Using Software Metrics and Program Slicing for Refactoring

Dr. Ricky E. Sward
U.S. Air Force Academy

Dr. A.T. Chamillard
University of Colorado at Colorado Springs

Dr. David A. Cook
AEgis Technologies Group, Inc.

Refactoring can improve the quality of a software system as measured by coupling, cohesion, and cyclomatic complexity, but knowing which refactoring choices should be implemented is key. This article presents an approach that guides the refactoring of software systems by combining the use of software metrics and a technique called program slicing. Program slices produced from a single software module are sorted by the respective values of the metrics; a design that provides the most beneficial metric values can be selected from these. This approach can produce a software system with higher quality and maintainability as measured by the metrics.

Many systems designed today have very long life cycles, especially in the military. Often, a software program is expected to perform for many years, and undergo frequent updates and requirements changes. Large-scale software systems are prone to quality problems [1] during development. Constant changes to existing systems only lead to additional quality problems.

One way to help control defects and reduce high maintenance costs is to use refactoring. Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [2]. Refactoring is an option during both the development and maintenance phases. Unfortunately, refactoring the design can be very resource intensive, and automated tool support is considered crucial [3].

This article presents an approach for automating a large part of the evaluation and refactoring process. By combining the use of software metrics and a technique called program slicing, the refactoring process is guided toward a design with higher quality and more maintainability. First, we discuss how several different software metrics can be used to evaluate software quality and the effects those metrics have on defects, testing effort, and maintenance cost. We then discuss how program slicing can use those metrics to guide design-refactoring decisions. The final section presents our conclusions.

Software Metrics

We use software metrics to try to quantify particular characteristics of software systems, such as quality, maintainability, or reliability. In general, however, these characteristics cannot be measured directly. Instead, we directly measure particular attributes of software by using software metrics and then infer information about quality from those direct measurements [4].

Three commonly used software metrics are coupling, cohesion, and McCabe's Cyclomatic Complexity [5]; all three have been extended from their original definitions for use with object-oriented systems (OOS). In this article, we discuss our ideas in the context of a system that was implemented using structured design techniques, though the process could also be extended for use with OOS.

"One way to help control defects and reduce high maintenance costs is to use refactoring."

The first metric to consider is coupling, which measures the strength of the connections between the software modules that comprise a particular system to quantify the dependencies between the modules. The key idea is that the more interdependent the modules in the system are, the more difficult the system is to understand and the more likely it is that changes to one module will affect other modules in the system.

Yourdon [6] originally described several different kinds of coupling, including data coupling, control coupling, hybrid coupling, and so on. McConnell [7] has updated coupling to include classes of coupling. In the technique described in the following section, we only consider data – or normal – coupling. In other words, the main focus in terms of coupling is on the information that flows between the modules in the system. We measure this coupling by counting the number of parameters (i.e., pieces of information) passed into and out of each module.

As Yourdon points out, "The coupling between modules in tentative structural designs can be evaluated to guide the designer toward less expensive structures." Our idea is to provide precisely this kind of guidance, but to do so with extensive automated tool support. This guidance would be useful in both the design and the maintenance phases, though we believe most refactoring occurs in the maintenance phase.

The second metric to consider is cohesion, which measures how strongly the elements of each module are related to each other. Cohesion was originally defined in an article by Stevens [8], and the concept has been updated as programming languages and their capabilities have evolved. McConnell [7] contains a working definition of the current classes of cohesion. At a high level, a module with high cohesion accomplishes a single function using only the data required to accomplish that function. As with coupling, Yourdon defined multiple levels of cohesion, though we limit our interest to functional cohesion in this article.

Coupling and cohesion are related, though not perfectly correlated. As we increase the cohesion of the modules in the system, we tend to reduce the coupling between those modules. This is an important consideration, because although researchers have proposed various ways to measure cohesion [9], it is much more difficult to measure than coupling. In this approach we do not measure cohesion directly, but instead rely on the relationship between coupling and cohesion to infer information about the quality of a module. If we find

that we also need to directly measure cohesion to make our approach more effective in practice, we can extend the metrics calculated by the tool to include cohesion as well.

