Software Remodeling: Improving Design and Implementation Quality

Using audits, metrics and refactoring in Borland® Together® ControlCenter™

A Borland White Paper

By Richard C. Gronback

January 2003

Borland®

Contents

Abstract

The manual reviewing of source code remains an integral part of most successful software projects. Pairing this manual process with automated techniques available through static source code-level analysis can markedly increase the success of these programs. This paper presents information on how the quality assurance audits and metrics available in the Borland® Together® ControlCenter™ can streamline the code review process while increasing the design and implementation quality of your software.

Introduction

Software Remodeling is a methodology for improving design and implementation quality using a combination of audits, metrics, and refactoring. The "subtle science and exact art" of software engineering necessitates formal code reviews within most development organizations. These code reviews are necessary as part of a quality assurance program.

A typical outcome of code reviews is the refactoring of source code. Refactoring aims to change the internal structure of software to make it easier to understand and less expensive to modify without changing its observable behavior [Fowl99]. This is not a new concept, but it is extremely important to the long-term success of software projects.

Recently, it has become stylish to develop software with agile methodologies. These methodologies focus on the production of working software rather than mountains of requirements specifications. Notably, the review of source code has carried over from more traditional processes, signifying its importance. In Extreme Programming (XP), code reviews are done continuously, along with integration and refactoring.

As a code base increases in size and complexity, it becomes unlikely that a manual review process, no matter how frequent, can uncover all issues with a high degree of accuracy. Automation is necessary for tasks that are particularly well suited to machine processing. With its extensive support for audits, metrics, and refactoring, ControlCenter is a tool that

helps organizations conduct more effective code reviews in a fraction of the time it takes to perform them in a "traditional" way.

Software inspection

Code reviews are just one part of a more broad-reaching inspection program. As defined by the IEEE Standard Glossary of Software Engineering Terminology, an inspection is a formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems [IEEE98].

The IEEE also defines the objective of software inspection. The Standard for Software Reviews and Audits states the objective is to detect and identify software element defects. This is a rigorous, formal peer examination that does the following: (1) Verifies that the software element(s) conforms to its specifications. (2) Verifies that the software element(s) satisfies applicable standards. (3) Identifies deviation from standards and specifications. (4) Collects software engineering data (for example defect and effort data). (5) Does not examine alternatives or stylistic issues [IEEE90].

Historically, the concept of software inspections began with Michael Fagan in the 1970s at IBM. Mr. Fagan applied industrial hardware statistical quality methods to software and created the Inspection technique [Gilb93]. This technique is also referred to as Fagan's Inspection. As part of this process, source code, requirements documents, test plans, and so on, were reviewed to improve the quality of not just the software but of the process itself. This paper focuses only on the source code review process.

A trend in the software industry is toward more "agile" approaches to software development. In fact, the recently written "Agile Manifesto" states that working software should be valued over comprehensive documentation [Beck01]. This would seem to imply that if reviews are to be done with an agile process, it is likely that source code would be the only artifact deemed worth reviewing. Taken to the extreme, XP provides a form of continuous code review, as development is always done in pairs. Other lightweight processes, such as Feature-Driven

Development (FDD), incorporate design and code reviews as part of their activities. So, it is clear that even if the processes used to develop software have become lighter, the review of source code is still regarded as a necessity.

Source code reviews

The majority of professional application developers understand and recognize the benefits of code reviews. If a review is done correctly, it not only improves the understandability, maintainability, and design of software, but it will also become an instrumental tool in training junior members of the team. XP fosters "continuous code review" through its use of pair programming. While many organizations may be unwilling to require programming to be performed in pairs, all can benefit from a reasonable code review program.

A "reasonable" program is key, as over-engineered code reviews can quickly become an exercise in frustration and futility—in addition to wasting time and paper. It can often be difficult to manage and focus code reviews to ensure their success. All too often, they result in lengthy philosophical discussions that typically have little practical foundation. A code review program needs to strike a delicate balance between providing value in the minds of the attendees and contributing to the overall success of the project.

Unfortunately, this balance is rarely found, and code reviews are left until the end of the project, if they are done at all. Given that many projects do not have "running" software until the last moment, the chances of effective code reviews being performed at critical periods in the project become increasingly slim. That being the case, code reviews often do not receive thorough testing, performance tuning, and documentation.

Another detrimental aspect of some code reviews is the lack of focus on business logic. Unless unit tests are written to perfection and exhaustively test business functionality, these areas need to be addressed in code reviews. Automated audits and metrics are great at pointing out qualities of syntax and design, but they are not capable of determining the correctness of business logic. Currently, human inspection and generation of unit test logic is required to verify correctness. The automation of detectible audits and metrics therefore allows for more time to be spent on this critical issue of code reviews.

Code review categories

Source code reviews fall into four major areas: coding standard/best practice violations, design-related violations, business logic reviews, and performance-related violations.

Audits are static checks of program source code that compare the actual usage of the language to recommended best practices. For example, audits can check for everything from poor exception handling techniques to the inappropriate use of operators such as +=. Most tools have a facility for the formatting of source code to specified standards, thus taking care of curly brace-style issues. More complex analysis of source code identifies issues related to the underlying programming language best practices. Audit violations are easily detected by human code reviews, although the accuracy and breadth of these checks deteriorate as the code base grows.

Metrics are a less well-defined part of code reviews; this is where "subtle science" comes into play. While "curly brace wars" have accounted for untold hours of lost productivity, they pale in comparison to design review discussions based upon gut feel and supposition. Metric data can help guide these discussions by providing solid information regarding certain aspects of software design. Concepts such as cohesion, coupling, encapsulation, inheritance, and complexity roll off the tongues of object-oriented practitioners, while few have utilized any form of quantitative measurement regarding these principles in practice.

Making a judgment with respect to design without solid information is comparable to making performance improvements without profiling data. As metric data is not foolproof, the understanding of what is being measured and how to interpret the results in the application is key. The problem with the collection and analysis of metric data is that it can be extremely time consuming; thus, it is necessary to leverage automated methods for this data collection and display. Furthermore, as the rigor with which metrics are developed and analyzed improves, the potential gains for an organization that leverages them increases. Currently, due to the relatively immature state of metrics and their usage by the development community, the benefits of a metrics program have yet to be realized by the many corporate development organizations.

A major focus of human code reviews should be the inspection of business logic. After all, it is the reason software is written. Even the partial automation of code reviews means being able to spend more time validating the correctness of business logic. Ultimately, tests can validate this aspect, but it is up to a human to identify the need for testing and to write the tests. Until a means by which automated interpretation and implementation of requirements is achieved, this will be the case.

Performance reviews should involve data from profiling a running application. This data, together with audit and metric reviews, can lead to refactorings that target improved application performance. The analysis of profiling data is not addressed in this paper.

Benefits of automation

Clearly, unless you have a photographic memory, the task of reviewing every line of code for every possible known audit and metric is an unrealistic expectation. Tools can be extremely accurate at identifying audit violations in source code at a fraction of the time an error-prone human is capable of reading. However, a computer is not likely to correctly analyze business logic and point out missing requirements or insufficient test cases. Test teams need to seek to strike a balance between business logic-savvy humans and the efficient, precise computational analysis of source code by machines.

As an example of the gains that can be achieved by using automated tools to identify audit violations, four mentors at Borland were given 15 minutes to analyze a small set of Java™ source code. The code had been "engineered" to contain various well-known violations. The results of this unscientific experiment reveal an obvious conclusion; human reviews alone are insufficient. The mentors found an average of 21 violations in the 15 minutes allotted, while Together ControlCenter identified 150 violations in approximately 2 seconds using the complete set of Java audits.

