1. INTRODUCTION
Program source code is bound to change over the course of its lifetime. Be it due to design improvements, bug fixes, or addition of new features, the source code constantly evolves from its original version.

Unfortunately, changing software source code is often tedious and error-prone. The process is complicated because a conceptually simple change may entail pervasive large-scale modifications to the entire body of source code. Examples of such changes abound in the many maintenance tasks faced by developers. For instance, consider the following steps for adding a new internal status code to a software system.

1. **Add a new constant representing the new value.** This step is the simplest of all and is quickly performed “by hand.”
2. **Update every location where the status value is computed to take the new code into account.** This is difficult and tedious, requiring a change to every place where the status value is returned by a routine.
3. **Update every location where the status value is checked to handle the new code.** A time-consuming, yet important task — missing one such check may jeopardize the stability.
4. **Update the debugging routine that translates status codes into a human-readable form.** While this is another trivial step, it can be easily overlooked by the developers if the debugging code is not frequently used.

Various proposals have been made for systematic modification to existing source code. However, few tools have found their way to the "programming trenches.” Our research attacks several major issues with prior approaches: generality, acceptance by the user community, improved abstraction management, and proliferation of proprietary extensions to the existing programming languages.

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

2. INTERACTIVE TRANSFORMATIONS
Our approach it to enable software developers to express code changing operations using concise user-friendly notation. To facilitate use of this notation, we will also provide interactive support within developer’s programming environment.

Following the established terminology, such code changing operations are called *transformations*. Transformations can be construed broadly. In addition to replacing existing code, transformations can also generate new code fragments.

2.1 The Human Factor
Many existing tools support specification and execution of transformations on program source code. (Several such systems are discussed in Section 3.) However, these tools are intended for expert use on large and complex tasks. By contrast, our system is oriented toward *end-programmers* — the end-users of traditional development environments. We draw this distinction to differentiate end-programmers from *language tool experts*. Language tool experts are those who understand the structure of program source code from the perspective of tools, such as compilers. Tool experts may be comfortable thinking about source code in terms of syntax trees and other linguistic data structures. We do not expect end-programmers to possess this knowledge.

Devising a transformation system for end-programmers is especially challenging. One the one hand, transformations are highly structural and are closely tied to the syntax and semantics of the programming language. On the other hand, we cannot expect our users to understand the transformation tool’s internal representation for syntactic and semantic data.

Nevertheless, end-programmers’ understanding of program source is based on its structure. This is supported both by the experimental results in the field of psychology of programming (Detienne 2001) and by our own empirical observations of developer expression. When describing source code to one another, programmers say things like:

“Put p := link(p) into the loop of show_token_list, so that it doesn’t loop forever.” (Knuth 1989)

“Change BI.* macros to BYTE_.* for increased clarity.” (Wing 2002)
Programmers evoke notions such as variables, expressions, statements, loops, and assignments. They directly refer to names found in source code. They use patterns to describe large classes of similar changes. Inspired by these kinds of examples, we can design a formal language for source code transformations.

2.2 Language for Source Code Manipulation

In order to enable developers to describe transformations using familiar concepts, our language for source code manipulation must be targeted toward a particular programming language. We chose Java as our initial testbed due to its widespread use in both academic and industrial environments. The transformation language that we designed is called iXj, for “Interactive Transformation for Java.” While iXj is a language tightly coupled with Java, we expect that the design methodology employed for iXj can be extended to other programming languages. In fact, in future we envision an entire family of iX languages such as iXc (for C), iXxml (for ML), etc.

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

Armed with this knowledge, we designed the first version of iXj. The core of the iXj language is based on the selection/action programming model. A selection is a query that describes a set of Java source code fragments. One or more actions are bound to each selection and specify some code changing operation to be applied to every selected fragment.

Selection is achieved by combining queries over the declarative structure of Java programs (packages, classes, methods) with structural pattern matching below the declaration level. For instance, to update the computation of status value in the second step of our example, the user describes how to locate all methods for the Employee class. The environment will generate a selection query that matches only that expression as well as a stub for the selected expression.

Our tool will generate a selection query by introducing wildcards into the generated specification. While the generated code may have no trouble specifying the control structure of pattern matching and transformations in a textual notation. At the same time, the transformation environment augments iXj with direct manipulation. Selection patterns can be created “by-example,” whereby the user selects a source fragment that represented a single matching instance and then abstracts the generated pattern to match a larger class of code fragments.