The final metric, which is probably the most commonly used metric of those discussed here, is McCabe's Cyclomatic Complexity. At the module level, this metric is the number of linearly independent paths through the module. Modules that contain many possible paths are more complex than those with fewer paths, so as the cyclomatic complexity of a module increases, so does its complexity. We note that this metric is equal to one more than the number of decisions contained in the module.

Based on the discussion above, our approach uses coupling and cyclomatic complexity metrics as described in the following section. The coupling metric provides insight into the interaction between the modules in the system, while the cyclomatic complexity metric gives insight into the complexity of each individual module. Remember that we do not directly include cohesion, though we could extend our approach to do so if it proves to be helpful in practice.

As stated in the introduction, our goal is to use software metrics to provide guidance to those undertaking refactoring efforts. It is important to note, of course, that we do not refactor code simply for the sake of better code; rather, we expect some return on the investment expended on any refactoring efforts. We must therefore consider some of the important relationships between our software metrics and software quality, testing costs, and maintenance costs.

Intuitively, we expect software with high coupling and low cohesion to be of lower quality than software with low coupling and high cohesion. The additional programmer effort required for understanding highly interrelated modules and their effects on each other leads to a higher potential for mistakes. Similarly, a programmer working on a module with low cohesion needs to keep track of multiple functions being performed by the module rather than on a single function, which also increases the potential for programmer error. Our intuition turns out to be true in practice. Research on operational systems has shown that modules with high coupling/low cohesion contained seven times as many errors as modules with low coupling/high cohesion [10]. In addition, programmers spent almost 22 times as many hours correcting the errors in those modules with high coupling/low cohesion. Clearly, coupling and cohesion have a significant impact on both the quality of the software and the effort required to fix errors in the software.

We also expect all three metrics to have an impact on the amount of effort associated with software testing. Because coupling measures the dependencies between modules, higher coupling implies the need to expend more effort accomplishing integration testing of the modules. Modules with low cohesion implement more than one function; testing the functionality of that module (typically during unit testing) requires more test cases to cover all of that module's functionality. Cyclomatic complexity essentially measures the number of paths through a module, so modules with higher cyclomatic complexity will require more test cases to cover all the paths.

"The coupling metric provides insight into the interaction between the modules in the system, while the cyclomatic complexity metric gives insight into the complexity of each individual module."

Discussions of software quality and testing effort apply both to the original development of a system and maintenance of that system. We are also concerned, of course, with the cost of maintaining systems. Research shows that we can account for more than half of the variability of maintenance productivity by taking cyclomatic complexity into account [11]; in other words, we can significantly improve our estimates of maintenance costs through consideration of the cyclomatic complexity (and lines of code) for the system to be maintained. Perhaps even more importantly, we know that modules with higher cyclomatic complexity are more difficult to maintain, so if we can reduce the complexity of the modules we can reasonably expect a corresponding reduction in maintenance costs.

It is clear that refactoring software to improve coupling, cohesion, and cyclomatic complexity of the software yields improvements in overall software quality and reductions in testing and maintenance costs. Despite the clear benefits associated with refactoring, the amount of effort required to refactor large systems without tool support is generally prohibitive [3]. Metrics and other methods have been proposed to help guide program refactoring [12, 13, 14, 15]. One problem with traditional metrics is that they are often not useful for making fine distinctions between routines and modules [7, 16]. Refactoring does not have this limitation. The following section describes our approach for guiding the refactoring process through the use of program slicing and software metrics.

Program Slicing, Metrics, and Refactoring

When considering options for refactoring, a technique known as program slicing can be used to isolate portions of a software system. A program slice is a projection of the behavior from a software module that is needed to produce a particular value in the module [17]. By slicing a particular variable or parameter in a software module, only the lines of code required to produce that variable or parameter are extracted from the module. The resulting lines of code can be built into a separate module with variables and parameters of their own.

