Refactoring Tools for Extreme Programming: An Overview

Dan B Goldman
CSE503: Software Engineering

February 25, 2002

Contents

1 Introduction

I find the idea of refactoring interesting because it meshes well with my own philosophy of software development, and provides a "way out" from the constant redefinition and redevelopment of systems that happens so often when programmers get tired of their old design. Refactoring says to those programmers: Not only is it OK to spend time improving the old system, but you can end up with as good or a better design than if you were to write a new one from scratch. It bestows honor and prestige to maintenance by giving it the structure and organization it often lacks.

This paper will attempt to survey the state of the art in refactoring research.

Section 1 introduces the concepts of refactoring and extreme programming, attempting to define each of them concisely and elaborate a bit on their relationship.

Section 2 covers a selection of recent literature related to refactoring. It includes subsections on the fundamental work in the area, on automating their implementation, and attempts to assist the user in determining when to use refactorings.

Finally, section 3 lists some available tools which implement automated refactorings.

1.1 What is Refactoring?

Refactoring has as many definitions as practitioners, but perhaps the most concise and certainly the most widely cited definition is as follows: "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour." [FBB+00] A more practical-minded definition is "a technique in which a software engineer applies well-defined source-level transformations with the goal of improving the code's structure and thus reducing subsequent costs of software evolution." [KEGN01].

Refactoring can be said to be a formalization of the processes most good programmers do instinctively to restructure their code. It is a way of describing any behaviour-preserving program modification in terms of atomic procedures of renaming, moving, deleting, and introducing new code. Refactoring is best distinguished from other program evolution and restructuring paradigms by the atomicity of these operations. This indivisibility of individual refactorings has two important consequences: First, it makes manual implementation *practical*, by suggesting that program restructurings can be divided into bite-size pieces. Second, it makes it possible to prove *correctness* of certain restructurings, as discussed in section 2.1.

It's important to note that refactorings don't change or improve software from a functional point of view: The program is intended to do the same thing after refactoring that it did before refactoring. In this respect it shares traits with code transformation and optimization in compilers. But whereas the goal of code transformation is generally computational or caching efficiency, the goal

of refactoring is readability, reusability, extensibility, or some combination of the above.

1.2 What is Extreme Programming?

The popularity of refactoring has evolved alongside that of *Extreme Programming* (XP), a software development methodology which interleaves design and coding, rather than considering them as serial processes. Proponents of XP have argued that much time in traditional software development is wasted developing extensible designs which are never utilized. Furthermore, in traditional software development there is no established route for changes to requirements to propagate through specifications and design to code. [XP]

The intermingling of design and coding in XP is intended to recognize and embrace the fact that designs change throughout the development process. In order to cope with these constant changes, refactoring is seen as an essential component of Extreme Programming. Indeed, one of its basic tenets is to refactor on a continual basis. [Rob99]

This continual refactoring must necessarily be accompanied by extensive unit testing. In Extreme Programming, each refactoring is followed by a test cycle to verify that the program still behaves properly.

(Extreme Programming espouses a plethora of specific techniques for software development beyond refactoring and unit testing. Some of them, such as Pair Programming, are quite unique and controversial, and sources of considerable interest. However, in this paper we will not expend much discussion on these other aspects of XP.)

2 Refactoring Research

This section will describe some of the fundamental work in refactoring, followed by approaches for automating its execution and approaches for guiding selections of refactorings. I'll also cover a few papers which don't fit cleanly into these categories, and give my opinion on the most interesting areas for future research.

2.1 Fundamentals

The fundamental work on refactoring has focused on the definitions and mechanics of the refactoring operations, and proof of their correctness. The definition of correctness for a refactoring is that the operations did not change the behaviour of the program, and therefore any program which meets its specifications before a refactoring will continue to meet those specifications afterwards. Since refactorings in general are operations on (nearly) arbitrary source code, these proofs generally pair a precondition defining the circumstances in which a given refactoring can be applied, along with demonstrations that the dependency graphs are unchanged by the application of the refactoring. Opdyke [Opd92] has presented twenty-six low-level refactorings and three high-level ones, and provides

proofs of their correctness.

