A Survey of Refactoring Tool Researches

Zhiying (Vicky) Wang

Student ID: 200050143
Table of Contents

1. Introduction ............................................................................................................................................. 3
2. Refactorings .............................................................................................................................................. 4
   2.1 Refactoring definition ............................................................................................................................. 4
   2.2 Properties of refactoring .......................................................................................................................... 4
3. Requirements for Refactoring Tools .......................................................................................................... 5
   3.1 Practical Criteria ...................................................................................................................................... 5
   3.2 Technique criteria ..................................................................................................................................... 5
4. Implementation Issues of Refactoring Tools .............................................................................................. 6
   4.1 Program Representations ........................................................................................................................ 7
   4.2 Refactoring transformations .................................................................................................................... 8
   4.3 Internal Representation For a Refactoring Tool ..................................................................................... 8
     4.3.1 CFG or PDG ..................................................................................................................................... 8
     4.3.2 Multiple Representations ................................................................................................................ 10
     4.3.2 Abstract Syntax Tree ....................................................................................................................... 10
   4.4 Refactoring Process ............................................................................................................................... 11
   4.5 Proof of Behavior Preserving ................................................................................................................ 11
5. Survey on Refactoring Tools ...................................................................................................................... 12
   5.1 Existing Refactoring Tools ................................................................................................................... 12
   5.2 Refactoring Browser for Smalltalk ......................................................................................................... 13
6. Discuss on Automating Refactorings ......................................................................................................... 14
7. Conclusion .................................................................................................................................................. 15
Appendix: ..................................................................................................................................................... 16
Bibliography ................................................................................................................................................... 18
1. Introduction

Refactoring is a technique that transforms a program to improve its internal structure, design, simplicity, understandability, flexibility or other features without affect its external behavior. Initially developed in the early 1990s [Opd 92], refactoring has become a key topic in the context of reengineering object-oriented applications [Tic 01] and as part of the mainstream software development process practice [Fow 99].

Refactoring provides a new view of object-oriented software development process: programs are designed to satisfy immediate need, and future changes can be done later if they are really needed. That is: encompass adjusting of design to accommodate the changes of requirements and features by applying refactoring. For example, one of basic principles of eXtreme Programming is just apply refactoring continually, as a fundamental part of the software development process [Bec99].

Refactoring also finds its application in the context of reengineering, it provide the ability to transform software quickly and safely. However, integrating refactorings in reengineering environments offers much more than this straightforward application provided suitable tools are available. Not only can tools analyze software or apply standalone refactorings, combining of both capabilities can allow a tool to detect problems and propose solutions to resolve such problems and perform the required transformations [Tic 01].

Although it is beneficial, refactoring is not widely applied in practice as it might be. One of the largest barriers to refactoring code has been the woeful lack of tool support for it [Fow99]. Refactoring is risky since it may introduce subtle defects into program, it is also tedious, time-consuming and error prone if performed manually. Automated refactoring tools must be supplied to support fast and safe refactoring.

This paper will address the following questions:
• Which problems need to be solved in order to develop a practical refactoring tool?
• How to provide automated support for refactoring?
• What program representations should be used for refactoring tool?
• What are steps a refactoring tool should follow to perform refactoring transformations?

This paper addresses all the above questions by surveying researches on automated refactoring tools in both forward engineering and reengineering context. Instead of identifying which tools can perform which kinds of refactorings, this survey aims at understanding the implementation of refactoring tools by analyzing existing refactoring tools on the basis of publications.
2. Refactorings

2.1 Refactoring definition

Martin Fowler [Fow 99] defines a refactoring (noun) as:

- a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

He also defines to refactor (verb) as:

- to restructure software by applying a series of refactorings without changing its observable behavior.

Compare with definition of Restructuring given by Chikofsky and Cross [CC 90]:

- Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the system's external behavior.

Refactoring seems to be just a special case of Restructuring, namely within object-oriented context and focused on the code level.

Furthermore, refactorings can be associated with a catalog of widely discussed, interactive, perhaps tool-supported changes, which are usually many repeatedly applied small changes, and explicit unit tests applied before and after each minimal change.