For example, consider the Ada procedure shown on the left side in Figure 1 (see page 22). This procedure produces both the Highest_Max parameter and the Lowest_Min parameter. The Ada procedure shown in the upper right side of Figure 1 shows the program slice built by slicing on the Highest_Max parameter. The Ada procedure shown in the lower right side of Figure 1 shows the program slice built by slicing on the Lowest_Min parameter. Note that only those parameters needed to produce either Highest_Max or Lowest_Min are included in their respective program slices.

Program slicing is useful for refactoring software systems [18] because it isolates portions of the software. For example as shown in Figure 1, instead of a single procedure that produces both the Highest_Max value and the Lowest_Min value, we now have two procedures that each produce a single value. The challenge with using program slicing for

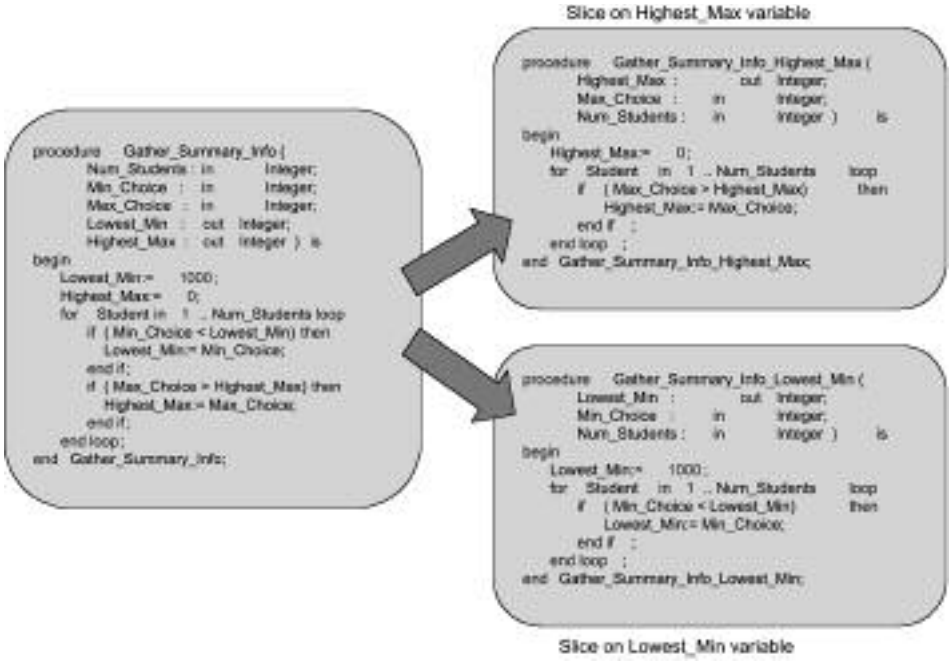


Figure 1: *A Predictable Substation Assembly*

refactoring is determining how to slice the software system properly in order to maximize maintainability and quality.

