing of both APIs

[2]Source code for the MineSweeper game was obtained from `http://www.dcs.qmul.ac.uk/~mmh/ItP/resources/MineSweeper/Notes.html`

```
public class StreamTokenizer {
    // Symbolic constants for the token type
    public static final int TT_NUMBER;
    public static final int TT_WORD;
    public static final int TT_EOF;

    public int ttype;   // Type of the last token read
    public String sval; // String value of the last token read
    public double nval; // Numeric value of the last token read

    // Reads next token from the input and returns its type
    public int nextToken();

    ...
}

public class Scanner {
    public boolean hasNext();        // Is next token a word?
    public String next();            // Gets next word
    public boolean hasNextInt();     // Is next token an integer?
    public int nextInt();            // Gets next token as integer
    public Exception ioException();  // Gets last exception thrown (if any)

    ...
}
```

Figure 6.1: `java.io.StreamTokenizer` and `java.util.Scanner` interfaces that were available to the participants in the user evaluation.

#	**Before (x is StreamTokenizer)**	**After (s is Scanner)**
T1	`x.ttype == StreamTokenizer.TT_NUMBER`	`x.hasNextInt()`
T2	`x.ttype == StreamTokenizer.TT_WORD`	`x.hasNext()`
T3	`x.ttype == StreamTokenizer.TT_EOF`	`x.ioException()` `  instanceof EOFException}`
T4	`(int) x.nval`	`x.nextInt()`
T5	`x.sval`	`x.next()`
T6	`x.nextToken()`	`// no longer needed`

Figure 6.2: Transformations needed in the MineSweeper source code to convert the uses of `java.io.StreamTokenizer` to the uses of `java.util.Scanner`.

(Figure 6.1) and with a table showing sample transformations needed in the MineSweeper source code (Figure 6.2). We instructed participants that they should not interpret the transformations in the table literally. For example, `t` can stand for any expression whose type is `java.io.StreamTokenizer`, and `s` can stand for any expression whose type is `java.util.Scanner`. Likewise, the transformation T1 never appears in the MineSweeper source code verbatim. Rather, the source code contains statements like `if (t.ttype != StreamTokenizer.TT_NUMBER) {...}`, requiring the result of the transformation to include negation. The participants were told that they could construct these transformations in arbitrary order.

6.5 Metrics

During evaluation we measured time to completion for each of the transformations presented in Figure 6.2. We also recorded total time to completion of the entire task, but we did not find that metric very useful—some 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 that consisted 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 presented in Chapter 2. We have put the CDs framework to dual use: in addition to making it part of our usability evaluation, we applied the framework in the early design stages to gain additional insight into the problem. In our design, we attempted to achieve high marks along each of the dimensions. Thus, in addition to the overall usability picture, the responses to the CDs questionnaire provide feedback on our design targets. In order to make the results easily quantifiable, we augmented the traditional qualitative CDs questionnaire [6] with a seven-point semantic differential scale. We present full questionnaire in Appendix C.

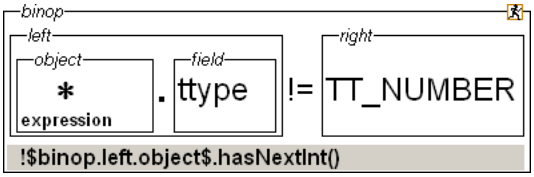
6.6 Hypotheses

The key hypothesis of our evaluation 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.

6.6.1 Expected Solutions

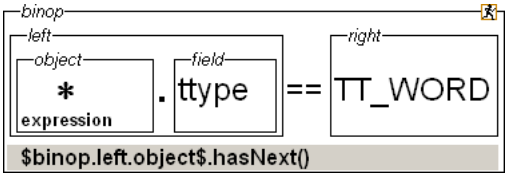
This section presents transformations that we expected our participants to construct in the course of the evaluation. We show a solution for each of the transformation tasks in Figure 6.2, presenting several alternatives, if possible.

T1: `x.ttype == TT_NUMBER` ⇒ `x.hasNextInt()`

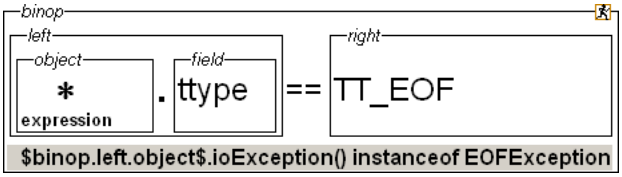


This was a tricky task because the non-negated version of this boolean expression does not appear in source code. We expected the participants to realize that and create this transformation as above.

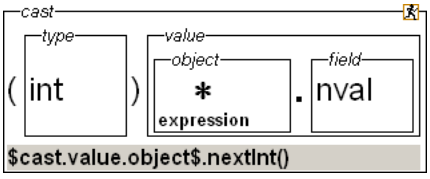
T2: `x.ttype == TT_WORD` ⇒ `x.hasNext()`



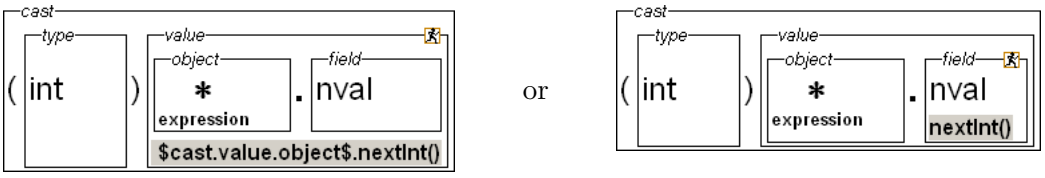
T3: `x.ttype == TT_EOF` ⇒ `x.ioException() instanceof EOFException`



T4: `(int) x.nval` ⇒ `x.nextInt()`

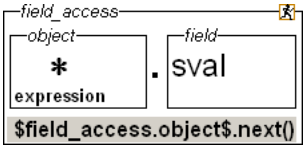


The above transformation presents the most obvious solution that replaces the entire cast expression. It is also possible to modify just the casted value. Such a transformation has two forms:

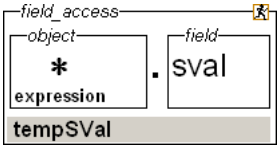


The left version replaces the entire casted value. The right version modified the casted value without using any pattern variables in the action. Both of these transformations leave unnecessary casts to `int`. These casts can be cleaned up with another transformation similar to the ones presented in Section 4.4.2.

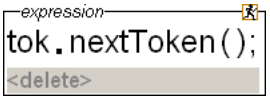
T5: `x.sval` ⇒ `x.next()`



This was another tricky task because `java.util.Scanner.next()` not only returns the next token from the scanner, but also advances the input position. This means that reading from `java.io.StreamTokenizer.sval` several times in a row without interleaving calls to `java.io.StreamTokenizer.nextToken()` is not equivalent to calling `java.util.Scanner.next()`. Such a sequence occurs in the MineSweeper game. The correct transformation is to assign the value of `java.util.Scanner.next()` to a temporary variable (such as `tempSVal`) and replace accesses to the `sval` field with a temporary variable as follows:



T6: Remove `x.nextToken()`



6.7 Results

We found the overall opinion of the participants to be very favorable. All participants were able to complete the task and were satisfied with their work. During the post-study interview the participants indicated that they enjoyed working with our tool. In this section we present the summary of participants' performance, the results of the cognitive dimensions questionnaire, and the analysis of common mistakes, errors, and misconceptions.