Roberts [Rob99] has augmented Opdyke's definitions by adding postcondition assertions. This simplifies analysis of sequential refactoring: Compositions of refactorings can be chained together by showing that the precondition of one is guaranteed by the postcondition of the previous.

Fowler, Beck and Brant have summarized the work of Opdyke by cataloguing his refactorings (and many others which he has collected over the years) in a reference form which makes them easy to find and implement. [FBB+00] His work is considered by many to be the definitive reference in the field.

2.2 Automating Refactorings

It's been observed [KEGN01] that refactoring isn't applied as often as might be useful. One of the reasons cited is that it can be tedious and error-prone to follow the mechanical steps involved. Therefore, a major focus of Refactoring research has been in the development of techniques for automating these steps.

Although Fowler's taxonomy of refactorings [FBB+00] does not distinguish refactorings by their scope, he notes elsewhere [Fow] that not all refactorings are equal from the point of view of automation. Some simple ones, like *Rename Method* or *Rename Variable*, are just identifier changes, and are therefore little more than syntactic modifications. But others, like *Extract Method* – which creates a new method from a tagged section of code – require considerably more semantic analysis. Also, refactorings are language-dependent. Obviously *Move Method* – which moves a method to a different class – only applies to object-oriented languages, but there are more subtle language dependencies discussed below. [OJ93] gives a detailed analysis of some of these issues with regards to creating abstract superclasses via refactoring.

Smalltalk Refactoring Browser

The best known and most widely used work in the area of automation has been the *Smalltalk Refactoring Browser* [RBJ97, Rob99], which incorporates refactoring operations directly into a Smalltalk source code browser. Its main selling points are the number of refactorings implemented, and its integration into the Smalltalk development tools. One of the criteria for its design is speed, so refactorings which might be slow to implement automatically have been omitted. Nonetheless, this tool automates over two dozen refactorings.

Roberts et al. have noted that the *Extract Method* refactoring is fairly complex when code being extracted uses variables local to the original method [RBJ97]. If the variables are reference but not assigned, they can simply be passed as parameters to the newly created method. If they are used only in the extracted code, they can become temporaries in the new method. But if the variables are assigned in the extracted code and referenced in the original method, then the code can't be extracted, because Smalltalk passes by value. This is an example of the language-dependence of refactorings.

The *Browser* takes advantage of Smalltalk's reflective facilities to create annotated parse trees. These trees are used both for context detection (matching trees) and also code modification. Smalltalk's dynamic typing makes some code impossible to statically analyze, so dynamic analysis is used to determine the actual types of variables for refactoring. The browser also uses method wrappers to perform refactorings dynamically, using "lazy evaluation" to find parts of the code that access a method being renamed or moved. This limits the completeness of the algorithm, since pieces of code not exercised by the test suite will never be analyzed and refactored. However, this limitation has been found to be unobjectionable in practice, and the tool is widely used. [RBJ97]

Graphical program restructuring

Other researchers have taken a visualization approach to refactoring. Whereas the *Smalltalk Refactoring Browser* operates in a largely text-based environments, others have attempted to develop systems in which the code and dependencies are displayed graphically. One such prototype tool for manipulating Scheme programs is described in [GB93]. Using this tool, visual representations can be manipulated using drag-and-drop. These manipulations implicitly initiate refactorings, creating new classes and moving variables or methods between classes in the underlying source code.

This approach is limited to relatively gross transformations involving whole objects or methods. Refactorings involving code fragments still must rely on a text-based interface to specify the relevant sections. The authors also concede that the abstraction to visual representations may obscure subtle relationships and implementation details which are pertinent to the design of the program as a whole.

Refactoring by Field

Even in text-selection or browser-based tools, refactorings involving code fragments present special problems in user interface. The sections of code which must be extracted to form a new method may not be contiguous, and may even require unrolling or duplication of loops and other control structures.