For instance, the workflow for creating an iXj transformation in the second step of our example could be as follows:

1. **Use the mouse to select one of the expressions in the program that calculates a status code.** The environment will assist the user in selecting a syntactically complete structure.
2. **Ask the environment to generate a transformation template for the selected expression.** Our tool will generate a selection query that matches only that expression as well as a stub transforming action that replaces the expression with itself.
3. **Generalize the selection query by introducing wildcards into the generated specification.** While the generated code may be freely edited, the system also provides a wizard-like assistant for those unfamiliar with the iXj notation. Immediate feedback on matching locations informs the user whether the selection query is sufficiently general.
4. **Rewrite the transforming action to insert an expression that computes the new status code.** As with the selection query, a context-sensitive assistant can guide the user through the unfamiliar process. The result of the transformation can be previewed before the action is executed.

The transformation environment can also capture the source code change history in terms of high-level transforming operations. Such a capability helps to document important aspects of program evolution, as well as supports selective rollback of high-level changes days, months, and even years after they had been performed.

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

2.4 Evaluation
The research described in this overview will be evaluated against several criteria. First, we will evaluate the iXj notation and the transformation environment using the Cognitive Dimensions framework (Green & Petre 1996). This framework offers a set of techniques for a broad-brush usability analysis of information artifacts and can provide quick feedback on a partially implemented design.

Our second benchmark will involve using iXj to specify some commonly used transformations, such as those catalogued by Fowler (1999) in his book “Refactoring: Improving the Design of Existing Code.” Initially, we do not expect all refactorings to be expressible in iXj since it lacks some of the more sophisticated program analysis infrastructure required for these transformations. Yet, the ability to specify some of the well-known refactoring transformations will serve as a testament to iXj’s expressiveness and will show whether iXj provides a possible implementation platform for authors of the refactoring tools.

Finally, we will conduct traditional usability studies of the transformation environment prototype. The system will be instrumented to collect information about how it is used. We will study users carrying out a fixed set of modification tasks with and without our tools, and will compare such factors as modification time, user-perceived ease or difficulty of the task, and quality of the resulting transformations.

2.5 Implementation Details
The interactive transformation environment described in this overview is being prototyped using the Eclipse platform (Eclipse 2004), an open-source framework for building interactive development tools. Using Eclipse not only provides a stable and well-featured interactive programming environment, but also allows us to conduct user evaluations using a “real” IDE that can be utilized for day-to-day programming activities. Additionally, the Eclipse’s Java development tools include integrated refactoring facilities, which will serve as a baseline for evaluation of our interactive transformation capabilities.

In order to support sophisticated program analysis required for implementing source-to-source transformations, Eclipse has been integrated with the Harmonia framework (Boshernitsan 2001). The framework includes an incremental GLR parser, a static semantic analysis engine, and other language-based facilities. Program source code is represented by annotated abstract syntax trees augmented with non-linguistic information such as whitespace and comments. The analysis engine can support any textual language that has formal syntactic and semantic specifications.

In addition to augmenting Eclipse with fine-grained analysis services for Java, the Harmonia framework enables us to implement sophisticated language-based support for the iXj programs described in Section 2.3.

3. RELATED WORK
Using automated assistance to change program source code is hardly a new idea. Source-to-source transformations, refactoring, and metaprogramming have all been proposed as techniques that can both alleviate the tedium of editing source code by hand and reduce (and, in case of refactoring, even eliminate) the errors that such editing might introduce.

The simplest (and, perhaps, the oldest) transformation tool is the Unix SED utility, which offers pattern substitution facilities based on regular expressions over character strings. More sophisticated tools such as LSME (Murphy & Notkin 1995) and TLex (Kearns 1991) operate on the lexical structure of the program. Many developers use simple text-based tools akin to SED in their day-to-day work. However, text-oriented tools are not sufficiently powerful to manipulate syntactic or semantic structure of source code and do not offer any integrated support for constructing transformations.

To enable more complex manipulations, several tools expose internal structure of program source code structure as annotated syntax trees. Notable examples include ASTLog (Crew 1997), TAWK (Griswold et al. 1996), A* (Ladd and Ramming 1995), and the IP environment (Simonyi 1995). Other systems, such as REFINER (Burson et al. 1990) and TXL (Cordy et al. 1988), go even further by enabling the user to provide a description of the source code structure that is suited to the transformation task. Though powerful, these tools are aimed at language tool experts, such as suppliers of canned transformations, and are difficult to master for end-programmers wanting to automate mundane editing tasks.

It has been suggested that end-programmers require no more than a well-defined set of expert-provided canned transformations to support common code manipulation tasks. Indeed, the recent advent of support for refactoring transformations (Opdyke 1992) in popular development environments appears to support this hypothesis. However, refactorings do not provide a fully general approach to source code manipulation. Frequently, a large-scale editing operation, such as the update to the computation of a status value in our transformation example, falls outside of what can be expressed with canned refactoring transformations.