Transformation	Participants				
	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 6.3: Time in seconds spent by each of the participants on each of the transformations listed in Figure 6.2. **Init** represents the time spent on the initial attempt. **Fix** represents the time spent on a subsequent correction, if any. **Total** represents total time spent for a transformation. Not all participants attempted all transformations.

6.7.1 Performance

Figure 6.3 lists times (in seconds) spent by each of the participants on each of the transformations listed in Figure 6.2. 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 did not introduce appropriate wildcards into a transformation pattern on the first attempt. After realizing this, they went back to an earlier transformations from the pending transformation list to correct their mistakes. Total transformation times reflect the complexity of transformation, though there was a great amount of variation depending on the order in which the participants attempted the transformations.

Figure 6.4 presents the time (in seconds) spent by each participant on each task *in the order these tasks were performed*. (This is the same information as in Figure 6.3, reordered for easier presentation.) Transformation task times demonstrate the participants' increased fluency with the transformations they wrote later. (We also confirmed this observation subjectively when analyzing screen recordings.) For example, everyone attempted transformations T2 and T3 in sequence because these transformations occur close together in the source code. These transformations are comparable in pattern complexity and the second transformation in the sequence was always specified more quickly than the first one.

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

Figure 6.4: Time in seconds spent by each of the participants on each of the transformations listed in Figure 6.2 *arranged in the order of completing a task*. **Init** represents the time spent on the initial attempt to construct a transformation. **Fix** represents the time spent on a subsequent correction, if any.

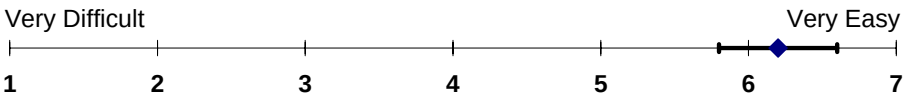
Three of the participants missed one of the transformations from the list. We attribute this to their inability to rely on the compiler to detect errors that would otherwise lead them to places in the source code still needing transformation. (The compiler was not available to them.) This limitation will be addressed in a future version of our tool.

6.7.2 Cognitive Dimensions Questionnaire Evaluation

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.

Below we present the questionnaire results for each of the twelve cognitive dimensions that we considered during evaluation. For the nine dimensions that are amenable to quantification we provide the average numeric score and the standard deviation plotted on the semantic differential scale. In addition to a numeric score, the participants had an opportunity to include a verbal description of the issues related to a particular dimension. We present a summary for each of those issues. Those comments also enabled us to evaluate the three non-quantifiable dimensions.

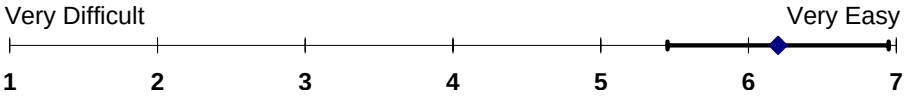
Visibility: How easy is it to see or find various parts of the transformation description while it is being constructed or changed?



In general the participants felt that visibility of the transformation description language was good. One participant expressed concern about the long name references in the transfor-

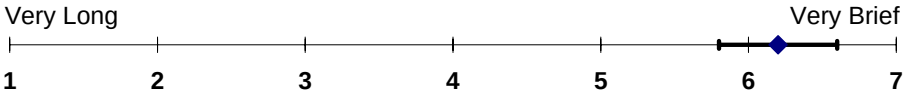
mation action, noting that they get hard to read when the name refers to a deeply nested structure, as in `$if.test.value.left.conditional.then$`. This name is required, for example, to refer to the variable `y` inside `if (!(x ? y : z)) { ... }`.

Viscosity: How easy is it to make changes to the parts of the transformation description that you completed?



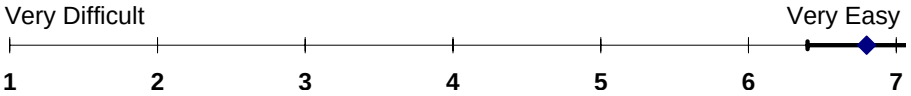
The participants felt that viscosity was low, with one participant noting that making changes was "much easier than [he] expected". Another participant appreciated the ability to make changes to the completed transformations that needed adjustment by taking them out of the pending transformation list.

Diffuseness: Does the transformation notation let you describe what you want reasonably briefly or is it long-winded?



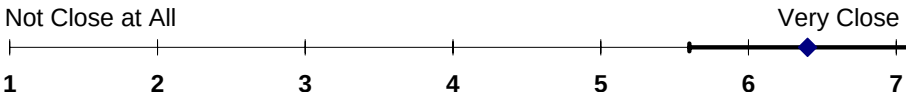
The participants reported low diffuseness, although two of the participants expressed concerns about the long name references in the transformation action. Coupled with another participant's feeling that this reduces visibility (above), this issue emerged as one of the problems that we need to address.

Role Expressiveness: When looking at the transformation description, how easy is it to tell the purpose of each part in the overall scheme?



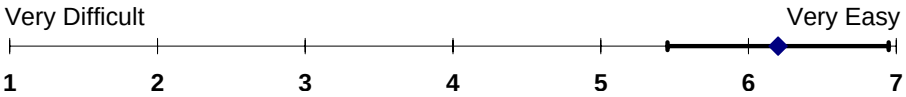
iXj received high marks for role expressiveness. The participants stated that "it is obvious where each part comes from," and thanked us for "not showing these as a tree."

Closeness of Mapping: How closely does the transformation description match your intuitive understanding of program structure?



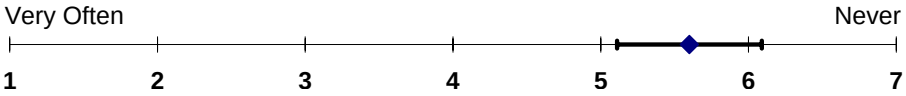
The participants reported that iXj achieves close mapping between the transformation description and their understanding of the program structure. One participant noted: "The pattern looks, visually, like source code. It makes sense to edit it as an example of the change you want to make and convert things to wildcards where they are unnecessarily specific." Another participant commended iXj for "great indication of wildcarding."

Progressive Evaluation: How easy is it to stop in the middle of creating a transformation description and check your work so far?



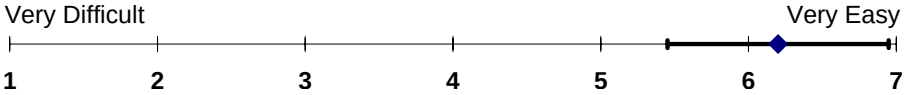
Most of the participants were satisfied with the ability to evaluate a transformation-in-progress. One participant noted that he had trouble "mak[ing] sure [he] had grabbed all matches that [he] intended." This problem was also mentioned by other participants in the post-study interview.

Error Proneness: How often do you find yourself making small slips that make the transformation process frustrating?