2.2 Properties of refactoring

Refactoring is not rewriting but a kind of reorganization. Typical goals of refactoring are to improve the simplicity, understandability, flexibility or performance [Bec99].

The main property of a refactoring is the preserving the external behavior. It doesn't matter if refactoring is used to make a program simpler, or more flexible, or just cleans up the code as long as the program functionality does not change. This property sets the requirements and constrains for refactoring tools.

Many refactoring relies largely on the communications of OO programming, i.e. the implicit indirectness of message sending and the substitutability of inheritance. For example, refactoring of modifying class hierarchy, creating subclasses and superclasses to get a shorter code in the system. Refactoring occurs at different levels, which include high-level refactorings that are major design changes, and low-level refactorings such as renaming a variable. High-level refactorings can be implemented in terms of several low-level refactorings [Opd92]. If we can implement the low-level refactorings correctly, then the high-level refactorings will be correct.

Most refactorings introduce into program more layers of indirection, decoupling, and information hiding. Refactoring has a major objective, getting each piece of functionality to exist in exactly one place in the software, and thus each routine performs only one
logical functionality [Fow99]. This complies with the principle of good OO design. Once that has been achieved, if there is something wrong with a particular method, there's only one method need to rewrite. Thus continuous refactoring makes most rewriting unnecessary.

3. Requirements for Refactoring Tools

This section will identify those criteria that make a refactoring tool useful based on summary of some researches. A practical refactoring system would have some requirements.

3.1 Practical Criteria

Speed.
The main purpose of a refactoring tool is to allow programmers to refactor code easily and quickly, so the tool must be both fast and reliable to be useful. If programmers don't gain from using the tool, they will perform it manually instead of using the automatic tools. Ideally, a software tool provides the user with the experience of immediacy [Ung97]. The distance in time, space or meaning between a user's action and the tool's corresponding reaction should be small enough for the user to recognize causality.

Undo
A refactoring tool should allow user to push the code around and see how it looks under the new design. If the user don't like the code after the application of a refactoring, it should be easy to undo a refactoring. Automatic refactoring should allow an exploratory approach to design, some researchers observed the necessity of an undo operation in tools that are used in an exploratory manner [BG97]. We need a refactoring tool to have build in support for undoing refactorings.

Interactivity
A refactoring tool should offer easy way to get feedback from the user to control the execution of a refactoring, give the user interactive control over when and where individual transformations are performed, rather than blindly restructuring when possible.

Integration in IDEs
A stand-alone refactoring tool will has very few users due to the inconvenience of using it. Integration directly into IDE will make much difference.

3.2 Technique criteria

Accuracy and Correctness

The application of a refactoring should always preserve the external behavior of code. [Fow99] The only way to verify the external behavior of software is applying tests, therefore, testing is an important part of the process of refactoring. When refactoring
manually, the programmer must run the test suite after each small change to ensure that the program behaves the same as before the refactoring. The technical requirements for a refactoring tool ensures that the test suite will not have to be rerun after every little change [Rob 99]. Correct, automatic refactorings can reduce the need for test step, and can speed up the refactoring process significantly. But still, refactoring tools must offer build in support for applying tests.

Recent discussions about integrating refactoring into a commercial development process emphasis the importance of having a test suite before performing any refactoring, whether manual or automatic [Fow99]

The refactoring tool must verify that program behavior will remain unchanged, or make any needed compensating changes. Verification involves checking syntactic and semantic program properties that affect the refactoring transformations. If the checks fail, the tool should disallow the transformation, perhaps displaying reasons for the failure to the user.

**Capability of handling large amounts of code**

A refactoring tool must be able to handle large amounts of source code. Scalability is an important property for any software tool architecture. To be scalable, the internal data structures of the tool must grow (approximately) linear with the size of the program being manipulated. Without this property, a tool will always be severely limited to the size of programs on which it can operate, despite of future improvements in computing power.

**Code representations**

Since refactorings are behavior preserving program-restructuring operations, a practical refactoring system must provide a program representation that is rich enough to support precondition checking and perform consistent updates. This point will be discussed in detail in next section.