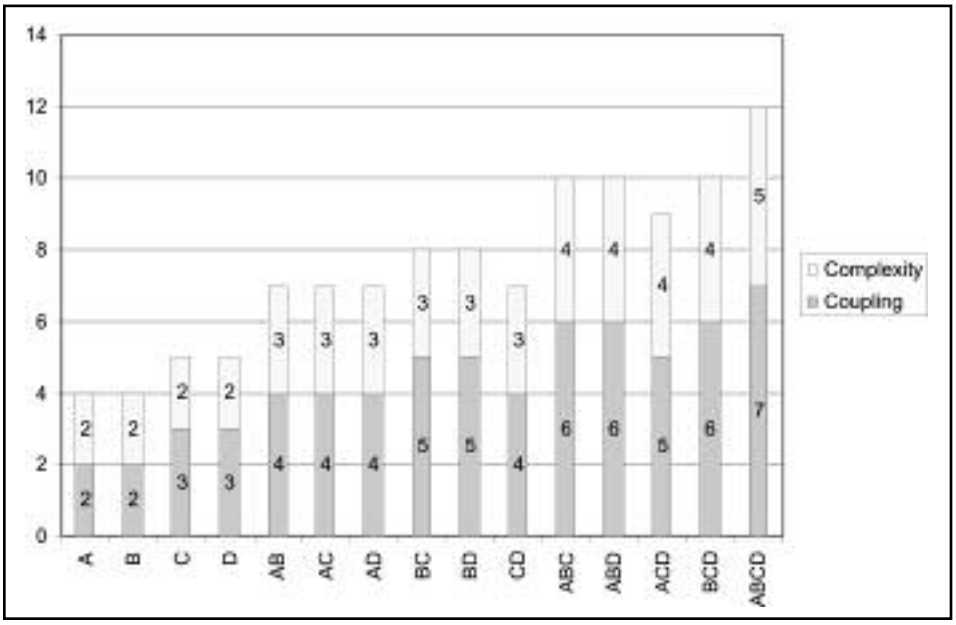
Since we want to refactor our software to improve coupling, cohesion, and cyclomatic complexity, software metrics can guide our choices when we use program slicing for refactoring. Each time we produce program slices, we can compare the values of the metrics from the original procedure to the resulting program slices. Clearly, program slices that produce better coupling, cohesion, and cyclomatic complexity are better than the original procedure and should be included in the refactored software system.

During refactoring, program slicing may reduce the coupling between modules in the software system. For example, in Figure 1 the original procedure includes five parameters, but each program slice includes only three parameters. As we discussed previously, the number of parameters in a software module can measure coupling, so in this case, program slicing has reduced the coupling between modules.

Program slicing may also improve the cyclomatic complexity of a software module during refactoring. For the procedure shown on the left side of Figure 1, the value of the cyclomatic complexity metric is three. For each of the program slices, the value of that metric is two. In this case, refactoring using program slices has resulted in software modules that have a lower value for the cyclomatic complexity metric.

Using program slicing for refactoring can therefore improve the quality and maintainability of software modules as measured by coupling, cohesion, and cyclomatic complexity. Admittedly, the example shown in Figure 1 is simplistic, but it demonstrates how using program slicing and metrics can guide the refactoring process.

Figure 2: *Metrics for Program Slices*



Slicing on Combinations of Variables

As is often the case, software systems contain modules that are much more complicated than the one shown in Figure 1. Software modules often have many different parameters (high coupling) and contain high levels of cyclomatic complexity. These modules present an opportunity to improve overall system quality by refactoring using the program slicing technique on different combinations of the parameters.

For example, consider a module that produces four values. For illustration, we will call the module Produces_Four and call the four parameters A, B, C, and D. By using program slicing, we can build 15 different software modules, including the original module, from the possible combinations of these parameters. We can then calculate the coupling and cyclomatic complexity metric for each of the 15 modules individually.

Figure 2 shows the values of the coupling and cyclomatic complexity metrics for the 15 modules. The reader should realize that these are the values for program slices that were built from the Produces_Four module that we used in our example. These values would differ for other program slices built from other modules depending on the code contained in those modules. Note that in Figure 2, the *ABCD* column represents the Produces_Four module.

The following discussion shows how the information in Figure 2 helps to guide refactoring decisions. As we can see from the figure, slicing the original module into four separate modules for A, B, C, and D results in the lowest average coupling and cyclomatic complexity for the overall system. The average coupling and complexity of these four separate modules is lower than that of the original module, so breaking this module into four separate modules would be the best refactoring choice. The point is to lower the overall complexity of the sys-

tem. By replacing the Produces_Four module, which has high complexity and high coupling, with four new modules that have lower complexity and lower coupling, we can lower the average complexity and coupling for the system. We focus on the average of the metrics because we want to show that it will be easier to maintain the four new modules instead of one legacy module.