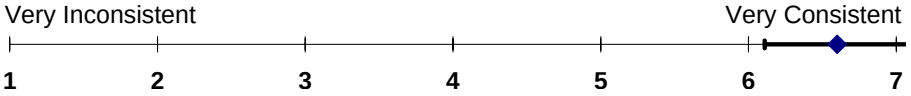
Participants' marks and comments on the error-proneness dimension confirmed some of the observations that we make in Section 6.7.3. One participant indicated that "[he] was often not sure that [he] made the pattern sufficiently generic." Two other participants noted that it is easy to mistype a variable name in the transformation action. Another participant disliked small icons for pattern box handles.

Provisionality: When working on a transformation description, how easy is it to explore various directions when you are not sure which way to go?



The participants felt that they could easily explore 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."

Consistency: How would you rate the consistency of the transformation notation?



Everyone felt that transformation notation was consistent, with one participant emphasizing that he "really liked that the pattern looks like the source code itself."

Hard mental operations: What kinds of things require the most mental effort when constructing a transformation description?

In this category all participants listed the same two problems: (1) deciding which parts of the pattern need to be "wildcarded" and (2) knowing when to stop adding wildcards.

Hidden Dependencies: Are there any parts in the transformation description that, when changed, require you to make other related changes to other parts of the description?

Most participants did not notice any hidden dependencies in the transformation description. Our subsequent analysis of the screen recordings, however, exposed two hidden dependencies that caused confusion for the participants.

Premature Commitment: When working on a transformation description can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first?

Most participants felt that they could work on a transformation in any order. 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.

6.7.3 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. This section presents the summary of our analysis.

Mistakes

Forgetting to click outside of the action editor to preview transformation. Two of the participants kept forgetting that they needed to click outside of the action editor to activate the transformation and preview its results. This shortcoming can be addressed by introducing a timeout that activates the transformation when the user stops modifying the action for a given period of time.

Forgetting to accept transformation. Some participants kept forgetting to click on the "Add to Pending Transformations" button to move a transformation to the pending transformation list. Others also found that the concept of the currently edited transformation being separate from other pending transformations confusing. We intended to redesign this part of the user-interaction model to avoid confusion.

Insufficient wildcarding. 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 problem was also mentioned under the cognitive dimension of hard mental operations. This result contrasts with our initial design intuition that the users will be able to "reason" about a transformation and will add all of the necessary wildcards based on the intent of the transformation. For example, we assumed that if the users wanted to

create a transformation that replaces a call to `nextToken()` on all instances of `java.io.StreamTokenizer`, they would select one such method call and convert the reference to a `java.io.StreamTokenizer` expression (the instance argument) to a wildcard. In practice, this was never the case: the participants wanted to introduce as few wildcards as possible.

Our best guess is that this problem was caused by the participants' concern that too general a pattern will result in more changes than they desire. We plan to fix this by implementing a more flexible pattern-matching algorithm that can produce approximate matches. The approximate matches will indicate to the user which parts of the transformation should be "wildcarded" to include more source fragments into the transformation and which variations of the pattern structure exist in the program. Such an approach reduces the need for the "upfront" wildcarding in anticipation of possible matches.

Unintended wildcarding of a pattern element. This simple slip resulted from users clicking on the wildcard handle on the wrong pattern box. This happened infrequently and we attribute this slip to the size of the icons (one of the participants mentioned this problem in his responses to the questionnaire). When the participants made this mistake, they were very pleased by the ability to toggle the wildcarding and "undo" their mistake.

Picking wrong pattern element for replacement. This mistake occurred when the participants wanted to attach a transforming action to one of the nested pattern elements. In two cases they were confused about the appropriate level at which the transforming action needs to be specified. We believe that this problem can be addressed through better visualization of which part of the source code is affected by an action.

Errors

Missed inversion of the method call result. Three of the participants missed variations between the source code and the transformations described in the API mapping table. They mistakenly replaced `s.ttype != StreamTokenizer.TT_NUMBER` with `s.hasNextInt()`, not realizing that the result of the method call needed to be inverted. This represents a conceptual error that renders the transformed program incorrect. After introducing such an error, the only possibilities of discovering it are through testing or through code inspection. We believe that this error is largely due to the participants being unfamiliar with the `java.io.StreamTokenizer` and the `java.util.Scanner` interfaces and following the list of transformations from the supplied table. In practice, the users are likely to have better understanding of the transformation task because they would be the ones formulating it. Still, the possibility of such errors is a concern.

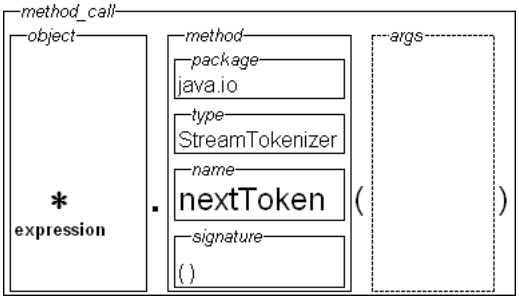
Ignoring method call side-effects. Two participants introduced a conceptual error in the program when replacing `t.sval` with `s.next()` (transformation T5). The method `java.util.Scanner.next()` has a side-effect of advancing the current position in the input stream. But the `doCommand()` method in the MineSweeper source code repeatedly examines the current input token as follows:

```
private void doCommand() throws IOException {
    if (t.sval.equals("reveal")) {
        ...
    } else if (t.sval.equals("mark")) {
        ...
    } else if (t.sval.equals("unmark")) {
        ...
    }
}
```

Replacing the accesses to the `sval` field with calls to `next()` is erroneous. Two of the participants made that mistake. Again, we attribute this error to the participants' unfamiliarity with the `java.util.Scanner`'s behavior.

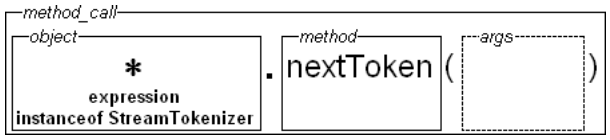
Misconceptions

Confusion about type constraints on the expression wildcard. Several participants expressed hesitation prior to converting the instance argument expression to a wildcard. For example, when generalizing `getTokenizer().nextToken()` to `*.nextToken()`, it was not clear to them that this pattern will only match calls to `nextToken()` when the instance argument is an instance of `java.io.StreamTokenizer`. In iXj patterns this restriction is implied by the name scoping rules. `nextToken` is a method name that refers to a method in `java.io.StreamTokenizer`. Thus, the expression appearing as its instance argument can only be of the `java.io.StreamTokenizer` type. This can be seen by expanding the pattern elements representing `nextToken`:

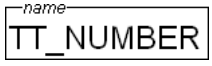


This information, however, is not shown in the pattern editor unless the user "drills down" into the `nextToken` pattern element. When this point was clarified, the participants seemed more comfortable with wildcarding.