### 4. Implementation Issues of Refactoring Tools

This section discusses some of the issues that would arise in implementing a practical refactoring tool.

The main obstacle to building practical refactoring tools that satisfy the criteria presented in above section is the lack of efficient way of obtaining, storing and updating global semantic information about a large system while it is being modified.

Refactoring is source-to-source program transformations that improve program internal structure without changing its behavior. One of the first requirements for a refactoring tool is the ability to search for various program entities across the entire program, to get program information necessary performing refactoring transformation. A suitable technique to get as rich and accurate as possible program information, at the same time
balance the space and time cost for the tool being practical, need to be identified for the refactoring tool.

4.1 Program Representations

This section identifies the suitable program representation to capture program information necessary for automated analysis and for performing refactoring transformation.

There exists a range of techniques that can be used to capture program information. At one end are fast, textual tools such as “grep”, and at the other end are complicated analysis techniques such as dependency graphs. Somewhere in the middle is syntactic analysis using abstract syntax tree. There are tradeoffs between speed, accuracy, and richness of information when deciding which technique to use.

A textual representation of program source is not rich enough to support refactoring [Opd 92]. Consider a refactoring that changes the name of a member function. This refactoring need change the function definition and all places where that function is called. Depending on the access control mode of the member function, it may be referenced in its containing class, its subclasses and other places that reference an instance of one of above classes. This refactoring must change the function name in all these places. A simple textual scan and replacement approach cannot fulfill this task and is considered too unsafe for refactoring environment since this change requires analysis of program syntax and type information that textual representation doesn’t contain. For example, a name replacement should not be made as it may be used somewhere else in the program to refer to a variable or a class, distinguishing this case need program syntax analysis. Or, another unrelated class may define a different function with the same name, distinguishing this case requires type information.

There are three intermediate program representations that are the internal representations of choice for many types of software tools, they are intermediate between the source and target languages in the context of a compiler [Aho86].

- **Abstract Syntax Tree (AST)** [Aho86]. A data structure representing something has been parsed, often used as a compiler or interpreter's internal representation of a program while it is being optimized and from which code generation is performed. AST is closest to the source code, and may be easily annotated to allow regeneration of the original source code.

- **Control Flow Graph (CFG)** [Aho86]. Abstracts away the grammatical features of the program source. The CFG consists of nodes that represent the computations in a program connected by edges showing the flow of control from node to node. It can be constructed directly from an AST or during parsing, and serves as the basis for many data flow analysis algorithms.
• **Program Dependence Graph (PDG).** The PDG is a more abstract semantic program representation and shows only the control and data dependences between program operations. Constructing a PDG requires both control flow information (e.g. a CFG) and data flow analysis. The PDG is useful for compiler optimizations, [FOW87] program slicing [Ott84] and integrating multiple program versions [Hor89].

### 4.2 Refactoring transformations

Refactoring transformations explicitly apply to source code. There is similarity and difference between refactoring and optimizing compiler, which also performs program transformations. Both of them require a detailed understanding of program semantics and keep certain program behavior invariant. However, these two types of transformations differ in both their goals and the program representations that they modify. A refactoring changes the source code to improve program properties concerned by the software engineer such as understandability, maintainability, modularity, or reusability. A compiler optimization affects the performance of the machine code, though it is often applied to some intermediate program representation. [Mor97]

Most refactorings need to manipulate parts of the system that are below the method level. Often they are references to program elements that are being changed. Consider a refactoring that renaming an instance variable, all references within the methods of that class and its subclasses must be updated. Some refactorings are entirely below the method level, such as extracting a portion of a function into its own separate function.

### 4.3 Internal Representation For a Refactoring Tool

Refactorings need to manipulate program at or below method level, this entail the statement level program analysis and the method structure manipulation, which requires ASTs [Rob99]. But AST doesn’t offer information on the types of variables, the declaration of a variable, references etc, which is necessary for refactoring transformations. A refactoring tool should annotate the abstract syntax tree with type and reference information. [Opd92]. Refactoring specifications shouldn't concern with type checking or referencing.

Now another problem arises, is AST annotated with type and reference information sufficient for performing program transformation that specified by refactorings?