These results illustrate the possible gains, although processing a full-scale production application is even more telling. In one case, nearly 200,000 lines of source code from over 1200 classes were analyzed with Together ControlCenter, resulting in 32,443 violations from the default set of audits.

So, what does this mean? Running audits and metrics on completed applications can be revealing and possibly guide some much-needed refactoring. However, running them in parallel with development is preferred. Running audits frequently during development helps to enforce established best practices, while routine metric analysis can help to maintain design issues in check throughout the development of an application.

The inclusion of audits and metrics in the development process goes hand in hand, not only with the agile methodologies that stress continuous integration, testing, and refactoring, but also with any process that involves best practices in coding standards and object-oriented design concepts. The automated audits and metrics complement comprehensive unit testing suites.

Software audits

Source code formatting standards are found in most organizations. Requiring legible source code is important on projects where communication needs to be facilitated. However, there are many categories of audits including coding style, superfluous content, and errors for a particular computer language. Many of the audits, found in books such as “Effective Java” by Joshua Bloch, are easily automated with tools such as Together ControlCenter. Leveraging the auditing capabilities of Together ControlCenter as part of your routine development tasks can improve the quality of your developers as well as your code.

As mentioned above, books can be a valuable source of information regarding the ever-changing landscape of best practices for software coding. Additionally, your own experience and the experience of others in your group increase the knowledge of best practices. One of the primary reasons for code reviews is to increase the knowledge of junior developers. In this case, automated auditing tools complement the shared experiences of the team. A tool, thereby freeing up your senior team members from answering those same questions time and again, can pick up many of the more “obvious” violations.

Lastly, outside help may come into consideration, particularly if the entire team has recently transitioned to a new language, and there is no resident “language lawyer” to guide the way.

Sample audit

Together ControlCenter comes with more than 100 source code audits for Java and nearly half as many for C++. These range from the simple (for example, Order of Appearance of Modifiers) to the more complex and specific (for example, Managing Threads in an EJB™). The tool also gives you the ability to write your own custom audits using the Borland OpenAPI.

The remainder of this section looks at one specific Java audit. The audit selected for review here is also found as Item 31 ("Avoid float and double if exact answers are required") in Effective Java. In Together ControlCenter, the audit is named CFPT—Comparing Floating Point Types—and is found in the Possible Errors category.

Floating-point types are intended for use in scientific or mathematical equations where numbers of large magnitude are calculated. They are not intended for use in comparing small values, particularly those involving currency values. In this case, rounding in calculations may cause errors, particularly when dealing with values in the negative powers of 10. For example, if you declare two doubles d1 = 1.03 and d2 = 1.13 and compare d1 – d2 to the expected answer of –0.1, you find that they are unequal using the == operator. Additionally, when overriding the equals() method, comparisons to an object's floating point type fields should for the same reasons use the Double.doubleToLongBits() and Float.floatToIntBits() methods for double and float fields, respectively.

The prospect of scanning through all of an application's source code to find potential errors such as this illustrates why automation is required. Because the audit functionality in Together ControlCenter is incorporated into the development environment, routinely running audits during development provides the greatest long-term benefit.

Case studies

The increasing popularity of open source development provides ample fodder for audit and metric case studies. For example, the Sun® Code Conventions for Java audit set was run on the Blueprint Program PetStore 1.3 source code. A total of 5258 violations were detected, 851

of which fell into the Together ControlCenter default set, which does not count many of the "trivial" violations required by the Sun Code Conventions for Java set.

Apache™ Ant version 1.4 was also analyzed, resulting in 2752 audit violations from the default set, including 442 of "high" severity.

While few would argue that Ant is not a successful software project, it is clear that many issues within the source code warrant some attention. It is unlikely that anyone will fix even the 442 "high" severity hits at this point, which stresses the importance of using audits from the beginning of a project, if possible. Primarily, the issues of maintainability, readability, and bug identification are the aim of audit-hit resolutions.

While Ant is currently functional software, its readability and maintainability going forward will be increasingly important. This is especially true in open source projects where the source code itself may be the only form of documentation available.

Summary

To summarize, audits focus on conformance to source code standards and on best practices. These are typically specific to the programming language syntax, focusing on readability, maintainability, robustness, and performance issues. Given the evolutionary nature of these issues, it is nearly impossible to keep a large code base in conformance unless an automated means is available. Enabling developers to identify these issues allows them to gain experience faster while improving the quality of their code and boosting productivity.

Software metrics

Fundamentally, metrics involve counting or measuring. Based upon what is being measured, a relative scale can be used to analyze the results, indicating the condition of source code. Whereas audits focus on mostly syntactic analysis, metrics focus more on design-level parameters. These range from the obvious to the complex and the obtuse, depending on your point of view. The goal is to gather data that may later be analyzed to identify flaws. The

reverse may also be stated; in which case, metrics may provide some assurance that an underlying body of code is without serious flaw. Given the fact that metrics are currently found in the "subtle science" category, these statements are impossible to prove. Fortunately, there is increasing rigor being applied to the "strict art" of metric analysis, with the hope being that someday software quality will be mathematically provable.

Until that day, it's important to understand a bit of the theory behind these metrics and how to best put to use what they show. This section begins with more information about the state of today's metrics, followed by some basic metrics and explanations of using them in conjunction with code reviews.

There are many compelling reasons to use what metrics have to offer software development. Object-oriented programming has matured to the point where it is commonly accepted as the favored paradigm for developing software. With this shift in the way software is developed comes a new suite of metrics aimed at validating the quality of object-oriented designs. The goal is improving the process of developing software and finding automated ways of analyzing the results. Naturally, the analysis provided by metrics focuses on well-known principles of object-oriented programming. These include cohesion, coupling, complexity, inheritance, polymorphism, encapsulation, and so on. Therefore, the intelligent application of metrics to existing software necessitates a firm understanding of these principles so that refactorings based upon them improve the underlying design.

Knowing what to look for requires knowledge of metric theory, object-oriented principles, and your domain. However, this knowledge does not help if you have to manually gather metric data on a project of any size. Tools are of paramount importance to the success of a metrics program being instituted at your organization.

Internal versus external metrics

Metrics can be broken down into two categories: internal and external [Hend93]. Internal metrics are objective and focus on the internal characteristics of source code, which are measured with accuracy. Internal metrics inform the developer of characteristics such as size, coupling, cohesion, and complexity. External metrics are stochastically based measures that

attempt to assess the external characteristics of source code. These are less tangible concepts such as cognitive complexity, maintainability, quality, and understandability. Whether a link exists between internal and external metrics has yet to be determined with certainty.

The focus of this paper is limited to internal metrics. It will take some time and data gathering before science can be applied to establish the link between internal and external metrics. If software development is to become a true engineering discipline, metrics need to be more widely used and scrutinized. Only after evaluation and feedback occurs in a large scale will the accuracy likely improve. This should lead to the future possibility of a functional relationship between internal measures and external characteristics.

A disclaimer

There is no "silver bullet" metric suite. You can use metrics to guide projects, but be careful not to let them paralyze a project when developers overanalyze. There are also reasons why automated tools can produce misleading results. This is often due to particular language features that cannot be accounted for in a general sense. For example, the use of reflection in the Java programming language makes static analysis of source code problematic. In this case, reflective use of a class and its methods will appear to static analysis tools as the mere utilization of the Reflection API and not the actual class involved. Metrics indicate false dependencies when they encounter the use of this feature.