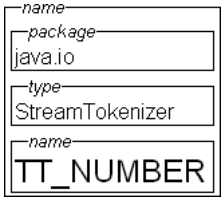
This problem represents an example of a hidden dependency, although the participants did not list it as such on the cognitive dimensions questionnaire. We intend to implement a solution to this problem by automatically generating a type constraint dictated by the expression context when the wildcard is created. For example:



Confusion about name representation. One of the participants encountered a situation in which the automatically created transformation pattern did not look like the source code fragment that he selected to create that pattern. This confusion occurred when the participant selected a reference to a static field `java.io.StreamTokenizer.TT_NUMBER`. This selection produces a pattern that looks as follows:



Because in the source code the `java.io.StreamTokenizer` part of that name exists solely for the purpose of qualifying the reference to `TT_NUMBER`, in the pattern that information became hidden inside the fully-qualified representation of that name. `TT_NUMBER`'s association with `java.io.StreamTokenizer` can be seen only when the user "drills down" into the `TT_NUMBER` pattern element:



The participant's ensuing confusion illustrates another hidden dependency in the transformation pattern language. This is an example where our pursuit of consistency resulted in a design error that led to a hidden dependency. We will address this problem by rethinking this part of our design.

Wanting to wildcard an operator. When working on the transformation T1, four participants wondered if they could convert the token representing the inequality operator into a wildcard. Their intent was to match both `t.type == StreamTokenizer.TT_NUMBER` and `t.type != StreamTokenizer.TT_NUMBER`. We do not currently support wildcarding of individual keywords and tokens in iXj. It is possible to extend the language to add this feature, but it is not clear that it would be a worthwhile addition—those participants that wanted to wildcard an operator subsequently realized that that would not be appropriate for their transformation.

Confusion about operator precedence. Two of the participants expressed concern when working on the transformation that introduces the `instanceof` operator (transformation T3). They felt insecure about introducing an operator into the replacement string without

considering the expression context in which that string will appear in source code. If other operators of higher evaluation precedence are present in that expression, the evaluation order of that expression can change. This is a valid concern. Consider the following example:

```
if (!s.atEndOfFile()) {
    ...
}
```

Suppose a transformation calls for replacing `s.atEndOfFile()` with `s.ioException() instanceof EOFException`. (This is a slightly modified version of transformation T3.) A naive transformation produces:

```
if (!s.ioException() instanceof EOFException) {
    ...
}
```

This code sequence fails to compile because the negation operator (`!`) has higher precedence (binds more tightly) than the `instanceof` operator. As a result, the negation becomes associated with `s.ioException()`, which is not a boolean expression.

There are two ways to fix this problem. The first option is to require the developers to parenthesize the transformation action whenever there is a possibility of causing precedence errors. This would constitute a hard mental operation. The second option is to enhance the transformation engine to analyze the context in which a replacement expression is inserted to ensure that the evaluation order is unchanged. We intend to implement this mechanism in a future version of iXj.

6.8 Discussion and Observations

We found the results of our evaluation to be very encouraging. We were pleased to confirm many of our design decisions and glad to uncover some design issues that can be addressed in a future version of the transformation tool.

The evaluation confirmed our hypotheses. The participants were able to learn iXj and the transformation editor quickly and were able to complete the transformation task successfully. The participants found the pattern notation understandable and easy to use. The participants found the interaction model with the transformation editor to be intuitive. Several participants expressed interest in using the transformation tool in their daily coding and maintenance activities.

We found that our evaluation strategy was limited in several respects. First, the transformation task presented to the participants was fairly simple and was executed on a small source code base. These restrictions were needed to ensure that the evaluation session can be completed within reasonable time without tiring the participants. At the same time, this created a somewhat unrealistic setting. In practice, we expect that the developers will be

performing a much wider range of transformation tasks and will be working on their own code bases. A more thorough evaluation of iXj should include a longitudinal case study of developers using our tool on their own source code for an extended period of time.

6.8.1 Cognitive Dimensions of Notations as Evaluation Strategy

The Cognitive Dimensions of Notations presented a useful framework for the post-study discussion. In particular, the framework provided the necessary lexicon that enabled participants to talk about features of the transformation language and the transformation editor. The CDs questionnaire motivated participants to think consciously about many issues that they would otherwise not consider. Because we have used the CDs framework to guide our design, we found that the numeric scoring that we introduced to the CDs questionnaire provided good feedback on our design targets.

Still, on several occasions the participants weren't sure what aspect of our system we had in mind when we asked them about a particular dimension. We attribute this confusion to our insufficient attention to the concepts of *notation*, *environment*, and *medium*. Blackwell and Green distinguish notation, environment, and medium as follows.

> "The notation comprises the perceived marks or symbols that are combined to build an information structure. The environment contains the operations or tools for manipulating those marks. The notation is imposed upon a medium, which may be persistent, like paper, or evanescent, like sound." [5]

When we applied the CDs framework to iXj, we considered the transformation description language as the notation, the transformation editor as the environment and the computer screen as the medium. Yet, we failed to distinguish it from a second notational layer formed by the *interaction language*—a series of commands that the user uses to manipulate the information structure. Just like the transformation notation, the interaction language also has syntax and semantics embodied in our user-interaction model. This secondary layer should be analyzed separately, but we did not consider this difference in our evaluation.

Furthermore, our evaluation was hampered by the participants' limited exposure to the system. Because the participants only worked on a few hand-selected transformation tasks, 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 hidden dependencies and hard mental operations.

When our transformation tool is made available to a broader population of software developers and when these developers use our system more extensively, it would be beneficial to perform another round of cognitive dimensions evaluation.

6.8.2 Implications for Developer Productivity

We computed the average time to complete a transformation to be 107 seconds. This average is across all participants and all transformation tasks. Because time to complete a transformation varies greatly with the developer proficiency and the complexity of a task,

we cannot claim that *most* transformations can be completed in 107 seconds. We can, however, use this metric to judge the efficacy of interactive transformations in improving developer productivity.

The keystroke level model (KLM) [14] is a technique for computing the time to perform a task using keyboard and mouse. KLM is one of the family of related techniques included in GOMS (Goals, Operators, Methods, Selection Rules) [41]. KLM has been used to model expert performance in text editing [13] and program entry [72]. A KLM is constructed using operators that express the time spent on individual actions taken by the user. The operators of interest to us include clicking a mouse button (B–0.2 sec.), typing a character (K–0.28 sec.), moving hand to mouse or keyboard (H–0.40 sec.), mental preparation (M–1.35 sec.), and pointing with a mouse (P–1.10 sec.). Mental preparation is added at the start of a task and whenever the user enters any user-defined value. These times are taken from Card, et al. [13] and are based on empirical observations.

Consider the transformation T5: `(int) t.nval` ⇒ `s.nextInt()`. We computed the average time spent by the participants on this transformation to be 102 seconds. Let us assume that the developer is using compiler errors to guide him to the locations in source requiring change. (In Eclipse, compilation is incremental and the errors appear in a separate pane below the editor window as the program is edited.) The sequence of operations to perform one change can be modeled as follows: (1) mental preparation–M, (2) move hand to mouse–H, (3) point with mouse at an error–P, (4) click mouse button–B, (5) move hand to keyboard–H, (6) three keystrokes to delete old text, assuming word-at-a-time deletion–K, (7) mental preparation–M, (8) eleven keystrokes to type the new text–K. Expressing this as a formula for predicted time yields $T = 2M + 2H + P + B + 14K = 8.72$ seconds. This indicates that after only twelve changes (102 sec. / 8.72 sec.) the time is spent on creating a transformation is completely recovered. When developers change source code manually there is also significant opportunity for introducing bugs and compilation errors. Fixing those problems adds to the time developers spend on a change.