#### 4.3.1 CFG or PDG

Many low level refactoring transformations, such as renaming identifiers or restructuring class hierarchies, require only limited semantic information [Opd92]. However, any transformation that reorders computations requires information about control and data flow properties of the program being manipulated [Mor97]. High-level object oriented refactorings involving class invariants also require data flow analysis [Opd92].
The program flow analysis required by refactoring tools is often global in nature, and must be applied on a much larger scale than what is used in a compiler. This analysis is simply too costly using traditional approaches adopted by compiler.

front end of compiler

source code

preprocessed source

abstract syntax tree

control flow graph

program dependence graph

target program

back end of compiler

Figure 4.1 Idealized Compiler Representation Pipeline Model

Above figure shows the idealized compiler representation pipeline model, but this model doesn’t apply to software tools for understanding and manipulating source code such as refactoring tools. Since the steady abstraction of the program from one representation to the next always involves a loss of some of the information contained in previous representations. CFG or PDG are ideal intermediate representations for a compiler but not for a source level tool. Information unimportant to optimization is abstracted away and only those properties essential to compilation remain. Thus CFG and PDG contain little information needed by a source level tool.

Dependency information is useful in refactoring transformation, but it often takes a significant amount of time to compute. Many refactorings need to perform lots of small step modifications frequently, and updating PDG is very expensive. Although an approach that incrementally update the PDG after each transformation is suggested to take advantage that, in most cases each step introduces only minor modifications to the source program, it is difficult to perform. And there is another problem, the difficulty of reconstructing source code from PDG, which will be addressed later.

Since refactoring is source-to-source program transformation, both the input and output of refactoring tools are source code, which closely matches the AST. A refactoring tool may need information in a CFG or PDG, but it must relate this information to the right location in the source code. A tool that does not alter the program could maintain a one way mapping back to some previous representation. Like a compiler saves source file and line number information for debugging purposes, a program slicer applied this approach
But again, this does not apply to refactoring tools, which perform program transformations.

Besides, re-deriving source code from CFG and PDG is very difficult. For example, the PDG has been used as the basis of an algorithm for merging program versions [Yan92, Hor89]. However, the PDG only stores dependences between computationally related program components, ignoring explicit statement order [Hor89]. To reconstitute the merged program as an AST, all operations must first be ordered. Unfortunately, this ordering step is NP complete, and may also fail if the PDG is not feasible (i.e., does not correspond to any likely program). Even this step is successful, the resulting output may not look like the original source code due to possibly different statement ordering and missing formatting and comments. One of the primary difficulties of source-to-source transformations is the preservation of non-semantic information about the program text (e.g., comments, white space, formatting, etc.) [Opd92]. Similarly, generating source code directly from the CFG also has difficulties. A standard CFG lacks information about scopes, declarations, and structured control flow statements (while, for, do, switch), and only a very simple version of the program source can be get from the CFG.

4.3.2 Multiple Representations

Since none of above discussed data structures explicitly contains all the information required by refactoring transformation, a meaning-preserving program restructuring tool used hybrid internal program representations [Gri95]. However, maintaining the consistency of multiple program representations in a program manipulation tool is difficult. That tool uses all three representations mentioned above, the AST, CFG, and PDG, together with two-way mappings to interrelate them. The mapping becomes challenging when the program is being modified because all representations and the mappings between them must be updated.

Using multiple representations in software manipulation tool has several serious shortcomings although they provide necessary semantic information. For example, redundant information is kept with greater time and space cost; mapping functions are required to relate items in one representation to items in another, which may be too complicate and memory space consuming; keeping all the representations synchronized is very computationally expensive, especially for tools that change the code frequently and are intended to be used as a interactive tool. Using incremental update can help solve this problem, but it is complex and costly to implement. [Gri91, Gri93] Therefore such tools are difficult to design and implement due to these complexities.

4.3.2 Abstract Syntax Tree

Fortunately, researches suggested that a single, simple program representation - AST, contains sufficient information to implement powerful refactorings and are created extremely quickly [Rob99]. Research suggests that AST is the data structure most appropriate for use in a source to source tool architecture [Mor97]. Morgenthaler proposed a demand driven approach to efficiently extract the information contained in CFGs and PDGs from ASTs rather than having to create the additional graphs [Mor97].
This answered the question of how to obtain the other types of required program information not available in the AST, eliminated the need to create and maintain these expensive data structure, therefore greatly simplified program modification.