The use of metrics fundamentally boils down to understanding what is being measured, how it is being measured, and how you interpret any results. This is why, at present, a human mind is still necessary to provide interpretation [Mari98]. Don't be too quick to jump to conclusions based upon metric results. Refactoring code should only be undertaken after considering the astute advice found in "Refactoring: Improving the Design of Existing Code by Martin Fowler" [Fowl99].

Finally, a set of etiquette rules should be part of every metric program; although it is likely they will be disregarded at one point or another. In general, the plan should be published and the importance of not assigning blame to individuals should be stressed. Metrics should not be used in a way that inhibits the process of producing working software.

Sample metrics

Together ControlCenter contains 55 quality assurance metrics for use in analyzing your Java and C++ source code. These cover a range of categories, including cohesion, complexity, coupling, polymorphism, inheritance, and encapsulation. Most of the principles associated with the promise of object-oriented analysis and design can be formulated into a metric and run on source code.

In addition to running metrics as defined by many authors in academia and industry, custom metrics can be written using the Borland OpenAPI. All metric results can be viewed in a variety of ways for analysis, including metrics run on audit results. As important as it is to run metrics in the analysis of your design, it is perhaps more important to spot trends in metric values as your project progresses. Export capabilities for spreadsheet analysis, in addition to a comparison display, are available in Together ControlCenter.

This section looks at some of the "core" metrics that are available. With the exception of Cyclomatic Complexity [McCa76], these object-oriented metrics were introduced by Chidamber-Kemerer in the early 1990s. Since the introduction of the CK metric suite [Chid94], there have been many refinements and additions to the original six metrics provided as part of the suite. To describe these six in detail, let alone the rest, would take a considerable amount of time. Therefore, this paper only touches on them to provide some background.

Depth of Inheritance Hierarchy (DOIH)

Inheritance is one of the features of object-oriented programming that was much touted and then greatly abused. It has since fallen somewhat out of favor, largely replaced by composition and interfaced-based design strategies [Coad99]. Where it is appropriately used, inheritance is a powerful design feature. Like all things, moderation is the key. Deep inheritance hierarchies can lead to code fragility with increased complexity and behavioral unpredictability.

An easy metric to comprehend, DOIH indicates the number of classes that are parents of a class, with the top-most class having a count of one (not zero as some computer scientists would think) [Chid94].

In ControlCenter, DOIH does not include the implementation of interfaces, while interface extension is counted. Figure 1 illustrates an inheritance hierarchy where the DOIH value for the AppException class is 5, while the DOIH value for RuntimeException is 4, and so on.

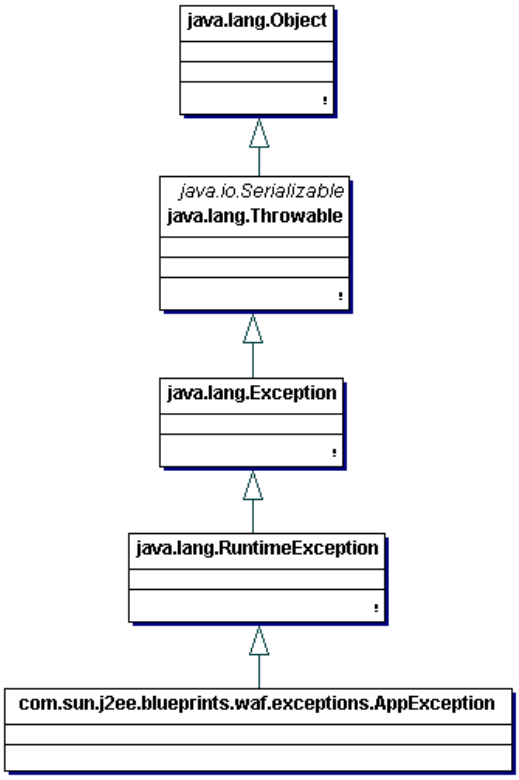

Figure 1. *Depth of Inheritance Hierarchy (DOIH)*

Number of Child Classes (NOCC)

For a given class, the number of classes that inherit from it is referred to by the metric Number of Child Classes (NOCC) [Chid94]. With increasing values of NOCC, the potential for reuse increases, while testing difficulty and potential misuse of inheritance also increase. Generally speaking, a high level of NOCC may indicate the improper abstraction of the class.

In the Java programming language, implemented interfaces count when calculating NOCC. For example, if an interface is implemented by a class, which has six subclasses, the NOCC value for the interface is seven, while the implementing class has a NOCC value of 6.

Figure 2 indicates how NOCC values are calculated. Starting at the bottom, each of the Event classes has no children and therefore receives a NOCC value of 0. Each of the 6 *Event classes extend the EventSupport class, giving it a NOCC value of 6. Finally, at the top of the hierarchy is the interface Event with its implementation (children) classes totaling 7.

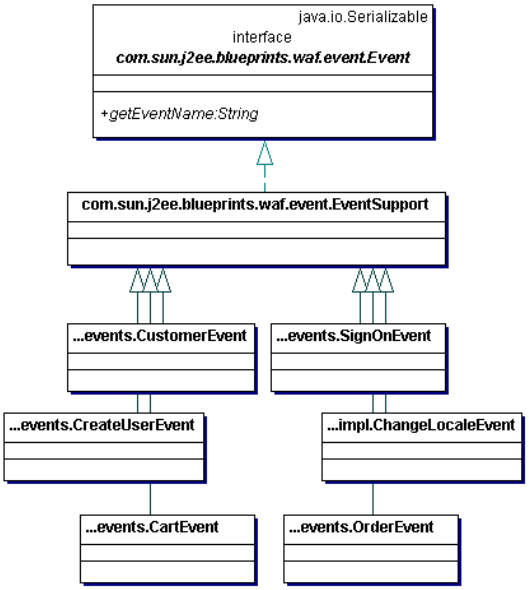

***Figure 2**. Number Of Child Classes (NOCC)*

Coupling Between Objects (CBO)

Object-oriented design necessitates interaction between objects. Excessive interaction between an object of one class and many objects of other classes may be detrimental to the

modularity, maintenance, and testing of a system. Coupling Between Object (CBO) counts the number of other classes to which a class is coupled, except for inherited classes, java.lang.* classes, and primitive types [Chid94]. A decrease in the modularity of a class can be expected with high values of CBO. However, some objects necessarily have a high degree of coupling with other objects. In the case of factories, controllers, and some modern user interface frameworks, you should expect a higher value for CBO.

An important note about the implementation of the CBO metric is that it only counts each type used in a class once. Multiple usages of a class as referenced, thrown as an exception, or passed as a formal parameter are only counted at their first occurrence. As mentioned, parent classes do not count, as is the case for implemented interfaces. There is an implicit high degree of coupling in these cases.

The sample code in Figure 3 indicates a class that is coupled to a total of four other classes. From the class declaration, you get 0, as implementation and extension do not count as couplings in the CBO metric. The class attributes int number and String string do not count because they are considered primitives. The List list attribute does count, but only once, even though a List type is passed as an argument to method one(). The Map m passed as the second argument counts, creating the second coupling. Finally, in the method two(), the throws AppException declaration counts as a coupling, in addition to the local File file variable. Again, the List type has already been counted.

```
public class CBO
  extends Base
  implements Metric {

  private int number;
  private List list;
  private String string;

  public void one(List l, Map m) {
    list = l;
  }


  private List two()
```

```
   throws AppException {
   File file;
   //...
   return new List();
  }
 }
```

Figure 3. *Coupling Between Objects*

Response for Class (RFC)

The number of methods, internal and external, available to a class is known as the Response for Class (RFC) metric [Chid94]. The enumeration of methods belonging to a class and its parent, as well as those called on other classes, indicates a degree of complexity for the class.