It could be the case, however, that organizational or management policies prevent you from selecting this option. For example, limits on the total number of modules in the system – or lower limits on the size of those modules – could preclude breaking the original module into four separate modules in our example. Effort should be expended to change such policies, especially when compliance will result in systems that are less maintainable than they could be. We also recognize, however, that some organizations will impose those policies regardless of the resulting impacts.

In this scenario, which refactoring option should you choose? In the following discussion, we assume that our policies constrain us to select exactly two modules in refactoring the original module.

As shown in Figure 2, the module for values CD has the same coupling and complexity as the modules for values AB, AC, and AD. All of these modules have lower coupling and complexity than the modules for BC and for BD. Selecting the module for AB along with the module for CD is a reasonable choice to replace the original module. The average coupling for this choice is 4 and the average complexity for this choice is 3. This option results in a lower average coupling and complexity for the overall system. It is also a better choice than selecting the module for AD along with the module for BC because the module for BC drives up the average coupling to 4.5. The values of the metrics for these program slices can help the software engineer select the best possible refactoring option that fits within the constraints placed on a software system.

Further analysis shows that the optimal choice in this situation is to choose the module for B along with the module for ACD. The average complexity for this choice remains at 3, but the average coupling is reduced to 3.5. This is a better choice than selecting the modules for AB and CD, since the average coupling and cohesion are less. Clearly, this is the best choice if the developer is constrained to selecting exactly two modules in the refactoring process.

This illustrates how refactoring decisions can be guided by using program slicing and the values of metrics of the resulting program slices.

Conclusion

Coupling, cohesion, and cyclomatic complexity have become accepted metrics for measuring the maintainability and quality of software systems. Refactoring can improve the quality of a system as measured by these metrics, but which refactoring choices should be implemented? We suggest using program slicing in conjunction with software metrics to guide the refactoring process. By slicing the software system on one or more variables, different refactoring options can be examined and evaluated using these metrics. The choices that program slicing provides can be sorted by the respective values of the metrics, and a design that provides the most beneficial metric values can be selected. It is the combination of program slicing and software metrics that guides the refactoring process.

"The return on investment in this refactoring process can be measured in lower error rates, fewer test cases per module, and increased overall understandability and maintainability."

A software system that has gone through this refactoring process has higher quality and is more maintainable. The return on investment in this refactoring process can be measured in lower error rates, fewer test cases per module, and increased overall understandability and maintainability. In both the design and maintenance phase, these advantages can be realized almost immediately.◆

References

1. Jones, Capers. Assessment and Control of Software Risks. Prentice Hall, 1994.
2. Fowler, Martin. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
3. Tahvildari, L., and K. Kontogiannis. "First International Workshop on Refactoring: Achievements, Challenges, and Effects." REFACE '03, Victoria, British Columbia, 13 Nov. 2003 <http://swen.uwaterloo.ca/~ltahvild/Publications/REFACE03.pdf>.
4. Fenton, Norman E., and Shari L Pfleeger. Software Metrics: A Rigorous and Practical Approach. 2nd ed. Boston: PWS Publishing Co., 1997.
5. McCabe, Thomas J. "A Complexity Measure." IEEE Transactions on Software Engineering 2 (1978): 308-20.
6. Yourdon, Edward, and Larry L. Constantine. Structured Design. 2nd ed. New York: Yourdon Press, 1978.
7. McConnell, Steve. Code Complete. Microsoft Press, 1993.
8. Stevens, Wayne, G. Meyers, and L. Constantine. "Structured Design." IBM Systems Journal 13.2 (May 1974): 115-39.
9. Bieman, James M., and Linda M. Ott. "Measuring Functional Cohesion." IEEE Transactions on Software Engineering 20 (1994): 644-57.
10. Selby, Richard W., and Victor R. Basili. "Analyzing Error-Prone System Structure." IEEE Transactions on Software Engineering 17 (1991): 141-52.
11. Gill, Geoffrey K., and Chris F. Kemerer. "Cyclomatic Complexity Density and Maintenance Productivity." IEEE Transactions on Software Engineering 17 (1991): 1284-88.
12. Simon, F., F. Steinbruckner, and C. Lewerentz. Metrics-Based Refactoring. Proc. of the European Conference on Software Maintenance and Reengineering, Mar. 2001.
13. Tahvildari, L., K. Kontogiannis, and J. Mylopoulos. "Quality-Driven Software Reengineering." Journal of Systems and Software 66.3 (June 2003): 225-239.
14. Tourwe, T., and T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. Proc. of the European Conference on Software Maintenance and Reengineering Mar. 2003.
15. Kataoka, Y., T. Imai, H. Andou, and T. Fukaya. A Quantitative Evaluation of Maintainability Enhancement By Refactoring. Proc. of the IEEE