By annotating the nodes of the AST with formatting information, the original source code may be recovered. Thus, An abstract syntax tree, annotated with type and reference information, is a representation that can be used in a refactoring tool. Simple tool input and output make the AST a good choice for an internal refactoring tool representation. Opdyke's refactory [47], prototype tool Cstructure [Mor97], Refactoring Browser [Rob99] and many other refactoring tools all use the AST as internal representation.

4.4 Refactoring Process

This section describes refactoring process that a refactoring tool may follow. [Opd92] Program source must initially be converted to an abstract syntax tree representation before refactoring, and then the updated program source must be regenerated from the representation after refactoring. Figure 1 in appendix shows the stages in the refactoring process. Within a refactoring system, a parser creates a AST representation of the source code. The parse trees are passed to the refactorer, which first analyzes the trees and adds type and crossreference information. Then manipulates the representation according to commands from the user. Finally, a pretty printer is invoked to print the refactored code.

4.5 Proof of Behavior Preserving

Ideally, the behavior preserving of refactorings should be proven formally, but this generally has not been done in practice and in previous research. There are two reasons identified by [Lan01]: First, to automate refactorings requires a significant engineering effort to build compilers that allow AST manipulations of programs and that have convenient metaprogramming facilities for code generation. Second, proving that refactorings are behavior preserving requires a formal semantics for the target language to be defined.

Instead of formal proofs, Opdyke proposed a set of seven invariants to preserve behavior for refactorings [Opd92]. Other researchers adopted the similar approach [Rob99, Kup00, Tic01, Lan01] Opdyke’s refactorings were achieved by proofs that the preconditions that he identified for each refactoring ensure the invariants. Preconditions are assertions that a program must satisfy for the refactoring to be applied. Opdyke did not prove that preserving these invariants guarantee preserving program behavior.

For example, Opdyke’s first invariant is that each class must have a unique superclass and its superclass must not also be one of its subclasses [Opd92]. When a refactoring is possible of violating an invariant, preconditions are added to guarantee that the invariant is preserved.
Thus, the practical approach that researchers have adopted is to define for each refactoring a set of preconditions that if true would ensure behavior preservation. However, due to the complexity of the languages studied, these conditions may not be sufficient. In fact, [Lan01] describes that experiments show Opdyke’s invariants are insufficient for C++ and gave counter examples—i.e., additional conditions—that must be considered for behavior preservation. Until main-stream languages become simple enough for language semantics to be defined formally and for proofs of behavior preservation to be practical, the approach proposed by Opdyke and taken by many other researchers is likely to dominate.

5. Survey on Refactoring Tools

Refactorings are well suited for automatic tool support although refactoring tools are still in their early days. There are generally two types of refactoring tools:

- Tools that actually perform program transformations once the programmer has identified a candidate and decide to refactor. Nearly all known tools fall in this category. For example, the original refactoring power tool “Refactoring Browser” for Smalltalk.
- Tools that automatically find candidate for refactorings. This kind of tools are rare, invariant detection tool called Diakon is an example.

These two types of automations seem to fit together quite naturally. The second type of tools is out of scope of this paper, so I will not discuss it here.

5.1 Existing Refactoring Tools

In this section I will briefly overview current existing refactoring tools. Emphasis will be put on one of them, namely Refactoring Browser for Smalltalk, since it is perhaps the most mature tool for automated refactoring [Fow99].

Opdyke [Opd92] and Griswold [Gri 91, Gri93] defined the early tools to apply refactorings and preserving the program behavior.