RFC differs from CBO in that a class does not need to use a large number of remote objects to have a high RFC value. Consider a class with a large number of exposed methods that is used by a client class extensively. In this case, the client class RFC value would be high while its CBO could be one. Distinguishing CBO as a coupling measure and RFC as a complexity measure is important. Many metrics have relationships and/or similarities to other metrics.

Because RFC is directly related to complexity, the ability to test, debug, and maintain a class increases with an increase in RFC. In the calculation of RFC, inherited methods count, but overridden methods do not. This makes sense, because only one method of a particular signature is available to an object the class. Also, only one level of depth is counted for remote method invocations.

RFC is the sum of local and remote methods available to a class. In Figure 4, class A has a parent class with six methods. Class A overrides four of them and adds none. This leaves six as the total number of local methods. Looking into the methods of class A, you count seven distinct calls to methods of classes B and C. However, the cardinality of the union of these methods gives us six, as f1() on class B is called from both f1() and f2() in class A. Thus, RFC = 6 + 6 = 12 for class A. Note that the methods available in the parent class (those not overridden) may use remote methods. These are not counted in the subclass RFC calculation.

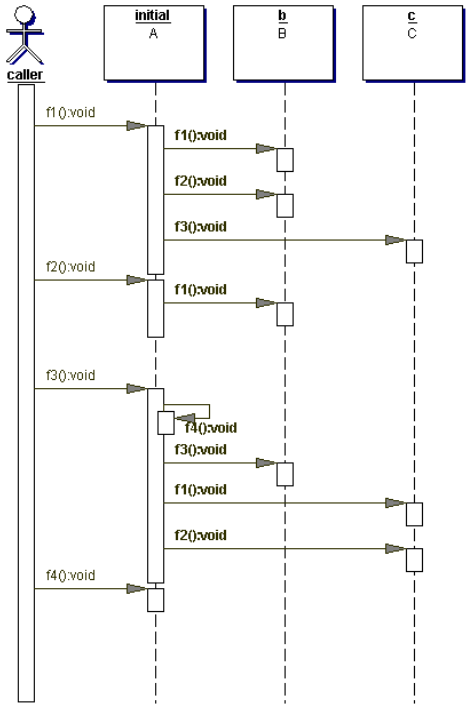

Figure 4. *Response for Class (RFC)*

Cyclomatic Complexity (CC)

Cyclomatic Complexity (CC) counts the number of possible paths through a method, as indicated by the conditional statements it contains [McCa76]. In other words, using if, for, and while statements as path indicators counts the total number of paths through the body of a method. Consider a method represented as a strongly connected, Directed Acyclic Graph (DAG). The formula for CC equals the number of edges *e*, minus the number of nodes *n*, plus two times the number of components *p*

$$CC = V(G) = e - n + 2p$$

It is important to realize that CC does not count into called internal or external methods. Optionally, Together ControlCenter can count case statements as branches.

As expected, the more complex a method is, the more difficult it is to test and debug. The value of CC for a method can be used as an indicator for the minimum number of test cases needed. However, the Nejmeh NPATH metric (1988) is a better indicator for the number of test cases required [Hend95].

To illustrate CC, consider the following method in Figure 5:

```
public void one(){
 if(true) {
  while(false) {
   two();
  }
 } else {
  for(int i=0;i<10;i++) {
   two();
  }
 }
}
```

Figure 5. *Cyclomatic Complexity (CC)*

In this method, there is an if statement containing a while loop with a for loop in the else portion. Considering the distinct paths through this method, it is obvious there are four, as the simplified illustration in Figure 6 shows CC = 8 – 6 + 2(1) = 4

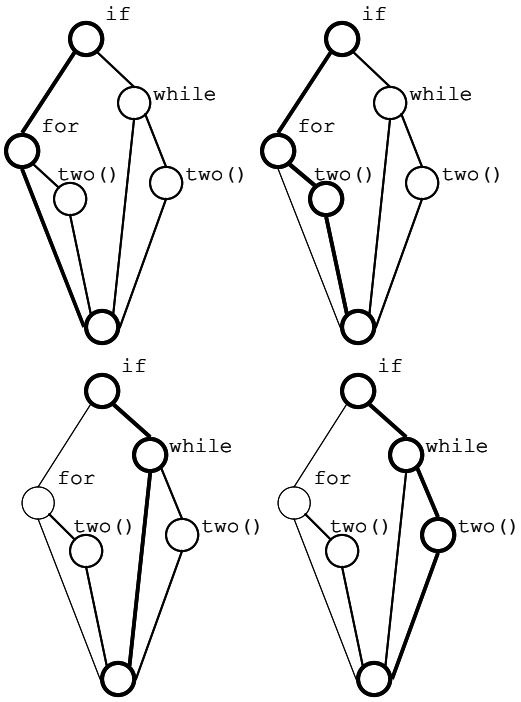

Figure 6. *Cyclomatic Complexity (CC)*

Weighted Methods Per Class (WMPC)

If you were to sum the values for CC for every method in a class, you would have the Weighted Methods Per Class (WMPC) metric value [Chid94]. Alternatively, a simple count of the methods and their formal parameters may provide the WMPC value. In Together ControlCenter, WMPC1 implements the former, while WMPC2 implements the latter. In equation form, consider n to be the number of methods and c to be a measure of the method's complexity. Therefore,

$$\mathrm{WMPC} = \sum_{i=1}^{n} c_i$$

As the WMPC value for a class increases, the time and effort for development and maintenance increases. Also, the potential reuse of the class may decrease as its WMPC value

increases. In measuring WMPC, inherited methods are not included, while getter, setter, and private methods are included.

Because this is a simple metric to conceptualize, it is important to see the value in using both WMPC1 and WMPC2 to better understand a class. For example, a class with one extremely long and complex method yields a high value for WMPC1 but a small value for WMPC2. Conversely, a class with many small methods that are not complex yields similar results for WMPC1 and WMPC2. Finally, a data or value object class with getters and setters for each attribute results in a higher value for WMPC2 than WMPC1.

As this paper explains later, there is a significant amount of information regarding a class that can be gleaned from a small sampling of metrics, particularly when displayed visually.

Lack of Cohesion of Methods (LOCOM)

As the most complex metric in the default set, Lack of Cohesion of Methods (LOCOM) measures the dissimilarity of methods in a class by attributes [Chid94]. Fundamentally, LOCOM indicates the cohesiveness of the class, although the values increase as the cohesiveness decreases.

LOCOM computes a measure of similarity for a number of methods m accessing a set of attributes Aj, where a is the number of attributes. The calculation of this metric alone should convince you that manual means of gathering measurements for metric analysis is not feasible.

$$LOCOM = \frac{\left(\frac{1}{a}\sum_{i=1}^{a}\mu(A_j)\right) - m}{1 - m}$$

Consider an example of a class with three attributes and two methods. The first method accesses two of the attributes while the second method accesses the third attribute. This is clearly a class lacking in cohesion, as verified by the LOCOM result of 100 and seen in Figure 7.