In addition to the immediate productivity benefits, the reduction in code changing effort and the increased developer confidence enabled by iXj can lessen the developer's resistance to making design-improving changes. This can lead to improved developer productivity in the long term, and, ultimately, to higher quality software.

6.8.3 Using iXj for Source Code Maintenance

The sample transformation task presented in this chapter essentially repeats the source code maintenance case study discussed in Section 2.3 with iXj. One of the key conceptual differences between iXj transformations and those afforded by SED, TXL, and the refactoring tools is the generality of the resulting transformations. As evidenced by our evaluation, the users are not keen on creating a fully general solution. Thus, the resulting iXj transformations may lack sufficient wildcards to be reusable in a broader context. We believe, however, that this does not present a problem. Thanks to the iXj's iterative approach to transformation development, any set of transformations deemed sufficient for a given task and stored in a transformation library, can be generalized further when the need arises.

This makes development of iXj's transformations simpler, because the developer need not anticipate future uses of their transformations. In contrast, the refactoring transformations, for example, must be more general and more broadly applicable.

The transformations expressed in the visual transformation languages are more concise and more readable than solutions discussed in Chapter 2. While direct comparison is difficult, we can observe that all transformations needed for the sample task fit on a single printed page. This is in contrast to the TXL transformations for the same task that occupy four printed pages (see Appendix A).

Clearly there are TXL and refactoring transformations that are not expressible in iXj. Notably, those transformations that require control- and data-flow information, such as the *Extract Method* refactoring (Fowler [31], p. 110), are not supported. One of the challenges in designing iXj was the need to balance the expressiveness of the language with the ability of developers to understand and manipulate artifacts in that language. Our design represents a compromise between these two requirements and, as we believe, it successfully addresses its purpose of simplifying mundane and tedious source code editing operations.

Chapter 7

Conclusion

Making large and sweeping changes to source code can be a tedious and error-prone process, requiring the developer to perform many systematic and menial source code edits. Our work investigates the use of source-to-source program transformations in an interactive setting as a potential solution for automating systematic source code editing. The thesis of our research is that developers can use formal transformations of program source code effectively and that the use of these transformations will reduce the effort expended on mundane and time-consuming code editing tasks. This dissertation proves our thesis.

7.1 Contributions of This Research

The main goal of the work presented in this dissertation is to make the well-known concepts behind formal program transformations accessible to developers for lightweight source code manipulation. In pursuing this goal, we made several research contributions.

Program Model for Java Source Code Manipulation. In developing the concept of interactive program transformations we designed a new program model for Java source code that is more natural and understandable for human developers than existing tool-centric approaches. Our program model facilitates visual presentation of the program structure. It enables manipulation of program source code using entities and relationships that "make sense" to a typical software developer. The model only supports structural representation of program source code when necessary, falling back to text-based representation where the structure is not needed. This program model is embodied in a visual language for transformation of program source code.

Visual Transformation Language. We designed a visual language for representing structural transformation patterns. This language, called iXj, combines textual and graphical elements. iXj helps developers to visualize program structure and enables them to manipulate structures that do not have concrete text-based representation, such as sequences. The transformation language includes facilities to represent syntactic wildcards, patterns

involving type and scope information, and non-structural text-based patterns. The transformation language enables developers to associate transforming behavior with any structural element in the pattern.

The design of the transformation language was influenced by the Cognitive Dimensions (CDs) framework. Using this framework we designed a notation that exhibits high visibility, low viscosity (resistance to change), and excellent closeness of mapping to the developers' mental model of source code. We evaluated our transformation language as part of the usability evaluation of the iXj prototype.

User-Interaction Model for Constructing Transformations. Creating a structural pattern representation in a visual language can be challenging even in an interactive environment. We designed a user-interaction model that guides the developer through the construction of a transformation. Our interaction model is built upon two key principles—by-example construction and iterative refinement. By-example construction enables developers to start with a single instance of a transformation that is automatically generated by the transformation tool and to generalize that transformation to apply more broadly. Iterative refinement of transformations, guided by the feedback from the transformation editor, helps the developers to generalize the transformation pattern and to specify the transforming action.

The design of the user-interaction model was also influenced by the CDs framework. Our model facilitates progressive evaluation and provisionality (the ability to explore various potential directions). At the same time, the user-interaction model strives to eliminate hidden dependencies and to reduce premature commitment. We evaluated the interaction model as part of the usability evaluation of the iXj prototype.

Interactive Transformation Tool for Eclipse. Our implementation of iXj as a plug-in for Eclipse constitutes a technical contribution of this dissertation. The resulting tool is fully-integrated with the code editing workflow in Eclipse and affords the developers access to lightweight, as-needed transformations. Our implementation builds upon our previous work on Harmonia, a framework for constructing interactive language-based programming tools.

This implementation enabled us to evaluate the understandability of the program model, the visual transformation language, and the user-interaction model with professional software developers. We expect to contribute this implementation to the open-source community and to refine the existing prototype to the point where it can be used by developers for their day-to-day maintenance tasks (see Section 7.2).

Task-centered Design for Software Development Tools. One of the indirect contributions of this work is the validation of task-centered design as a viable and important methodology for building software development tools. Software development tools are unique in that their designers, developers, and users are often the same people. As a result, it is unusual for development tool designers to employ any user-centric design process. In our

work, the task-centered design workflow was instrumental in creating a novel notation for describing transformations and in devising a user-interaction model for manipulating that notation. We benefited greatly from the iterative evaluation prescribed by task-centered design. We introduced a new informal evaluation strategy into the methodology of task-centered design. This strategy, based on the Cognitive Dimensions framework, helped us refine the design and contributed to the user evaluation of our implementation. We conclude, from our experience, that task-centered design is an important technique for creating new tools for software developers.

7.2 Future Work

The work presented in this dissertation provides fertile ground for future investigation. This section summarizes several possible directions.

7.2.1 Engineering Challenges

As part of this dissertation we have built a prototype implementation of an interactive program transformation plug-in for Eclipse. Creating an implementation that is sufficiently stable for user evaluation was a significant engineering effort. Yet, we feel that we only partially succeeded in meeting the engineering challenges—further work is needed to turn the prototype into a usable product.

The primary focus in developing iXj further needs to be on refining the prototype to the point that it can be used by developers for day-to-day code-maintenance activities. Stability, performance, and scalability—all require additional attention. One of the options that we are currently exploring is to re-implement the program analysis components using Eclipse's own analysis engine. This will reduce the amount of research code in the plug-in and improve some aspects of the implementation.

Furthermore, the evaluation prototype is missing some of the essential features that are necessary for a fully-functional product. Some examples include:

- *The ability to save a constructed transformation set.* One of the participants in the evaluation inquired whether he would be able to reuse his `java.io.StreamTokenizer` to `java.util.Scanner` transformations on other pieces of code that he might come across. Doing so requires the ability to preserve transformations in an off-line form.

- *The ability to use iXj code patterns for searching source code.* This is often a pre-requisite step for building a transformation. For example, two participants in the evaluation wanted to find all uses of the `java.io.StreamTokenizer` class prior to beginning their work on the transformations. While this type of search is supported in Eclipse, they wanted to use iXj patterns to specify these searches.

