Matthew Beckler

Assignment #4

Research Paper

In modern software development cycles, maintainence of existing code is an extremely important, but time-consuming task. Fred Brooks claims that “The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it.” (*1*) Surely then, the average programmer must know about the benefits of maintainence? Suprisingly, the answer appears to be “No”, as Robert Glass comments, “Few computing academics teach maintenance, even in software engineering programs. It seems to be a topic most computing professionals would like to sweep under the rug.” (*2*) One of the most beneficial types of maintainence is a process known as refactoring, which has been defined by Fowler 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.” (*3*) Refactoring does not change the external interactions of a system, but merely improves upon the existing design, hopefully improving performance, correctness, and maintainability. In this paper, we will be introducing the basic concepts related to software refactoring, as well as exploring the motivations and justifications for the use of automated refactoring tools. Programmer opinion of these automated and semi-automated refactoring tools will be investigated, with a number of industry heavyweights profering opinions on the matter.

As previously mentioned, software maintainence occupies a significant portion of developer time. These maintainence activites can be targeted towards improving a number of things, but the IEEE defines a few standard maintainence targets, including fixing existing faults (Corrective Maintainence), adapting a software system to a new computing environment (Adaptive Maintainence), and maintainence intended to improve the performance,

maintainability, or other aspects of the system (Perfective Maintainence). (*4*) Restructuring, a generic term applied to most software maintainence and defined by Chikofsky and Cross as, “the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behavior,” (*5*), and refactoring, a term of similar definition but normally reserved for object-oriented systems, can be a tremendous aid to improve software maintainability, both at the time of the original development, as well as during maintainence of legacy code. Unfortunately, most maintainence-related restructuring is performed under looming deadlines, making it an, “...error-prone and expensive activity.” (*6*) Sometimes, managers discourage refactoring, as it does not produce any measurable output towards a product release. In other cases, developers may avoid refactoring for fear of breaking a subtle component of the system. (*5*) Automated tools of many varieties exist to automate the associated tedium, and eliminate much of the potential for developer error.

The most basic type of automated refactoring available to the software developer is that of compiler optimizations. True to definition, “their goal is to improve the performance of the program, yet preserve its behavior.” (*7*) Many times, this includes the in-lining of heavily called functions, or the un-rolling of loops. With different languages and processor architectures, compilers will select different optimizations. For example, if a particular language has a high overhead for function calls, a compiler may choose to inline many function calls. Many architectures utilize instruction pre-fetching, also known as pipe-lining, which is interrupted by looping constructs, so loops may be 'un-rolled' into one large flow, at the cost of increased size. In some environments, such as Reduced Instruction Set Computers (RISC), the limited number of instructions makes it nearly impossible for a human to efficiently choose the best instruction order, and must therefore depend on a semi-intelligent compiler to produce efficient code. While

this is certainly an important area of research, it is not the focus of this paper, and will not be discussed further.

At the other end of the refactoring automation spectrum lie the manual, human-operated tools. These tools, consisting mostly of the text-editor's find-and-replace utility, along with the cut-copy-paste commands, are the old way of doing things. While many novice programmers feel that these are sufficient for refactoring, experienced developers who have witnessed the accidental 'overwriting' of a re-used variable, or the confusion caused by an over-zealous find-and-replace, know that a good refactoring tool can save many headaches, and many hours of developer time.

Existing in an effective middle-ground, semi-automated refactoring tools provide a good trade-off for most developers. These tools are commonly built into an Integrated Development Environment (IDE), providing quick access to refactorings through a context menu. They are known as semi-automated because they require an intelligent user to detect areas in a program where a particular refactoring would potentially be useful, and instruct the tool to implement the appropriate refactoring. An example of this would be a developer wanting to remove ambiguity in a varible's name. A standard find-and-replace might be sufficient, but if two variables, in different scope, shared the same name, the other variable might be renamed as well. In his Ph.D. thesis, Cinnéide investigates using automated tools to "...remove the burden to tedious and error-prone code reogranisation from the designer." (*8*) Some modern IDE's provide a context menu to allow the developer to rename a variable, with the IDE handling the details of the variable's scope. These tools focus more on the application of a user-selected refactoring, rather than detection or identification of appropriate locations for refactorings. (*9*)

Quite often, a developer will notice one of the many "Code-Smells" described in detail in

Fowler's “Refactoring” (*3*), and utilize the IDE's semi-automatic refactoring tools to remove such a smell. Other times, a system is in need of a greater overhaul than just one-off refactorings. This situation occurs often when inheriting legacy source code. In cases such as this, the use of an automatic tool to identify areas of the code that are prime candidates for refactoring can be quite an advantage. The Daikon dynamic program invariant detector (*10*) is a software application that is used by Kataoka et al, to identify areas of source code that contain certain patterns of program invariants, which are likely candidates for refactoring (*5*). These potential refactoring locations can then be evaluaged by the developer, to determine if they really are good locations to refactor. Kataoka et al give an example of an Icon class that maintained separate *height* and *width* properties. It was discovered that the icons used in the software were always square, opening the possibility of refactoring them into a single property. In this particular instance, the developer chose to ignore this potential refactoring, as the increased flexibility was desired in a generic Icon class. (*5*)