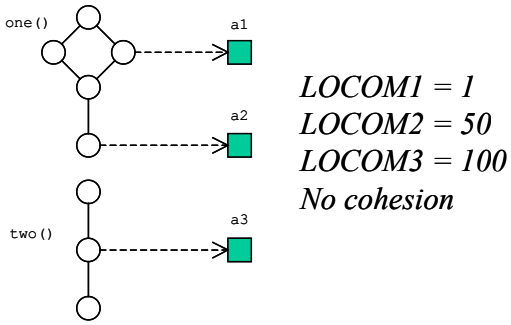

***Figure** 7. Lack Of Cohesion Of Methods (LOCOM)*

Suppose the class is modified such that the second method accesses all three attributes. Now, the LOCOM value is 34, indicating that each of the methods accesses 66% of the attributes. If the first method were also to access all three attributes, the value for LOCOM would drop to zero, indicating "complete" cohesion.

Notice the values for LOCOM1. In the first case, its value is 1 while it dropped to 0 for the second case. The main problem with LOCOM1 is the ambiguity associated with a value of zero, and the fact that it is possible to achieve the same result for classes that are very dissimilar from a cohesion standpoint. This ambiguity and other issues are discussed at length in "Object-Oriented Metrics: Measures of Complexity," by Brian Henderson-Sellers [Hend95].

With an increase in LOCOM, the encapsulation characteristics of the class are brought into question. Also, the complexity increases with a decreasing reuse potential for the class. This makes intuitive sense, as a class that has methods operating on separate attributes should bring the Extract Class refactoring to mind.

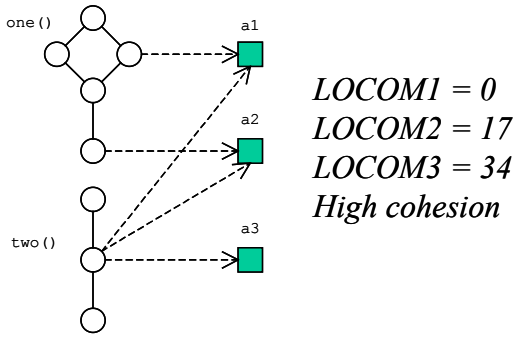

Figure 8. *Lack Of Cohesion Of Methods*

Note that Together ControlCenter multiplies LOCOM2/3 values by 100 to give a range of 0–100. Also, in LOCOM3 (the preferred variant), if there are three attributes and two methods with both methods accessing the same two of the three attributes, this results in a "perfect cohesion" of zero. In other words, it's the relationship between the methods with respect to their use of attributes that counts, not necessarily the fact that all attributes are fully accessed.

Metric interpretation

A collection of metrics data generated by static source code analysis is not too valuable in raw form. Just as the UML™ provides for the visualization of your design in the form of class diagrams, for example, metric results are best displayed graphically for interpretation. This section describes how to visualize metrics in bar, Kiviat, and distribution graphs. Also, because it is important to identify trends in metric results over a period of time, this section explains a method for comparing one set to another.

Bar graph

As the simplest form of display, the bar graph needs little explanation. In Together ControlCenter, a red bar indicates a metric result that exceeds the high limit indicated by a vertical red line (Figure 9). A green bar indicates a metric result that falls below the upper limit and above the low limit. A blue bar indicates a metric result that falls below the low limit.

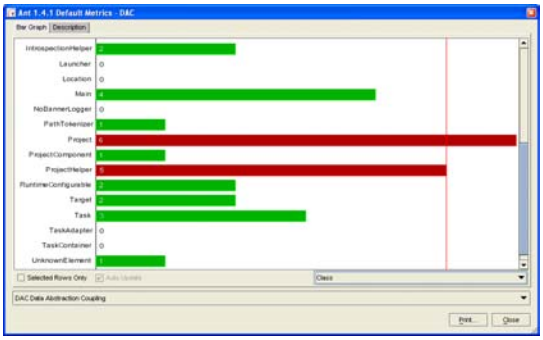

Figure 9. *Bar graph*

Kiviat graph

A Kiviat graph provides a powerful visualization of several metrics at once, each represented as an axis emanating from a central point (Figure 10). Each metric axis is scaled such that its upper limit is normalized, forming a red circle at some distance from the center. This allows for a quick scan of the results for the chosen class or package, making it immediately clear which fall outside their respective limits.

More information regarding the use of Kiviat graphs for class characterization and refactoring is presented later in this paper.

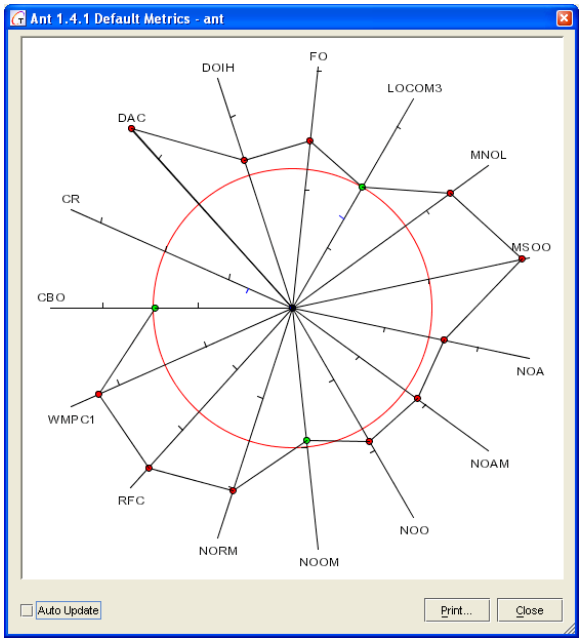

Figure 10. *Kiviat graph*

Distribution graph

A distribution graph provides information about how a particular metric result is distributed over its range in terms of the percentage of hits within the range (Figure 11). In other words, as the X-axis represents the metric value range, the Y-axis represents the percentage of examined elements that fall at or below that metric value. This distribution line always grows from 0 to 100% over the selected range of the metric limits.

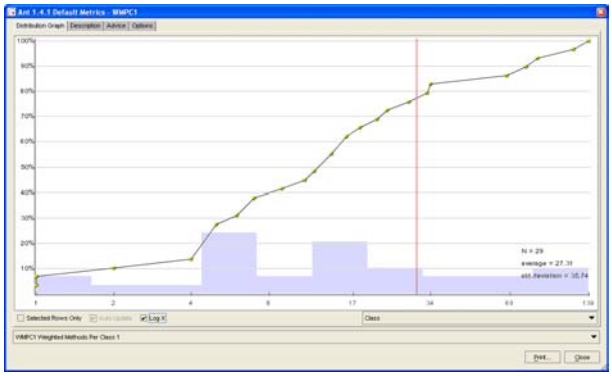

***Figure 11**. Distribution graph*

In Together ControlCenter, this line graph is also displayable as a logarithmic function of the X-axis. A greater slope value for the graph within a given range indicates a higher concentration of elements whose metric values fall into that range. A gently sloping graph indicates an even distribution of metric results for the given range. To aid in the recognition of the high-density ranges, a shaded histogram is overlaid on the graph. Finally, the number (N) of entities analyzed, a simple average, and standard deviation are provided.