One approach to assisting the user in selecting code fragments for *Extract Method* and related refactorings takes advantage of a diagnostic called *program slicing*. Program slicing "extracts from code of a program a set of statements that may affect the value of a variable of interest at a specified program point." [Mar01]. One may think of this in terms of "narrowing the field of view" to only those statements which contribute to the value of a variable.

In Maruyama's prototype tool for method extraction, a programmer indicates variables in a program rather than selecting code fragment text. The system automatically detects the lines of code which are dependent on and depended upon by that variable at that program point, and constructs a number of alternative refactored methods involving contiguous blocks of those statements. The user may then select the most appropriate choice for his or her needs. This

technique is still quite nascent; it's not clear whether it will scale well to "real" code due to the computational demands of program slicing.

Implementation Concerns

Although a number of tools do exist which support strongly typed languages (see section 3), they have been slower to appear, and many are limited in scope, implementing just a few of the simpler syntactic refactorings. Seguin provides a brief discussion of some of the difficulties of implementing the *Push Up Field to Superclass* refactoring in Java [Seg00]. The central difficulty occurs when multiple sibling classes have a variable of the same name with different types or scopes. The author implemented a simple but potentially problematic solution to this problem in his *JRefactory* tool, by changing the scope of the variable to protected and leaving identically named fields with different types untouched, so they may continue to override the refactored superclass field.

2.3 Automating "Smell Detection"

Automating the mechanical aspects of refactoring is still a topic of active tool development, but most still rely on an expert user to determine the need for a refactoring in any given situation. Kent Beck and Martin Fowler [FBB+00] describe this process of looking for structures in the code that suggest the possibility and desirability of refactoring as finding "bad smells" in code. Each "smell" is associated with refactorings that are likely to improve the code's readability and reusability. Although they opine that "no set of metrics rivals informed human intuition," that hasn't stopped anyone from trying; this seems to be a lively area for speculative research. Indeed, it seems likely that design problems which may be overlooked or ignored by a programmer might be identified by an automated or semi-automated approach.

Invariants approach

One approach to smell detection takes advantage of existing tools for program invariant detection. In [KEGN01], the authors describe a set of refactoring pre- and postconditions and "smells" in terms of program invariants. The *Daikon* invariant detection tool is used as a preprocess to detect invariants. Then by looking for these predefined patterns in the computed invariants, their tool can suggest likely refactorings. Daikon uses dynamic analysis to identify invariants, so this approach can identify smells which are not obvious from a static analysis of the code, even those which are difficult for a skilled programmer to detect. The tool has been tested on a large software system, and found to make orthogonal suggestions to a clone-detection tool based on text-based pattern matching.

Metrics approaches

Many "smells" are not as deterministic. The decision to move a method or field to a different class is rarely motivated by an invariant property of that element; rather it is dependent on more subjective concerns like "method Z of class A is called *more often* from class B than A, and uses more fields from class B than A, therefore I will move it from class A to class B." Fowler describes this smell as *Feature Envy*: "a method that seems more interested in a class other than the one it actually is in" [FBB+00]. While this subjective diagnostic may be easy to utilize for methods which only reference a few variables, it doesn't help determine whether moving a method or a field will give better results. In fact, moving a field can be quite hazardous, as there may be a huge number of other methods which will need to be modified. This suggests using statistical methods to search code for smells, and several researchers have taken this tack:

The *Crocodile* tool utilizes a distance-based cohesion metric which counts numbers of shared properties or attributes between code entities [SSL01]. This metric is used to compute the distances between methods and attributes. The distances can then be used as spring lengths in a dynamic point-mass spring simulation, resulting in a static three-dimensional scene which can be viewed and navigated using any VRML browser. Methods are represented as spheres and attributes as cubes, using different colors to represent members of different classes.

Such a representation gives a concrete visualization of the dependencies in the code, depicting related entities as being physically close together and unrelated entities further apart. With such a visualization, *Feature Envy* is readily observable as an object of one color which neighbors several objects of a different color. Although running the simulation can be quite slow on current systems, techniques such as this may prove viable in the future for targeting refactorings.