Recently there are more tools support refactorings. The following is an incomplete list:
- Refactoring Browser for Smalltalk - the original refactoring power tool.
- Moose Refactoring Engine - a research prototype for refactoring multiple OO languages. Currently it is a prototype with language front-ends for Smalltalk and Java. Moose Refactoring Engine is a component of Moose, which is a tool environment to reverse engineer and reengineer object-oriented software systems.
- Codemorpher - The secure refactoring browser for Java
- JRefactory – Open source Java refactoring tool, support fifteen refactorings, integrated into several Java IDEs.
- Design Pattern Transformer (DPT) - a prototype-tool for Java refactoring.
- Compost - a reengineering environment for Java
- **jFactor** – Commercial Java refactoring tool, which support eighteen refactorings.
- **Xref-Speller** – extends Emacs editor to support a set of refactoring for C and Java.
- **jFactor** - Refactoring Browser for VisualAge for Java
- **IntelliJ IDEA** - Commercial Java IDE with refactoring support, which support ten refactorings, such as renaming package/method/field/parameter, etc.
- **ECLIPSE** - The open source IDE from IBM, successor of the VisualAge product line supports the most important refactorings in the current version, more refactorings are in work. The architecture of ECLIPSE provides plugins - you may build your own refactorings.
- **Refactorit** - Integrates with NetBeans, Forte, JDeveloper and JBuilder5. Also a Standalone version. Provides many finder and filter tools in addition to refactorings.

### 5.2 Refactoring Browser for Smalltalk

Refactoring Browser is a commercial refactoring tool for Smalltalk developed by John Brant and Don Roberts, and Roberts wrote his PHD thesis on it.

Robers [Rob99] specifies the Smalltalk refactorings available in the Refactoring Browser and focuses on the possibility of combining refactorings. He observes that few of refactorings are applied on their own, so he gives a new definition of refactoring that focuses on preconditions and postconditions of the refactorings. The postconditions can be used to reduce the amount of analysis that later refactorings must perform, to derive preconditions of composite refactorings, and to calculate dependencies between refactorings. This definition allows precise calculating of postcondition-precondition dependencies among refactorings, which in turn allows compositions of refactoring to be defined. Furthermore, Roberts discusses dynamic refactoring in which the program checks for certain properties while running, applies appropriate refactorings, and then can retract those refactorings if the required conditions are later violated. The techniques mentioned above allow Refactoring Browser to support undo, user-defined composite refactorings, and multi-user refactoring.

Following I will look at the transformation framework of Refactoring Browser and how it matches the software refactoring process discussed in last section.

There are six parts to the transformation framework: the refactorings, the conditions, the parser, the tree rewriter, the formatter, and the change objects.

**Refactorings**
The Refactoring Browser implements all of the refactorings shown in Table1 of Appendix.

Each refactoring is implemented by a subclass of Refactoring. Each subclass implement two methods; preconditions, which returns the Condition object that stands for the reconditions that must be met for the refactoring to be legal, and performRefactoring, which performs the refactoring by using RefactoryChange objects. To transform the structures within methods, it uses its own parser and tree rewriter. Other parts in
framework are components used to implement the two methods that every subclass of Refactoring must implement.

**Conditions**
Condition objects represents the analyses that the refactorings need to carry out.

**The Parser**
Refactoring Browser has its own parser, which is a standard parser for Smalltalk augmented with the ability to parse a syntax that includes pattern variables.

The reason why not using standard parser exists in the difficulty of source-to-source transformations discussed in last section, which is the preservation of non-semantic information about the program text such as comments, whitespace, formatting, etc. Traditional parsing techniques do not maintain this information because parsers are usually the front-end of compilers. Since this information has no effect on the code, it is overlooked. The parser of Refactoring Browser add information to the nodes that it creates that identifies where comments were located. Information about the format of the code is left to the Formatter class.

**Tree Rewriter**
The tree rewriter is the part of the transformation framework that changes the individual methods. The parse tree rewriter takes a template in the form of a Smalltalk AST, and attempts to find a subtree that matches it. Tree Rewriter corresponds to Refactorer in the refactoring process model.

**Formatter**
Since the methods of the program are transformed by using the rewriter to perform the tree-to-tree transformation, some way must be find to get the source back from the tree. To do this, printString is implemented on ASTs to return nicely formatted code. The Formatter is corresponding to pretty printer in the refactoring process model.

**6. Discuss on Automating Refactorings**

The question of how far a refactoring-tool should support the automatic transformation may deserve a separate discussion.