Trend analysis

To spot increases, decreases, and new additions to your project's metric analysis, compare one set of results with another. In Together ControlCenter, it is possible to output metric results in several ways. One way is as a loadable format. With this capability, you can load previous results and overlay current results in a color-coded table view (Figure 12). In this case, a red block indicates an increase in value, and a blue block indicates a decrease in value. For examined classes that may be missing from the previous, gray shading is used. To compare actual numbers, you can as you hover over each cell to display a tooltip.

Exporting metric results to a delimited file may be more useful in spotting trends. Once you import these files into a spreadsheet program, you can aggregate them and use them to display metric trends over the life of a project.

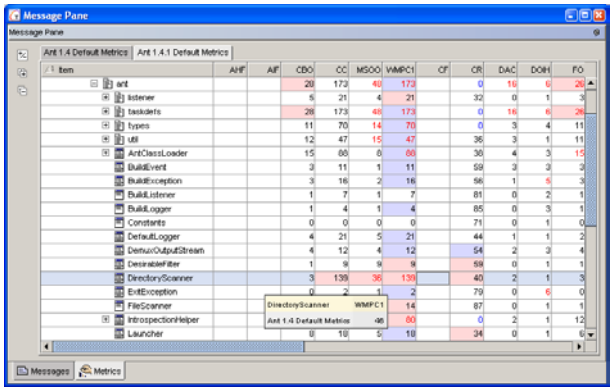

***Figure 12**. Metric comparison*

Summary

Metrics are indeed "deep and muddy waters" [Hend95]. A firm grasp of object-oriented design concepts and an understanding of what a metric is designed to detect are key components in a successful metric program. If you are new to metrics, it is best to start with a small subset, such as those described here, and not get too caught up in the details. In other words, take an "agile" approach to implementing a metrics program; focusing on the results you can obtain by using them effectively and not on using them to fulfill a process requirement.

Together with audits, metrics usage can provide significant long-term value to the underlying quality of your source code and the design it implements. As with audits, the collection of metric data must be automated to provide value.

The remainder of this paper focuses on how audits and metrics, in particular, can lead to refactorings in your source code to improve its design and implementation quality.

Refactoring

Refactoring is one of the latest buzzwords conversations about software development. Made famous by a book of the same title [Fowl99], it is a term for what has always occurred during the development of software. Basically, refactoring involves changing source code in a way that does not change its behavior but that improves its structure. The goal is to more easily understand and maintain software through these usually small changes in its design and implementation.

Refactoring support in development tools is now commonplace. It is much more likely that a developer will perform a refactoring if all affected sections of source code can be presented ahead of time. However, other than the daily, smaller focused refactorings, how does a developer know what to refactor on a large, somewhat unfamiliar code base? Audits and metrics drive refactorings. An automated, easy-to-use environment where these complementary activities can take place is necessary. Together ControlCenter provides such an environment.

Metric-driven refactoring

Refactoring can be a double-edged sword. Be careful not to "bite off more than you can chew," so to speak, while refactoring a code base. This goes hand-in-hand with having unit tests to run before and after a refactoring, thereby verifying that the functionality of the system is preserved. With this, it may be beneficial to run audits and metrics before and after refactoring to see the impact. This helps to check that you're moving in the right direction.

The process can work as follows:

1. Run unit tests, audits, and formatting tools.
2. Run and analyze metric results on your code.
3. Navigate to the offending areas, looking for problem areas.
4. Refactor the code, keeping design patterns in mind to potentially apply.
5. Rerun the audits, metrics, and unit tests.

In Together ControlCenter, you can save your metric selections and limits for running again later. Using this functionality, it is also possible to save configurations of metrics that are preconfigured to point out potential refactorings. For example, the Extract Class refactoring may be driven by a combination of metrics that point out high couplings (CBO), lack of cohesion (LOCOM3), number of attributes (NOA), and so on. Another example is using the metrics Maximum Size Of Operation (MSOO), Response For Class (RFC), Maximum Number Of Levels (MNOL), Cyclomatic Complexity (CC), Weighted Methods Per Class (WMPC1 & WMPC2), and so on, to identify potential Extract Method refactorings.

This technique is most effective when cleaning up a large code base, where there are known issues and metrics may aid in understanding the problems. On a smaller project, a custom set of metrics can be configured with highly restrictive limits, so that as a project continues, the quality can be more accurately maintained. It's important to keep tabs on the quality of your source before you are too far outside the limit so as to preclude "safe" refactoring.

Audits may also be run to target certain refactorings. For example, to identify the Reverse a Conditional refactoring, you can run the Negation Operator on 'if' Statement (NOIS) audit. Running audits before metrics can be key, as some metric results are altered based upon the running and correction of certain audit violations. For example, Unused Private Class Member (UPCM) may identify an unused class attribute, while Unused Local Variable or Formal Parameter (ULVFP) may identify an unused local or formal parameter. In these cases, the metrics WMPC2 and NOA are affected.

Finally, the formatting of source code to your company standards can make it more readable for refactorings but may also impact the result of certain metrics, such as Lines of Code (LOC), Comment Ratio (CR), and True Comment Ratio (TCR). As an alternative to LOC, consider using the Halstead Program Length (HPLen) metric, as it is unaffected by format.

Metric-driven refactoring example

Once again, open source projects and their published historical code bases make for interesting case studies. Using Tomcat 4.0.3 as an example, this paper will demonstrate metric-driven refactorings with Together ControlCenter.

Running the default set of metrics on the Tomcat code base reveals a number of violations. Initially, it is important to get a clear picture of what an overall project Kiviat graph is telling you about the "health" of the underlying code (Figure 13).

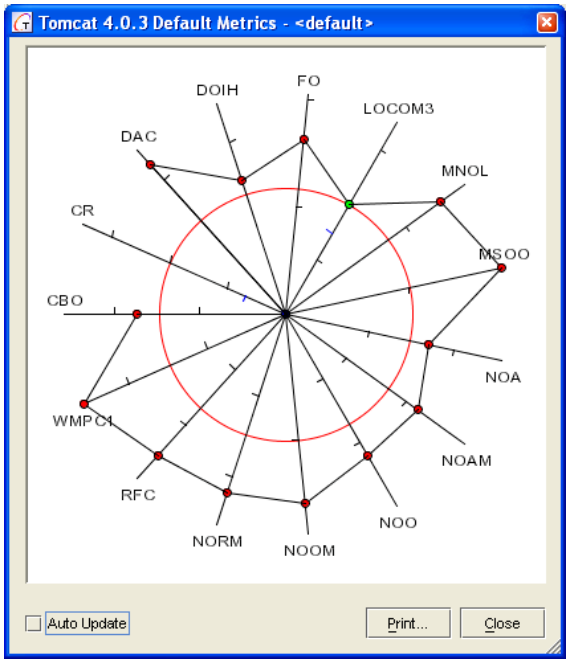

***Figure 13**. Tomcat Project-Level Kiviat*

Be careful to interpret this view correctly. This Kiviat is displaying an entire code base for Tomcat, with metric values displayed according to their aggregation and granularity settings with package-level limits. At this level, you're typically viewing the maximum values of each

metric for each class in the project. This is true as long as the metrics are set to an aggregation of maximum and a granularity of class, as is often the case. It is highly probable that your project will yield a similar looking Kiviat graph at this level.

Looking at the overall view of an entire project, it is likely that you will tend to use a few favorite metrics. Take a look at the metrics MSOO and WMPC1. The MSOO metric is just the maximum CC value for the methods in a class. Also, recall that WMPC1 is a summation of the CC values for all the methods in a class. In this case, the MSOO value for all the classes in Tomcat is 55, while its limit is set at 10. The value for WMPC1 is 377, while its limit is set at 30.