A much more aggressive and unusual metrics-based approach is taken in [MS99]. Unlike the technique used in [SSL01], which relies on a static analysis for its metrics, or [KEGN01], which uses dynamic analysis to find invariants, this method assigns weights to dependencies between methods according to how often they have been overridden by users of a framework. This gives a historical, and in fact a *user-dependent* metric for choosing refactorings, based on the observation that programmers will reuse and modify their frameworks in a consistent manner over time. *Extract Method* refactorings are executed automatically to assimilate "hot spots" in the framework, in an attempt to reduce the number of methods which need to be overridden.

This particular application leads to different refactorings for different programmers – which seems likely to cause confusion and incompatibility. However, the fundamental concept of statistically analyzing code modification history appears potentially quite useful. (It isn't explicitly characterized this way by [MS99], but I will consider this sort of 'historical analysis' as a complement to static and dynamic analyses.)

Fine-grained restructuring

The most radical approaches to program restructuring abandon the programmer's original design, attempting to reconstruct it based purely on automatically-detected similarity relationships.

The *Guru* tool takes just such an approach to restructuring code written in *Self* (a lisp-like language) by discarding the programmer's original class hierarchy and method boundaries, replacing them with a "fully-factored" class hierarchy which aggressively combines duplicated code all the way down to the expression level [Moo96]. (The tool is named *Guru* because it assists in *Self*-improvement.) This process is slow – a small class hierarchy took 8 hours to restructure – and apparently limited to languages with similar properties to *Self*, so it's not clear if this is a viable tool for many real-world systems. Furthermore, this tool could be described as a program restructuring tool rather than a refactoring tool, since it takes on the much larger task of reorganizing an entire class hierarchy or framework. However, I include it here because the mechanics of the restructuring are described in terms of atomic refactorings.

2.4 Relationship to Extreme Programming

A few researchers have explicitly examined the relationship between Extreme Programming and Refactoring. [Deu01] has analyzed how Extreme Programming affects program understanding. XP's emphasis on people and source code suggests that program understanding is at the core of XP, so an attempt to understand the process in terms of program understanding is crucial to evaluating its strengths and weaknesses. Van Deursen finds that the very process of refactoring enhances program understanding by forcing the programmer to systematically recognize and refine the relationships between elements of a program. However, refactoring in the context of collective code ownership can lead to conflicts: "what is intuitive and clear to one programmer may be hard to follow for another." [Deu01] Furthermore, as code is continually changing, more programmers may need to reunderstand code they once understood, even if the functionality is unchanged. [KHH+00] found that as the size of a code base increases, the proportion of time spent refactoring increases relative to the time spent developing new features. While this is not unexpected (refactoring can be seen as a maintenance task), it does point out some of the drawbacks of refactoring in the context of Extreme Programming.

2.5 Other Work

The following recent work is related to refactoring but doesn't fit cleanly into any of the previous categories:

Refactoring and Design

Some authors have begun to explore refactoring within the larger scope of design processes.

[Cin00] describes a prototype tool (*DPT - Design Pattern Tool*) in which a design pattern is chosen as a target for a code transformation. The transformation is decomposed into a sequence of *minipatterns*, which are design motifs appearing frequently in catalogues of design patterns. In one example, the *Factory Method* pattern is considered to be composed of three constituent minipatterns, *Abstraction*, *Encapsulate Construction*, and *Abstract Access*. Each of these minipatterns can be associated with a *minitransformation*, which is a sequence of refactorings leading to the implementation of the desired minipattern. By composing transformations entirely of refactorings which have been proven to preserve behaviour, the entire code transformation can be proven as such.

An even wider view extends the concept of refactoring beyond source code to encompass design models as well. [BX01] describes a cascaded sequence of refactoring feature models (eg. changing required features to optional or vice versa), refactoring use cases (eg. abstracting actors, merging actors or behaviours, etc.), refactoring architectural models (eg. splitting or merging interfaces, promoting or delegating services), as well as the "usual" refactorings involving class hierarchy and source code. This view may be too abstract to be practical...