- *The ability to retract transformations after they are applied to source code.* Having constructed all of the necessary transformations, several participants in our evaluation hesitated before hitting the "Apply" button. They inquired whether it is possible to

"undo" the application and go back to editing transformations, in case something did not behave as expected.

In order to realize these engineering goals, we intend to release the current iXj prototype to the open-source community. In addition to attracting new developers to our project, we hope to achieve broader penetration of the ideas underlying iXj and to introduce more software developers to the concept of interactive transformation of source code.

7.2.2 Open Research Issues

We learned a lot by constructing and evaluating the current iXj prototype and we were pleased that our evaluation participants found iXj a useful extension to their current workflow. We expect to gain additional insight when iXj is further developed and more widely deployed, allowing developers to use iXj for realistic maintenance tasks on their own code. Yet, we feel that further research can make interactive transformations even more broadly applicable. Below, we outline several possible directions.

Control- and data-flow information in the program model. Our structural representation of program source code incorporates typing and scoping rules. Some transformations, such as those needed for refactoring, require information about data and control dependencies in source code. These dependencies are not currently exposed as part of the program model. The challenge lies in incorporating this information in a way that makes it easily understandable.

More sophisticated pattern-matching algorithms. The current pattern-matching algorithm used by the transformation engine is fairly simplistic. For example, it does not support *non-linear* patterns that permit binding two parts of the pattern to the same pattern variable. Non-linear patterns restrict those parts to match the same structure—a feature we expect to be necessary for describing some transformations. The performance of the algorithm can be improved by implementing a more sophisticated tree-pattern matcher using a finite-state automaton. Further improvements can be achieved through *pre-matching*—scanning the text-based representation of source code to determine if token sequences occurring in the pattern are present in the source code text. This technique can eliminate computationally expensive tree-pattern matching in many situations.

Application of interactive program transformations beyond code editing. Transformations can be construed broadly. In addition to replacing existing code, transformations can also generate and insert new code fragments based on linguistic structure or on meta-information embedded in program source code. These types of transformations can be supported by iXj by extending the action language.

Support for more programming languages. We designed the iXj transformation language specifically to support transformation of Java programs. The underlying concepts of our design can also be extended to other programming languages. The challenge lies in devising a program model for those languages that is sufficiently expressive to support broad range of transformations and sufficiently simple for presentation to developers.

7.3 Final Summary

In this dissertation we introduce a novel approach to program manipulation using interactive transformations. We present iXj, a language for interactively transforming Java programs, and a plug-in for Eclipse that enables developers to construct source code editing transformations using this language. iXj provides a novel visual notation for representing source code patterns and demonstrates how a carefully designed user-interaction model enables smooth integration of the transformation process with a typical software development workflow. By enabling developers to manipulate source code with lightweight language-based program transformations, iXj reduces the effort expended on making certain types of large and sweeping changes. In addition to making developers more efficient, this reduction in effort can lessen developers' resistance to making design-improving changes, ultimately leading to higher quality software.

Bibliography

[1] Greg J. Badros. JavaML: a markup language for Java source code. *WWW9/Computer Networks*, 33(1–6):159–177, June 2000.

[2] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. ACM Press, 1990.

[3] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *International Conference on Software Engineering*, pages 625–634, 2004.

[4] Alan Blackwell. SWYN: A visual representation for regular expressions. In Henry Lieberman, editor, *Your Wish Is My Command*. Morgan Kauffman, 2001.

[5] Alan F. Blackwell and Thomas R. G. Green. Cognitive dimensions of information artefacts: a tutorial, 1998. `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf`.

[6] Alan F. Blackwell and Thomas R. G. Green. A cognitive dimensions questionnaire optimised for users. In In A.F. Blackwell and E. Bilotta, editors, *PPIG 13*, May 2000.

[7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley Professional, 1998.

[8] Marat Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical Report CSD-01-1149, Berkeley, CA, 2001.

[9] Marat Boshernitsan. Program manipulation via interactive transformations. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 392–393, New York, NY, USA, 2003. ACM Press.

[10] Marat Boshernitsan and Susan L. Graham. Designing an XML-based exchange format for harmonia. In *Working Conference on Reverse Engineering*, pages 287–289, 2000.

[11] Marat Boshernitsan and Susan L. Graham. iXj: interactive source-to-source transformations for java. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN*

conference on Object-oriented programming systems, languages, and applications, pages 212–213, New York, NY, USA, 2004. ACM Press.

[12] Scott Burson, Gordon B. Kotik, and Lawrence Z. Markosian. A program transformation approach to automating software reengineering. In *Proceedings of the 14th Annual International Computer Software and Applications Conference*, pages 314–322. IEEE Computer Society Press, 1990.

[13] Stuart K. Card, Thomas P. Moran, and Allen Newel. *The Psychology of Human-Computer Interaction*. Erlbaum, Hillsdale, NJ, 1983.

[14] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactice systems. *Commun. ACM*, 23(7):396–410, 1980.

[15] Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. JATS: A Java transformation system. *Brazilian Symposium on Software Engineering*, 2001.

[16] Hock Chan, Keng Siau, and Kwok-Kee Wei. The effect of data model, system and task characteristics on user query performance: an empirical study. *SIGMIS Database*, 29(1):31–49, 1997.

[17] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[18] Steven Clarke. Evaluating a new programming language. In G. Kadoda, editor, *PPIG 13*, May 2001.

[19] Don Coleman, Joel Confino, Peter Koletzke, Brian McCallister, Tom Purcell, and John Shepard. Java IDE shootout. `http://developers.sun.com/learning/javaoneonline/2004/corej2se/BUS-2864.pdf`, 2004.

[20] Michael L. Collard, Jonathan I. Maletic, and Andrian Marcus. Supporting document and data views of source code. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 34–41, New York, NY, USA, 2002. ACM Press.

[21] James R. Cordy, Ian H. Carmichael, and Russell Halliday. *The TXL Programming Language: Version 10.4*, 2005. `http://txl.ca/docs/TXL104ProgLang.pdf`.

[22] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Software engineering by source transformation-experience with TXL. In *Source Code Analysis and Manipulation*, pages 170–180. IEEE Computer Society, 2001.

[23] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. *Comput. Lang*, 16(1):97–107, 1991.

[24] Microsoft Corp. Visual Studio. `http://msdn.microsoft.com/vstudio/`.

[25] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile parsing in TXL. *Autom. Softw. Eng*, 10(4):311–336, 2003.

[26] Francoise Detienne. *Software Design - Cognitive Aspects*. Springer Verlag, 2001.

[27] Andrea A. diSessa and Hal Abelson. Boxer: a reconstructible computational medium. *Commun. ACM*, 29(9):859–868, 1986.

[28] Dale Dougherty. *sed & awk*. O'Reilly & Associates, Inc., 1991.

[29] Eclipse.org. Eclipse platform: Technical overview, 2003. `http://eclipse.org/whitepapers/eclipse-overview.pdf`.

[30] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimothy. Towards a standard schema for C/C++. In *Working Conference on Reverse Engineering*, pages 49–58. IEEE Computer Society Press, October 2001.

[31] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, June 1999.

[32] Thomas Genssler and Volker Kuttruff. Source-to-source transformation in the large. In *JMLC*, pages 254–265, 2003.