Remember, this view does not tell you that a single class possesses both of these characteristics—only that these maximums are found within the project.

Typically, those classes with large operations have correspondingly large complexities. Drilling down through the Tomcat hierarchy with MSOO as the guide leads you to the class WebdavServlet. Here, the highest value for MSOO in Tomcat does not correspond to the highest WMPC1 value. However, it is still rather high at 262. What this means is that while the class WebdavServlet has the longest operation (as measured by Cyclomatic Complexity), the sum of the CC values for all methods in the class is not the highest in the project; that honor belongs to the class StandardContext, whose WMPC1 value is 377. (For the curious, its MSOO value is 26.)

Prior to attempting any refactorings, it is important to run the JUnit tests, audits, and code formatter on the class. There is no JUnit test case, so you can overlook that step in this case (this is not recommended). Running the default audits for the WebdavServlet reveals two Unused Private Class Member (UPCM) hits. You can automatically fix these audit violations with Together ControlCenter. Finally, reformat the code to make it the most readable.

Now that you've identified the class with the largest MSOO, take a look at the class with the refactoring Extract Method in mind (Figure 14). Running the aforementioned set of metrics aimed at the Extract Method refactoring, you can see that the doLock() method is extremely long and gives the opportunity for several refactorings (Figure 15). In particular, the last

commented block of code before the method return typifies a problem area. This block can be more easily refactored into its own method named writeResponse() (Figure 16).

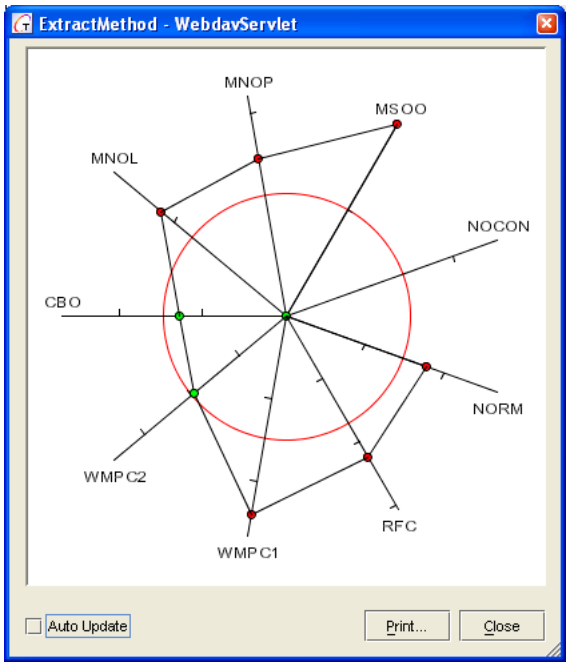

***Figure 14**. Extract Method (WebdavServlet)*

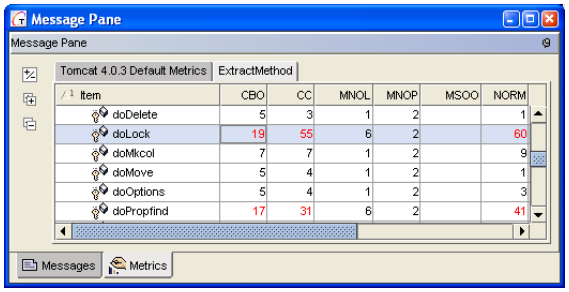

***Figure 15**. doLock() Metric Results*

Another interesting exercise with the Kiviat graph is to scroll through the results with "Auto update" selected. In the WebdavServlet, two operations immediately stand out when doing this: parseLockNullProperties() and parseProperties(). In fact, the shapes of the graphs are nearly identical. Both include violations for NORM, MNOP, MNOL, and CC (Figure 17).

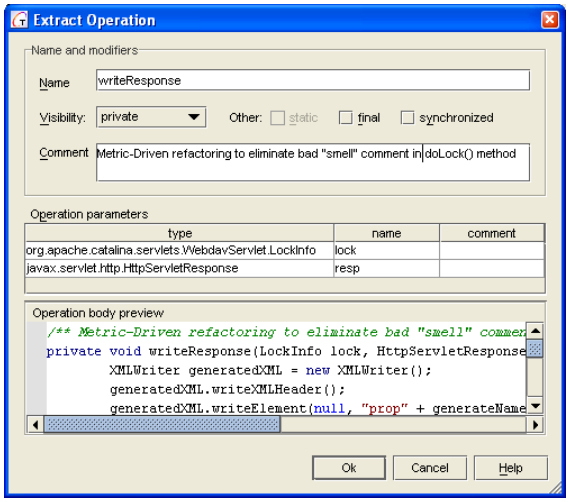

***Figure 16**. Extracting writeResponse()*

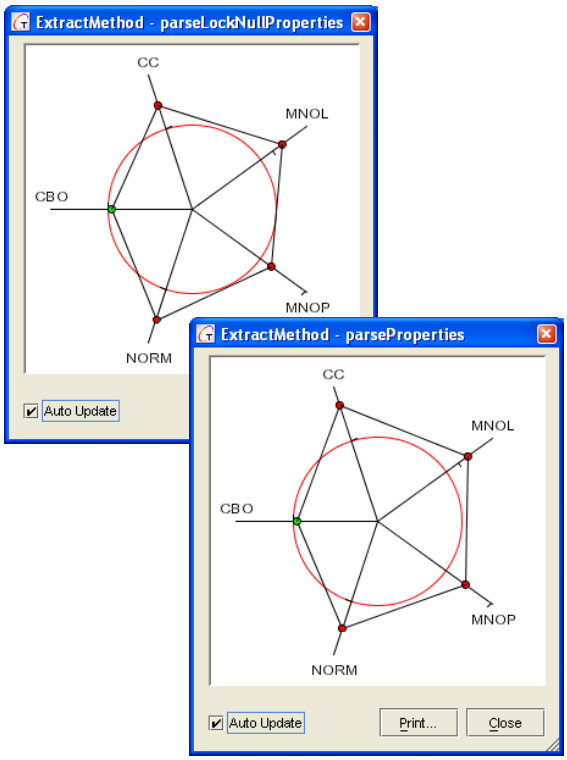

Figure 17*. Method comparison*

What do you suppose could best explain these results? A comparison of the bodies for each method should reveal that one was likely cloned from the other. This technique is a common form of "reuse" in software; however, it is not recommended.

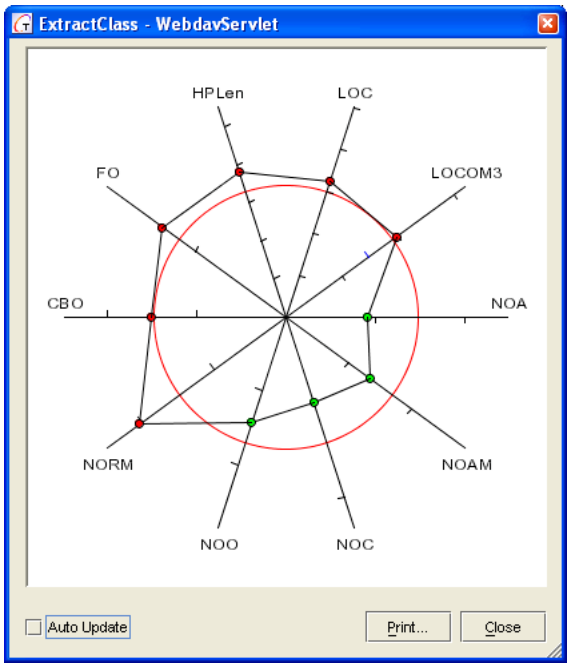

Figure 18. *Extract Class (WebdavServlet)*

Looking at the LOCOM3 value for the WebdavServlet, it is clear this class is lacking in cohesion by its score of 92. This leads to a number of questions regarding cohesion: Is it possible for all classes to be cohesive? Can WebdavServlet be more cohesive? What refactorings come to mind with a high value for LOCOM3? One possibility is Extract Class.