2.6 Topics for Future Research

With such a limited overview of the active research in refactoring and related techniques, it's difficult to assess what the most promising avenues of future research might be. But speculation is more fun than analysis, so here goes...!

Although refactoring has certainly begun to take hold in a variety of programming languages, it seems to have evolved in languages with strong reflective capabilities, like Smalltalk. This isn't too surprising: The programmers developing refactoring tools are naturally most likely to write them using the same language they are thinking about refactoring: The domain knowledge is the same both for the problem and the implementation of a solution. However, this suggests that refactoring might benefit from even stronger integration of reflective features in languages themselves.

For example, the Python refactoring tool *Bicycle Repair Man* (see 3.3) is built on a framework which replaces python's own parser module with its own wrapper, giving a set of higher level tools to the programmer who wants to implement refactorings. To change the name of a variable local to a module, one calls the parser function to evaluate the module, then walks the parse tree to find and rename all instances of the identifier, and re-emits the source code from the parse tree. All of this requires only a few dozen lines of code. Although native python does provide basic access to its own parser, the functions in the above example which rename identifiers and re-emit source code were implemented in the wrapper code. It's not hard to imagine that structured support for these types of operations in other languages could make both refactoring and smell detection more easily and widely implemented.

There are also ways in which improved support for refactoring in integrated development environments could lead to qualitative leaps in its applications.

One specific problem noted by [RBJ97] is that refactoring a framework or library often requires clients to refactor their own code in nontrivial ways. They suggest including "packages of refactorings" with each new code release. Much like a software patch, these packages could be executed on the client's code base to automatically implement the necessary changes to work with the new code.

Finally, the concept of historical analysis (see 2.3) could be combined with machine learning techniques to create some really unique smell-detection methods. [MS99] offers a first stab in that direction, whereas [DDN00] discusses how to identify refactorings executed by hand by "expert programmers". One could create a large database of such manual refactorings, paired with the sections of parse trees they impact, and try to mine patterns from the commonalities. By identifying similar patterns in other code, a machine learning algorithm might be able to identify bad smells using those learned heuristics. Some day perhaps we'll have jumping paper clips in our code browsers asking us:

"Are you sure you don't want to put that method in another class?"...

3 Tools

The following is a summary of some of the tools available with refactoring features, for a variety of programming languages and platforms.

3.1 Smalltalk Tools

Smalltalk Refactoring Browser

`http://st-www.cs.uiuc.edu/users/brant/Refactory`

The best-known refactoring tool, this integrated SmallTalk development environment features 28 automated refactorings. The original browser has been ported by the original authors to the VisualWorks, ENVY, and IBM VisualAge IDEs, though others have implemented ports on other platforms: The code is open source.

3.2 Java Tools

IntelliJ Idea

`http:www.intellij.com/idea`

This commercial IDE was designed with refactoring features in mind from the start. Martin Fowler [Fow] has noted this as one of the first working java refactoring tools to cross the "rubicon" of refactorings requiring parse tree analysis. The latest version features 25 refactorings and has been tested on Windows and Linux. It runs natively in java, using JDK 1.3.

XRefactory

`http://xref-tech.com/speller`

This shareware product is an emacs-based refactoring browser. It implements about 20 refactorings in java (and about 5 in C), and is notable for its indexing speed: It has been tested to index 1 million lines of java code in 2 minutes, and updates the index incrementally. It can also convert source code to hyperlinked HTML.

DPT

`http://dpt.kupin.de/`

This free tool, developed in a research context, includes 25 refactorings and 'wizards'. It is written in Java, and has been tested under Windows and Linux.

JFactor

`http://www.instantiations.com/jfactor`

This commercial plugin for VisualAge Java implements 17 automated refactorings.