Coming Events

August 14-17
CCCT Conference: Computing, Communications, and Control Technologies
Austin, TX
www.iiisci.org/ccct2004

August 19-20
2004 ACM-IEEE International Symposium on Empirical Software Engineering
Redondo Beach, CA
www.isese.org

August 23-27
International Conference on Practical Software Quality Techniques
PSQT 2004 North
Minneapolis, MN
www.qualityconferences.com

September 11-17
20th IEEE International Conference on Software Maintenance
Chicago, IL
www.cs.iit.edu/~icsm2004

September 13-16
Embedded Systems Conference
Boston, MA
www.esconline.com/boston

September 20-23
Software Development Best Practices
Boston, MA
www.sdexpo.com

September 24-26
IPSI 2004 Stockholm
Stockholm, Sweden
www.internetconferences.net

April 18-21, 2005
2005 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

International Conference of Software Maintenance, Oct. 2002.
16. Shepperd, M., and D. Ince. "Metrics, Outlier Analysis, and the Software Design Process." Information and Software Technology. Mar. 1989: 91-98.
17. Weiser, M. "Program Slicing" IEEE Transactions on Software Engineering SE-10 (4) (July 1984): 352-357.
18. Verbaere, Mathieu. "Program Slicing for Refactoring." Masters Thesis. University of Oxford, Sept. 2003 <http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/projects/nate/doc/MScThesis.pdf>.

About the Authors

Lt. Col Ricky Sward, Ph.D., U.S. Air Force, is an associate professor of Computer Science at the U.S. Air Force Academy. He is currently the deputy head for the Department of computer science and the course director for the senior-level two-semester Software Engineering capstone course. Sward has a doctorate in computer engineering from the Air Force Institute of Technology where he studied program slicing and reengineering of legacy code.

Department of Computer Science
2354 Fairchild DR
STE 6G101
USAF Academy, CO 80840
E-mail: ricky.sward@usafa.af.mil

A.T. Chamillard, Ph.D., is an assistant professor of Computer Science at the University of Colorado at Colorado Springs where he teaches the core Master of Engineering in software engineering courses. He also currently provides software engineering consulting services to a Department of Defense agency. Chamillard spent over six years as a project manager in the U.S. Air Force, and was also an associate professor of computer science at the U.S. Air Force Academy where he taught for six years. He has a doctorate in computer science from the University of Massachusetts, Amherst.

Computer Science Department
University of Colorado at Colorado Springs
1420 Austin Bluffs PKWY
Colorado Springs, CO 80933-7150
E-mail: chamillard@cs.uccs.edu

David A. Cook, Ph.D., is a senior research scientist at The AEgis Technologies Group, Inc., working as a verification, validation, and accreditation agent in the modeling and simulations area. He is currently supporting the Airborne Laser program and has more than 30 years experience in software development and management. He was formerly an associate professor of computer science at the U.S. Air Force Academy, a deputy department head of the Software Professional Development Program at the Air Force Institute of Technology, and a consultant at the U.S. Air Force Software Technology Support Center. Cook has published numerous articles on software-related topics. He has a doctorate in computer science from Texas A&M University.

AEgis Technologies Group, Inc.
6565 Americas PKWY NE
STE 975
Albuquerque, NM 87110
Phone: (505) 881-1003
Fax: (505) 881-5003
E-mail: dcook@aegistg.com