One important insight gained from the refactoring researches is that while many systems started out with the goal of fully automating the transformation process, it became obvious that in handling non-trivial programs a refactoring system required some interaction with the user.

The term *automate* refers to a refactoring’s programmed check for preconditions and its execution of all source code changes. The decision of which refactorings to apply is *not* automated and is always made by a person.
There is a difficulty related to preconditions: some conditions cannot be checked automatically. In any refactoring tool, the logical correctness of refactorings is assumed[La01]. In the cases where preconditions cannot be checked automatically, a refactoring tool must request human involvement to manually verify these conditions (or assert the conditions are satisfied by the design) before proceeding.

For example, the Refactoring Browser performs a lot of refactorings automatically, but it only does them when told by a programmer. There is a code critic called Smalllnt that gives many suggestions for improvement. Although most of its suggestions are good, if you just let it modify program without checking it, it would make a lot of changes that would not make your program better.

So, until a refactoring tool can truly understand programs, it should not rewrite the source except under the control of a programmer. It is different with code optimizers, which transform a program without asking programmer, because a programmer doesn’t need to look at the assembly language output.

7. Conclusion

Manual refactoring is slow, tedious and error-prone and can benefit from using automatic refactoring tool. The goal of an automated refactoring tool is thus to help programmers to perform refactorings easily. A practical refactoring tool has several requirements that discussed in this paper. Modifying a software system usually entails modifying the source code representation. Due to the properties of refactoring transformation, i.e., source to source behavior preserving transformation, the requirements for internal data structure are different with that of other type of program transformation tools such as compiler optimizer. Abstract syntax tree (AST) annotated with type and reference information can serve as appropriate internal representation for refactoring tool. A refactoring tool therefore needs to implement program to create an AST representation of the source code, then manipulate tree-to-tree transformations and rederive source code later. Refactoring Browser is an example of practical refactoring tool.

As with most disciplines, refactoring tools can help programmers to perform refactoring easily, but only if they use them, and human should do the judgments.
Appendix:

Figure 1: Software Refactoring Process [Opd92]
<table>
<thead>
<tr>
<th>Add Class</th>
<th>Add Instance Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove Class</td>
<td>Remove Instance Variable</td>
</tr>
<tr>
<td>Rename Class</td>
<td>Rename Instance Variable</td>
</tr>
<tr>
<td>Remove Method</td>
<td>Abstract Instance Variable</td>
</tr>
<tr>
<td>Rename Method</td>
<td>Create Accessors for Instance Variable</td>
</tr>
<tr>
<td>Add Parameter to Method</td>
<td>Add Class Variable</td>
</tr>
<tr>
<td>Remove Parameter from Method</td>
<td>Remove Class Variable</td>
</tr>
<tr>
<td>Rename Temporary</td>
<td>Rename Class Variable</td>
</tr>
<tr>
<td>Inline Temporary</td>
<td>Abstract Class Variable</td>
</tr>
<tr>
<td>Convert Temporary to Instance Variable</td>
<td>Create Accessors for Class Variable</td>
</tr>
<tr>
<td>Extract Code as Temporary</td>
<td>Convert Superclass to Sibling</td>
</tr>
<tr>
<td>Extract Code as Method</td>
<td>Inline Call</td>
</tr>
<tr>
<td>Push Up/Down Method</td>
<td>Push Up/Down Instance Variable</td>
</tr>
<tr>
<td>Push Up/Down Class Variable</td>
<td>Move Method to Component</td>
</tr>
<tr>
<td>Convert Instance Variable to ValueHolder</td>
<td>Protect Instance Variable</td>
</tr>
<tr>
<td>Move Temporary to Inner Scope</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Refactorings Supported in the Refactoring Browser [Rob99]
Bibliography


- [Kup 00] Design and development of program transformation tool by Alexej Kupin (Master thesis), 2000.


• [Hor 89] Horwitz, Susan, Prins, Jan Reps, Thomas. Integrating noninterfering versions of programs. ACM Transactions on Programming Languages and Systems, Vol.11, No. 3 (July 1989), pp. 345-387