JRefactory

`http://jrefactory.sourceforge.net`

Jrefactory is a tool with a command line interface as well as JBuilder and Elixir plugins. It implements about 15 refactorings in these environments.

Transmogrify

`http://transmogrify.sourceforge.net`

This open source plugin for JBuilder and Forte4Java implements 5 common refactorings.

Retool

`http://www.chive.com/retool.htm`

This commercial open tool for JBuilder has some primitive renaming and extract method refactorings.

3.3 Python Tools

Bicycle Repair Man

`http://bicyclerepair.sourceforge.net`

This open source project, still in alpha stage, has only one refactoring at the present time.

PythonWorks

`http://www.pythonworks.com`

This Python IDE has announced refactoring support in their next version, and the authors have shown demonstrations at international conferences, but the currently available version does not yet have these features.

References

[BX01] Greg Butler and Lugang Xu. "Cascaded Refactoring for Framework Evolution." *Proceedings of 2001 Symposium on Software Reusability*, ACM Press, 2001, pp. 51-57.

[Cin00] Mel Ó Cinnéide. "Automated Refactoring to Introduce Design Patterns." *Proceedings of the International Conference on Software Engineering (Doctoral Workshop)*, Limerick, 2000.

[DDN00] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, "Finding Refactorings via Change Metrics," *Proceedings OOPSLA 2000 (Object-Oriented Programming, Systems, Languages, and Applications)*, ACM Press, October 2000

[Deu01] Arie van Deursen. "Program Comprehension Risks and Opportunities in Extreme Programming." *Proceedings of the Eighth Working Conference on Reverse Engineering*, Oct. 2-5, 2001, pp. 176-185.

[Fow] Martin Fowler. *Crossing Refactoring's Rubicon.* `http://www.martinfowler.com/articles/refactoringRubicon.html`, February, 2002.

[FBB+00] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.

[GB93] William G. Griswold and Robert W. Bowdidge. *ICSE Workshop on Studies of Software Design*, 1993, pp. 127-139.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. "Automated Support for Program Refactoring using Invariants." *International Conference on Software Maintenance (ICSM '01)*, Florence, Italy, November 6-10, 2001, pp. 736-743.

[KHH+00] Jeremy Kivi, Darlene Haydon, Jason Hayes, Ryan Schneider, and Giancarlo Succi. "Extreme Programming: A University Team Design Experience." *2000 Canadian Conference on Electrical and Computer Engineering*, Halifax, NS, Canada, March 7-10, 2000, v. 2, pp. 816-820.

[Mar01] Katsuhisa Maruyama. "Automated Method-Extraction Refactoring by Using Block-Based Slicing." In *Proceedings of SSR '01, Symposium on Software Reusability*, Toronto, Ontario, Canada, May 18-20, 2001, pp. 31-40.

[MS99] Katsuhisa Maruyama and Ken-ichi Shima. "Automatic Method Refactoring Using Weighted Dependence Graphs." *Proceedings of the 1999 International Conference on Software Engineering,* Los Angeles, California, USA, pp. 236-245.

[Moo96] Ivan Moore. "Automatic Inheritance Heirarchy Restructuring and Method Refactoring." In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, California, USA, pp. 235-250.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[OJ93] William F. Opdyke. "Creating Abstract Superclasses by Refactoring." In *Proceedings of the 1993 ACM Conference on Computer Science*, pp. 66-73.

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[RBJ97] "A Refactoring Tool for Smalltalk." *Theory and Practice of Object Systems*, v. 3(4), 1997, pp. 253-263.

[Seg00] Christopher Seguin. "Refactoring Tool Challenges in a Strongly Typed Language." *Addendum to the 2000 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Minneapolis, Minnesota, USA, pp. 101-102.

[SSL01] Frank Simon, Frank Steinbrückner, Claus Lewerentz. "Metrics Based Refactoring." In *Fifth European Conference on Software Maintenance and Reengineering* March 14-16, 2001, pp. 30-38.

[XP] *Extreme Programming: A Gentle Introduction*, `http://www.extremeprogramming.org`, February 2002.