Coleman et al, identify a number of metrics which can be used to evaluate the overall maintainability of a software package. (*11*) These metrics can be used in various ways, such as determining if it would be more cost-effective to purchase an existing software package, or to develop an equivalent in-house. Another, more relevant, use, is to use some sort of software metric as a way to evaluate the effectiveness of a refactoring, by comparing a “before” and “after” score. The metrics used can vary greatly, with Coleman's group considering a polynomial-based metric, based on factors such as average lines of code per module and average number of comments per sub-module (function/method). (*11*) Other researchers have developed methods to evaluate cohesion and coupling, based on properties such as return-value coupling, parameter passing coupling, and shared-value coupling. (*12*) These various maintainability

scores can be very helpful when evaluating refactorings or other restructurings, but programmer intuition and understanding can often be more useful than even the best automated metrics.

Not all industry experts feel favorably about software refactoring tools. Steve Yegge, a prolific blogger and Google employee, feels quite strongly against automated software refactoring tools. He believes that programmers who program in a language with such “push-button” refactoring tools available become lazier, putting more emphasis on fixing already-written code. Code doesn't have to start out bad, and get fixed later, he says, but we should start with good design fundamentals, with attention to details. “How did that code get smelly in the first place?...We were making dozens, hundreds of little mistakes that added up to some pretty smelly code.” (*13*) Yegge argues that refactoring should be a design-time process, influencing the initial creation of a system, rather than a 'cleanup' step that occures during the maintainence phase, a mind-set that accustomes programmers to accept the notion that code always starts bad, and must be cleaned with magic tools after it's been written.

Ideally, the strict adherence to sound software development guidelines, starting with the initial design and frameworking steps, will lead to a well-designed, coherent system. In reality, with changing requirements and environments, software maintainence starts as soon as the first line of code is written. As Yegge notes, as soon as a bad smell is noticed, the offending section should be refactored immediately, or as soon as possible. (*13*) Often, this immediate refactoring will interfere with a scheduled release, especially with an oft-released agile project, so it may be necessary to save major refactorings for later. These should be completed as soon as possible after the release.

When working on an agile-based project, refactorings are a part of every developer's daily life. When work on a waterfall-based project, refactorings represent a much need

opportunity to improve the underlying layout and structure without changing any of the system's external behaviors. While there are fewer releases to worry about, presumably only one, many people believe that refactoring has no place in a waterfall-based project. There are many reasons behind these feelings, from unfamiliarity with newer, flexible, refactoring-based agile development schemes, to the untrue notion that the detailed specifications of the waterfall process disallows refactoring.

Altogether, software refactoring can greatly improve the readibility and maintainability of a software system. Many tools exist to help developers with refactoring, from the simple point and click, semi-automated tools common in many IDE's, to fully-automated routines, that identify potential trouble areas and suggest possible refactorings to alevate the trouble spot. Regardless of the type of tool used, or the specific refactoring applied, proper refactoring results in a software system that is both properly aligned with the real-world system it is representing, as well as being easier to navigate and maintain, from a developer's viewpoint.

Literature Cited

(1) F.P. Brooks, The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, Reading, Mass., 1982.

(2) Robert L. Glass, "Maintenance: Less Is Not More," *IEEE Software* ,vol. 15, no. 4, pp. 67-68, July/August, 1998.

(3) Fowler, M., “Refactoring: Improving the Design of Existing Programs," Addison-Wesley, 1999.

(4) IEEE Std. 610.12-1990, “Glossary of Software Engineering Terminology,” in *Software Engineering Standards Collction*, IEEE CS Press, Los Alamitos, Calif., 1993.

(5) G - Chikofsky, E.J.; Cross, J.H., II, "Reverse Engineering and Design Recovery: a Taxonomy," *IEEE Software*, vol.7, no.1 pp.13-17, Jan 1990.

(6) H - Griswold, W. G., “Program Restructuring as an Aid to Software Maintenance,” Ph.D. thesis, University of Washington (1991).

(7) Tom Mens, Tom Tourwe, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126-139, February, 2004.

(8) C - Cinnéide, Mel Ó. Automated Application of Design Patterns: A Refactoring Approach. Ph.D. Thesis, Trinity College, Dublin, Ireland, 2000.

(9) J - Mens, T., Demeyer, S., Du Bois, B., Stenten, H., Van Gorp, P., “Refactoring: Current Research and Future Trends”. Electronic Notes in Theoretical Computer Science 2003, vol. 82, no. 3.

(10) The Daikon dynamic invariant detector. http://pag.csail.mit.edu/daikon/ (accessed November 2006)

(11) Don Coleman, Dan Ash, Bruce Lowther, Paul Oman, "Using Metrics to Evaluate

Software System Maintainability," *Computer*, vol. 27, no. 8, pp. 44-49, August, 1994.

(12)Kataoka, Y. Imai, T. Andou, H. Fukaya, T., "A quantitative evaluation of maintainability enhancement by refactoring," International Conference on Software Maintenance, pp. 576 – 585, October, 2002.

(13)Yegge, Steve. "Transformation," http://www.oreillynet.com/ruby/blog/2006/03/transformation.html (accessed November 2006)