Metaprogramming tools represent another class of source code manipulation systems. Rather than exposing linguistic structure of program source code (however abstract), these systems present source code entities through a data model that reflects their semantic meaning. Semantic macro processors such as XL (Maddox 1989), OpenJava (Tatsubori et al. 2000), OpenC++ (Chiba 1995), extend the underlying programming language and provide the ability to implement such extension by transforming the data model. More generic metaprogramming tools such as InjectJ (Genssler & Kuttruff 2003) enable creation of standalone transformation scripts in a new programming language. Yet, none of these facilities were developed with end-programmer in mind and require a significant amount of effort even for simple code changing operations. Moreover, lack of integrated environment support makes these tools ill-suited for interactive use.

Finally, some of the problems we pose can be solved using aspect-oriented programming tools, such as AspectJ (Kiczales et al. 1997). Aspect tools allow a small set of canned compile-time or run-time transformations (primarily conditional code insertion) at well-defined points in program execution. However, they are
limited in power and scope, are language-specific, and are
designed for compiler-like use: the result of the transformation is
not exposed to the developer and cannot be further manipulated.

4. RESULTS AND CONTRIBUTIONS

The research presented in this overview contributes to the existing
body of work on program transformations in several important
ways. First, our empirical studies of programmers will teach us
how developers express high-level changes and improve our
understanding of the mental model utilized when thinking about
such changes. Second, we are developing a novel notation, iXj,
for describing such changes for the Java programming language.
We anticipate that the methodology employed in designing iXj
will be easily transferable to programming languages other than
Java. This methodology is another important contribution of this
research.

Finally, this research provides a fertile ground for future
exploration of user-accessible program transformation services.
For instance, it remains to be seen whether the transformation
language can be made even more expressive, without sacrificing
its usability. Can it be extended to permit specification of
invariants to implement meaning-preserving transformations?
Will it be useful in specifying simple code auditing analyses (e.g.,
ensuring that all variables follow a naming convention)? Is it
possible to permit specification of modular language extensions,
much like those supported by IP? All these questions can be
explored further upon completion of the research presented here.

5. CONCLUSION

This overview presents the problem of, and a solution for
performing large-scale systematic modifications on a large body
of source code. We combine the results from psychology of
programming, software visualization, program analysis, and
program transformation fields to create a novel environment that
lets the programmers express operations on program source code
at a level above text-oriented editing using source-to-source
program transformations.

The design space for a source code transformation system is large
and complex. Many systems for source code transformations have
been developed in the past and many will be built in the years to
come. The interactive transformation environment presented in
this overview represents a particular point in this vast design
space, which we believe strikes a good balance between
expressiveness, usability, and versatility for day-to-day coding
tasks faced by the developers in the “programming trenches.”

6. REFERENCES

for Constructing Interactive Language-Based Programming Tools.
M.S. Report, EECS, Computer Science Division, University of
California, Berkeley, UCB/CSD-01-1149

Transformation Approach to Automating Software Re-
Engineering. Proceedings of the 9th International Computer
Software & Application Conference.

the 1995 Conference on Object-Oriented Programming, Systems,

Rapid Prototyping System for Programming Language Dialects.
Proceedings of the International Conference on Computer
Languages, pp. 280-285.

syntax trees. Proceedings of the 1st Conference on Domain
Specific Languages, pp. 229—242.

Springer Verlag.

http://eclipse.org/

Code. Addison Wesley.

Genssler, T. & Kuttruff, V. (2003). Source-To-Source

Programming Environments: A 'Cognitive Dimensions'
Framework. Journal of Visual Languages and Computing,
7(2):131—174.

Flexible Syntactic Pattern Matching and Processing. Proceedings
of the IEEE 1996 Workshop on Program Comprehension.

21(8):805—821.

Proceedings of the European Conference on Object-Oriented
Programming (ECOOP). Springer-Verlag LNCS 1241.

Experience, 19(7):607—685.

Implementing Language Processors. IEEE Transactions on
Software Engineering, 21(11).

Master's thesis, University of California, Berkeley. UCB/CSD
89/545.

eXtraction. Proceedings of the 3rd ACM SIGSOFT Symposium on
the Foundations of Software Engineering.

PhD thesis, Department of Computer Science, University of
Illinois at Urbana-Champaign, Urbana, Illinois.

Simonyi, C. (1995). The Death of Computer Languages, the Birth
of Intentional Programming. NATO Science Committee
Conference.

OpenJava: A Class-based Macro System for Java. Reflection and
Software Engineering, vol. 1826, Lecture Notes in Computer
Science, W. Cazzola, R. Stroud, and F. Tisato, Eds.: Springer-
Verlag, pp. 117-133.

XEmacs ChangeLog: http://cvs.xemacs.org/viewcvs/cgi/
/XEmacs/xemacs-20/src/Changelog