[33] Thomas R. G. Green. Cognitive dimensions of notations. In *Proceedings of the HCI'89 Conference on People and Computers V*, Cognitive Ergonomics, pages 443–460, 1989.

[34] Thomas R. G. Green. Instructions and descriptions: some cognitive aspects of programming and similar activities. In *Advanced Visual Interfaces*, pages 21–28, 2000.

[35] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Vis. Lang. Comput*, 7(2):131–174, 1996.

[36] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In A. Cimitile and H. A. Müller, editors, *Proceedings: Fourth Workshop on Program Comprehension*. IEEE Computer Society Press, 1996.

[37] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[38] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Toward a standard exchange format. In *Working Conference on Reverse Engineering*, pages 162–171, 2000.

[39] JetBrains. IntelliJ IDEA. `http://www.intellij.com/idea/`.

[40] JetBrains. ReSharper. `http://www.intellij.com/resharper/`.

[41] Bonnie E. John and David Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(4):320–351, December 1996.

[42] Brian Johnson and Ben Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 284–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[43] Donald E. Knuth. The errors of TEX. *Software– Practice and Experience*, 19(7):607–681, July 1989.

[44] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, November 1981.

[45] Maria Kutar. A comparison of empirical study and cognitive dimensions analysis in the evaluation of UML diagrams. In J. Kuljis, L. Baldwin, and R. Scoble, editors, *PPIG 14*, June 2002.

[46] David A. Ladd and J. Christopher Ramming. A∗: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, November 1995.

[47] Clayton Lewis and John Rieman. *Task-Centered User Interface Design*. Shareware, 1994. `http://hcibib.org/tcuid/`.

[48] Henry Lieberman. *Your Wish Is My Command — Programming by Example*. Morgan Kaufmann, 2001.

[49] Martin Lippert. Towards a proper integration of large refactorings in agile software development. In *XP*, pages 113–122, 2004.

[50] Robert C. Martin and Robert S. Koss. Engineer notebook: An extreme programming episode. In Robert C. Martin, editor, *Advanced Principles, Patterns and Process of Software Development*. Prentice Hall, 2001.

[51] Sun Microsystems. *Java Native Interface*, 1997. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html`.

[52] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.

[53] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[54] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[55] Emmanuel Pietriga, Jean-Yves Vion-Dury, and Vincent Quint. VXT: A visual approach to XML transformations. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document engineering*, pages 1–10, New York, NY, USA, 2001. ACM Press.

[56] Peter G. Polson, Clayton Lewis, John Rieman, and Cathleen Wharton. Cognitive walkthroughs: A method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36(5):741–773, 1992.

[57] GNU Project. GNU Emacs. `http://www.gnu.org/software/emacs/`.

[58] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[59] Derek M. Shimozawa and James R. Cordy. TETE: A non-invasive unit testing framework for source transformation. In *STEP 2005: 12th International Workshop on Software Technology and Engineering Practice*, 2005.

[60] Omnicore Software. JustCode! `http://www.omnicore.com/justcode.htm`.

[61] E. Burton Swanson. The dimensions of maintenance. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[62] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8(1):89–120, 2001.

[63] Michael L. Van De Vanter. Practical language-based editing for software engineers. *Lecture Notes in Computer Science*, 896, 1995.

[64] Michael L. Van De Vanter. The documentary structure of source code. *Information & Software Technology*, 44(13):767–782, 2002.

[65] Michael L. Van De Vanter and Marat Boshernitsan. Displaying and editing source code in software engineering environments. In *Proceedings of Second International Symposium on Constructing Software Engineering Tools*, pages 39–48, Limerick, Ireland, 2000.

[66] Eelco 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.

[67] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. Ph.D. dissertation, University of California, Berkeley, March 11, 1998. Technical Report UCB/CSD-97-946.

[68] Tim A. Wagner and Susan L. Graham. Efficient self-versioning documents. In *Proceedings of 42nd IEEE International Computer Conference*, San Jose, CA, 1997.

[69] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–43, 1997.

[70] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly and Assoc., 2000.

[71] Richard C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, August 1988.

[72] Marian G. Williams and J. Nicholas Buehler. A study of program entry time predictions for application-specific visual and textual languages. In *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, pages 209–223. ACM Press, 1997.

[73] Ben Wing. ChangeLog entry for 2002-05-05. The XEmacs ChangeLog, 2002. `http://cvs.xemacs.org/viewcvs.cgi/XEmacs/xemacs-20/src/ChangeLog`.

[74] The XEmacs Project. XEmacs: the next generation of Emacs. `http://www.xemacs.org/`.

[75] Xrefactory. A C/C++ refactoring browser for Emacs and XEmacs. `http://www.xref.sk/xrefactory`.

Appendix A

Complete TXL Program for the Transformation Case Study

This appendix presents a full listing of the TXL program used in the source code manipulation case study in Chapter 2. This program implements the transformations specified in Figure 2.2. We refer the reader to the TXL programming language manual [21] for assistance in interpreting this TXL program.

```
include "Java.Grm"
include "JavaCommentOverrides.Grm"

function main
   replace [program]
        P [program]
   by
        P [transformTokenizerToScanner]
          [transformNextInt]
          [transformNextInt2]
          [transformNextDouble]
          [transformNextDouble2]
          [transformNext]
          [transformNext2]
          [transformNumberTest]
          [transformNumberTest2]
          [transformWordTest]
          [transformWordTest2]
          [removeNextTokenStatement]
          [removeNextTokenStatement2]
end function
```

```
rule transformTokenizerToScanner
   replace [qualified_name]
        java.io.StreamTokenizer
   by
        java.util.Scanner
end rule

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

rule transformNextInt2
   replace $ [expression]
        (int) (E [expression]) C [repeat component]
   by
        (E) C [transformNextIntInComponent]
end rule

rule transformNextIntInComponent
   replace $ [repeat component]
        .nval
   by
        .nextInt()
end rule

rule transformNextDouble
   replace $ [expression]
        E [id] C [repeat component]
   by
        E C [transformNextDoubleInComponent]
end rule

rule transformNextDouble2
   replace $ [expression]
        (E [expression]) C [repeat component]
   by
        (E) C [transformNextDoubleInComponent]
end rule

rule transformNextDoubleInComponent
```

```
   replace $ [repeat component]
        .nval
   by
        .nextDouble()
end rule

rule transformNext
   replace $ [expression]
        E [id] C [repeat component]
   by
        E C [transformNextInComponent]
end rule

rule transformNext2
   replace $ [expression]
        (E [expression]) C [repeat component]
   by
        (E) C [transformNextInComponent]
end rule

rule transformNextInComponent
   replace $ [repeat component]
        .sval
   by
        .next()
end rule

rule transformNumberTest
   replace [expression]
      E [id] C [repeat component] == TT_NUMBER
   by
      E C [transformNumberTestInComponentNI] ||
      E C [transformNumberTestInComponentND]
end rule

rule transformNumberTest2
   replace [expression]
      (E [expression]) C [repeat component] == TT_NUMBER
   by
      (E) C [transformNumberTestInComponentNI] ||
      (E) C [transformNumberTestInComponentND]
end rule
```

```
rule transformNumberTestInComponentNI
   replace [repeat component]
        .ttype
   by
        .hasNextInt()
end rule

rule transformNumberTestInComponentND
   replace [repeat component]
        .ttype
   by
        .hasNextDouble()
end rule

rule transformWordTest
   replace [expression]
      E [id] C [repeat component] == TT_WORD
   by
      E C [transformWordTestInComponent]
end rule

rule transformWordTest2
   replace [expression]
      (E [expression]) C [repeat component] == TT_WORD
   by
      (E) C [transformWordTestInComponent]
end rule

rule transformWordTestInComponent
   replace [repeat component]
        .ttype
   by
        .hasNext()
end rule

rule removeNextTokenStatement
   replace [statement]
        E [id] C [repeat component] ;
   where
        C [isNextToken]
   by
        ; % none
end rule
```

```
rule removeNextTokenStatement2
   replace [statement]
        (E [expression]) C [repeat component] ;
   where
        C [isNextToken]
   by
        ; % none
end rule

function isNextToken
   match [repeat component]
        .nextToken()
end function
```