Running the custom metric set that attempts to identify a need for the Extract Class refactoring on WebdavServlet yields the following Kiviat graph (Figure 18).

In this set, the upper limit for LOCOM3 has been set to 80 because Together ControlCenter does not recommend a limit by default. Looking at the Kiviat graph for the results indicates that WebdavServlet could stand for some additional refactoring. Interestingly, it seems to be within the limits for Number Of Operations (NOO), Number Of Classes (NOC), Number Of Added Methods (NOAM), and NOA. So, it's not as though the class has an extraordinary number of methods, inner classes, added methods (those added to its inherited methods), or attributes. However, it does appear to have low cohesion, high coupling and a high fan out, along with excessive LOC and HPLen.

Before extracting a class, take a look at what this Servlet does. A sequence diagram generated from its service() method should indicate what its primary purpose is (Figure 19).

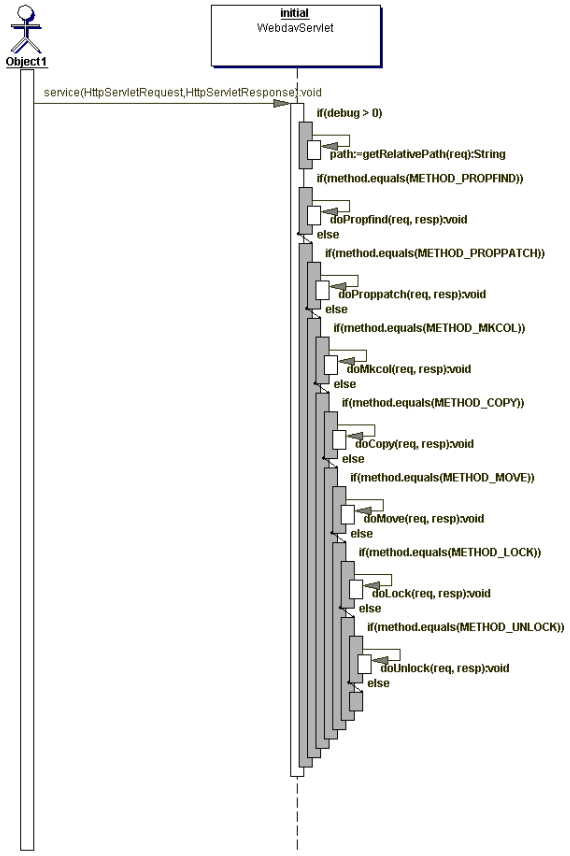

***Figure 19**. service() Method*

In this case, the service() method simply extracts the posting method for the incoming request and invokes a do* method for each of the available WebDav methods, passing the request and response objects along. What refactoring can best replace this ugly section of code? Perhaps Replace Conditional with Polymorphism? Suppose you created a new WebdavHandler

interface and have implementation classes for each of the WebDav methods. Then, you'd need a Factory pattern to return the appropriate handler given the passed in method that was extracted from the request object. In Together ControlCenter, you can use the built-in GoF Factory Method pattern to do just that. In the end, the WebdavServlet merely delegates to the appropriate handler class for each incoming WebDav request. This should significantly reduce the metric results for WebdavServlet, because its functionality is spread amongst the handler classes. Of course, this drives up some other metric values for Tomcat, given the newly created classes and interface. However, they should in themselves, and in summation, provide lower values for the offending metrics that started this Extract Class refactoring. It is left to you to complete this refactoring to see the exact results.

Conclusion

A great time to perform refactorings is during code reviews. Using Together ControlCenter with an overhead projector is recommended, because it is difficult to refactor, compile, and test code that has been printed out on paper.

Code reviews should occur frequently, as opposed to being left until the end of milestones or the entire project. With each developer using Together ControlCenter, the need to review mountains of source code for common audit hits is greatly reduced. In fact, given that code reviews are commonly skipped when project schedules tighten, automated audit and metric analyses are the only reviews likely to be done at all. When used routinely by all developers, the need for formal code review sessions may be eliminated altogether in the future, with the exception of business logic reviews. Until then, the use of audits, metrics, and automated refactorings serve as an accelerator to team reviews and ultimately raise the quality of not only the source code but also the developers who master them.

This paper merely introduces the notion of metric-driven refactoring and provides a crude example of its application. The refinement and standardization of metric result visualizations is needed to provide an added dimension to the recognition of class characteristics. It is not hard to imagine how a Kiviat graph, to the trained eye, can augment a UML class diagram in

displaying a more compete picture of class characteristics and interactions. Perhaps a UML profile can provide this visualization?

References

[Beck00] K. Beck, extreme Programming Explained: Embrace Change. Addison-Wesley, 2000.

[Beck01] K. Beck, et al, "Manifesto for Agile Software Development," Agile Alliance, 2001. http://www.agilemanifesto.org

[Bloc01] J. Bloch, Effective Java™: Program Language Guide. Addison-Wesley, 2001.

[Carm02] A. Carmichael and D. Haywood, Better Software Faster. Upper Saddle River, NJ: Prentice Hall PTR, 2002.

[Chid94] S. Chidamber and C. Kemerer, A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, Vol. 20 Number 6, pp476-493 1994.

[Coad99] P. Coad et al, Java Design: Building Better Apps & Applets, Second Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1999.

[Fowl99] M. Fowler, Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[Gack02] G. Gack, "The Value of Software Inspections: a case study", IT Effectiveness, Inc., 2002. http://www.iteffectiveness.com/valusoftinspect.htm

[Gilb93] T. Gilb and D. Graham, Software Inspection. Reading, MA: Addison-Wesley Publishing Co., 1993.

[Grad85] R. Grady, Software Metrics Etiquette. Cutter Information Corp., publishers of American Programmer, 6(2), 6-15.

[Hend95] B. Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity. Prentice Hall, 1995.

[IEEE90] ANSI/IEEE Std. 729-1983, IEEE Standard glossary of Software Engineering Terminology, 1990.

[IEEE98] ANSI/IEEE Std. 1028-1988, IEEE Standard for Software Reviews and Audits, 1998.

[Jone85] C. Jones, "A Process-Integrated Approach to Defect Prevention". IBM Systems Journal, 24(2):150-167 (1985).

[Keri02] J. Kerievsky, Refactoring to Patterns. Industrial Logic, Inc., version 0.14, 2002. http://industriallogic.com/papers/rtp014.pdf

[LiKo00] S. Li-Kokko, Code and Design Metrics for Object-Oriented Systems. Helsinki University of Technology, 2000.

[Mari98] R. Marinescu, Using Object-Oriented Metrics for Automatic Design Flaws Detection in Large Scale Systems. "Politehnica" University in Timisoara, ECOOP Workshop 1998: 252-255.

[McCa76] T. McCabe, "A Complexity Measure." IEEE Transactions on Software Engineering, SE-2 No. 4 (December): 308-20, 1976.