Appendix B

Partial Source Code Listing for the MineSweeper Game

This appendix presents a partial listing of the source code for the MineSweeper game that we used for user evaluation. We only list those parts of the source code that were affected by the transformations in Figure 6.2. The source code for the MineSweeper game was originally obtained from `http://www.dcs.qmul.ac.uk/~mmh/ItP/resources/MineSweeper/Notes.html` and modified to present more opportunities for transformation.

```
public StreamTokenizer getTokenizer() {
    return tokenizer;
}

public void play() throws IOException {
    while (!done) {
        ...
        getTokenizer().nextToken();
        if (getTokenizer().ttype == StreamTokenizer.TT_WORD) {
            doCommand();
        } else if (getTokenizer().ttype == StreamTokenizer.TT_EOF) {
            done = quit = true;
        } else {
            System.out.println("Unknown command -- try 'help'");
        }
        ...
    }
}

private void doCommand() throws IOException {
    if (getTokenizer().sval.equals("reveal")) {
```

```
        ...
    } else if (getTokenizer().sval.equals("mark")) {
        ...
    } else if (getTokenizer().sval.equals("unmark")) {
        ...
    } else if (getTokenizer().sval.equals("help")) {
        ...
    } else if (getTokenizer().sval.equals("quit")) {
        ...
    } else {
        System.out.println("Unknown command -- try 'help'");
    }
}

public boolean doMark(MineSweeper sweeper) throws IOException {
    StreamTokenizer t = sweeper.getTokenizer();
    t.nextToken();
    if (t.ttype != StreamTokenizer.TT_NUMBER)
        throw new IllegalArgumentException();
    int x = (int) t.nval;
    t.nextToken();
    if (t.ttype != StreamTokenizer.TT_NUMBER)
        throw new IllegalArgumentException();
    int y = (int) t.nval;
    ...
}

public int doReveal(MineSweeper sweeper) throws IOException {
    StreamTokenizer tok = sweeper.getTokenizer();
    tok.nextToken();
    if (tok.ttype != StreamTokenizer.TT_NUMBER)
        throw new IllegalArgumentException();
    int x = (int) tok.nval;
    tok.nextToken();
    if (tok.ttype != StreamTokenizer.TT_NUMBER)
        throw new IllegalArgumentException();
    int y = (int) tok.nval;
    ...
}

public boolean doUnmark(MineSweeper sweeper) throws IOException {
    StreamTokenizer st = sweeper.getTokenizer();
    st.nextToken();
```

```
    if (st.ttype != StreamTokenizer.TT_NUMBER)
        throw new IllegalArgumentException();
    int x = (int) st.nval;
    st.nextToken();
    if (st.ttype != StreamTokenizer.TT_NUMBER)
        throw new IllegalArgumentException();
    int y = (int) st.nval;
    ...
}
```

Appendix C

Cognitive Dimensions Questionnaire for User Evaluation

This appendix presents a questionnaire that we constructed for evaluating iXj using the Cognitive Dimensions framework (see Chapters 2 and 6). The language for the questionnaire was adapted from the *Cognitive Dimensions questionnaire optimized for users* [6]. We augmented the Blackwell and Green questionnaire with a seven-point semantic differential scale in order to produce more easily quantifiable measures.

Evaluation of an Interactive Transformation Tool

```
We would like to collect your views about the ease of use of our
interactive transformation tool.  The questionnaire includes a series
of questions that encourage you to think about the ways you use this
tool, and whether it helps you to do the things you need.


1. How easy is it to see or find various parts of the transformation
   description while it is being constructed or changed?

   Very Difficult                                      Very Easy
        1       2       3       4       5       6       7

a. What kinds of things are easier to see or find?

b. What kinds of things are more difficult to see or find? Why?

2. How easy is it to make changes to the parts of the transformation
   description that you completed?
```

Very Difficult Very Easy
1 2 3 4 5 6 7

a. What kinds of changes are easy to make?

b. Which changes are more difficult or especially difficult to make? Why?

3. Does the transformation notation let you describe what you want reasonably briefly or is it long-winded?

Very Long Very Brief
1 2 3 4 5 6 7

a. Which parts of the description seem particularly compact?

b. What sorts of things take more space to describe? Why?

4. When looking at the transformation description, how easy is it to tell the purpose of each part in the overall scheme?

Very Difficult Very Easy
1 2 3 4 5 6 7

a. Which parts are obvious and easy to interpret? Why?

b. Which parts are particularly difficult to interpret? Why?

c. Are there any parts that you don't really understand, but you put them in because they just seem to be required? What are they?

5. How closely does the transformation description match your intuitive understanding of program structure?

Not Close at All Very Close
1 2 3 4 5 6 7

a. Which part of the description seems to be a fairly natural way of describing something?

b. Which part seems to be a particularly strange way of doing or describing something? Why?

6. How easy is it to stop in the middle of creating a transformation description and check your work so far?

   Very Difficult Very Easy
   1 2 3 4 5 6 7

a. Can you do this at any time you like? If not, why not?

b. Can you find out how much progress you have made and check at what stage you are in your work? If not, what prevents you from doing so?

7. How often do you find yourself making small slips that make the transformation process frustrating?

   Never Very Often
   1 2 3 4 5 6 7

a. What are the mistakes that seem particularly common or easy to make?

8. When working on a transformation description, how easy is it to explore various directions when you are not sure which way to go?

   Very Difficult Very Easy
   1 2 3 4 5 6 7

a. What features of the transformation tool were most helpful in doing so?

b. What features that would help you experimenting are missing from the transformation tool?

9. A notation is consistent when parts of the notation that mean similar things appear similar on the screen and the parts that mean different things appear different on the screen. How would you rate the consistency of the transformation notation?

   Very Inconsistent Very Consistent
   1 2 3 4 5 6 7

a. What are the examples of things that ought to be different, but appear similar?

b. What are the places where some things ought to be similar, but the notation makes them different?

10. What kinds of things require the most mental effort when constructing a transformation description?

11. Are there any parts in the transformation description that, when changed, require you to make other related changes to other parts of the description? What are they?

a. Are all of the dependencies between these parts visible? What dependencies are hidden?

b. In what ways can it get worse if you are creating a large transformation description?

12. When working on a transformation description can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first?

a. If so, what decisions do you need to make in advance? What sorts of problems can this cause in your work?