To Appear in Proc. Workshop on Empirical Studies of
Software Maintenance (WESS 2002), 2002.

Coding Concerns: Do They Matter?

W. Munger, J. Bieman, and R. Alexander
Computer Science Department
Colorado State University
Fort Collins, Colorado 80523
{wmunger,bieman,rta}@cs.colostate.edu

In the heyday of C programming it was the requirement of many programming shops to check code for anomalies and potential bugs with a program called lint [4,6]. As the quality of compilers improved, many of the lint functions were added to compilers as 'warnings'. But many programmers soon became tired of all the compiler warnings and turned most of them off. Today compilers come with just a few warnings turned on. A programmer wishes may set the flags to have the compiler provide additional warnings. As a result, many programs are compiled with the flags set to only a few of the warnings that the original lint programs provided.

This leads to the question: does the common programmer practice of turning off most compiler warnings indicate that these warning are no longer useful?

The programming environment has changed greatly since the lint program was first introduced. Modern OO languages such as Java, through the use of inheritance, polymorphism, overriding, and overloading [1,5] have increased the number of places where possible errors can hide, thus increasing the complexity of the needed lint-like program. As these are places where problems can hide and are not actually errors, we prefer to call them *concerns*. We define a *concern* as an anomaly in the code that should be checked to confirm that it is not an error.

A few OO lint-like programs exist, but the tendency today is to add lint-like features to software design tools. For our research we chose two such commercial tools: Together and JStyle. Both provide a way to load code into a project and then generate both software metrics and coding concerns. In Together the coding concerns function is called *audit*, which generates a file called *auditx.txt*. In JStyle the coding concerns function generates *comment reports* in html, txt, and a database format.

Together can identify a total of 96 concerns in 9 groups while JStyle can generate 80 different ungrouped concerns. These two packages find concerns that partially overlap. Below are just a few:

From Together

- Avoid constants with equal values
- Avoid empty catch blocks
- Avoid hiding inherited attributes
- Hiding of names
- Multiple Visible Declarations with same name
- No assignment in conditional expressions
- Unused local variables
- Unused private class members
- Unused formal Parameters

From JStyle

- Having an empty catch block
- Overriding base class method with empty method
- Private field of class is not used
- Private method of class is not used
- Local variable is not used
- Field in the class, hides one of the super class fields
- Local variable hides a field with the same name
- Local variable assigned a value but not used
- Local private field assigned a value but not used
- Variable assigned a value successively, without being used

JStyle can also finds some concerns that may help a programmer make modifications to speed up code execution. Clearly, some concerns are more important than others.

To get a better understanding of the importance of the use of lint-like programs, we used Together and Jstyle to examine three OO systems. The first system (System A) is commercial system written by programmers not well experienced in object oriented software. The final version has 23,221 lines of code with 396 classes in 17 packages. The second system (System B) is another commercial package; it was written by experienced OO programmers. The final version contained 17,082 lines of code with 213 classes in 17 packages. We examined 18 pre-release versions for both commercial systems.

With the large amount of Open Source code now available [2,3], we wished to compare commercial software with Open Source software. Many Open Source advocates suggest that Open Source code should be of high quality, because the source code is open to all and is examined by many programmers. So one might ask: if a programmer knows that the world is going to look at his or her code, will program quality improve? If the answer is "yes", we should find fewer coding concerns in the Open Source code than in the commercial systems. We selected a large program, Sun's Netbeans, out of the many available software packages. While Netbeans is Open Source it is very tightly controlled by Sun. Sun does not allow Netbeans to be "rsinked". So rather than attempt to down load each version one version at time, we requested and received a tarball of the Control Version System (CVS) from Sun.

Due to time constraints and an attempt to control the amount of data generated we examine version 1, 6, 12 ,18 of the commercial packages and 1.0x and 3.31 of the Netbeans.

Table 1 Code Metric Compared to Concerns

	Packages	Classes	LOC	Concerns	CPLOC	CPPack	CPC
Netbeans 1.0x	99	284	327028	312357	0.955	3155	118.1
Netbeans 3.31	534	11808	647729	629003	1.068	1295	58.60
System A 01	15	394	22882	22402	0.979	1493	56.86
System A 06	15	389	22306	21882	0.981	1461	56.25
System A 12	16	392	22480	21916	0.975	1370	55.91
System A 18	17	396	23221	22589	0.973	1329	57.04
System B 01	17	102	7476	6663	0.891	392	65.32
System B 06	17	188	15232	13973	0.917	822	74.32
System B 12	17	211	15924	14620	0.918	860	67.87
System B 18	17	213	17082	15428	0.903	908	72.43

One problem that we encountered in the generation of the concerns was that both Together and Jstyle cannot process systems greater than about 50,000 lines of code, the program and/or JVM memory requirements exceed the available memory. For this reason, Netbeans has to be broken up into several subsystems (version 3.31 has 692,003 lines of code). Also these programs are very time consuming when flagging all concerns on large systems. It took over five hours to process version 3.31 of Netbeans

We set both Together and Jstyle to find all possible concerns. We then took the code metrics of number of packages, number of classes, and lines of code and compared these with the total number of concerns found (see Table 1). The most surprising finding is the large number of concerns found, approximately one concern for every line of code. As mentioned earlier, there is overlap between the concerns generated by Together and Jstyle. Even if there was complete duplication, one concern for every two lines of code is still very high. No programmer is going to check this number of concerns to be sure that they do not contain errors!

For the Netbeans package a total of 145 different concerns were found. This is 82.4 percent of the 176 possible concerns. That is, 31 of the possible concerns were not found. For commercial package A 102 different concerns were found or 58.0 percent. And in commercial package B, 82 different concerns were found or 48.9 percent. The difference (24.4 percent) between the commercial and Open Source here is quite large. Since these are only descriptive statistics, we cannot determine if the difference is significant. However, the Open Source system has 43 more different types of concerns than found in commercial system A --- a difference of 42.2 percent. The difference between the two commercial programs is small; they have 20 different types of concerns. System B's lower number of concerns is consistent with the Head Architect's observation that the developers of System B were more experienced OO programmers.

We compared the concerns found in A with those found in B. Only one concern found in B was not found in A ---

ATLF Avoid Too Long Files --- which is not very important. However, 21 concerns found in A were not found in B. Some of these concerns may be critical; they include hiding of class variables and methods and failing to call super.finalize when creating a finalize method. A comparison of the concerns found in the commercial package A with those found in Netbeans shows that 43 of the concerns found in Netbeans were not found in A. One of these 43 was also found in B. This means that 42 concerns were found in Netbeans that were not found either of the commercial packages. These 42 concerns include the following:

- *Hiding inherited static methods.*
- *Don't Compare Floating Point Types.*
- *Empty finally block is useless.*
- *The 'return' statement in 'finally' block nullifies the effect of 'return' found within 'try' block.*
- *It is not a good idea to propagate any exception out of main.*
- *If a catch block merely rethrows the exception, the block is unnecessary.*
- *One of the base classes of this serializable class is non-serializable and does not provide an accessible no-arg constructor.*
- *Deserializing an object of this class will result in an exception.*
- *Use 'equals' Instead Of '=='*

Table 1 shows three measures that quantify the *concern density*: CPLOC is the number concerns divided by lines of code. CPPack is the number of concerns divided by the number of packages. CPC is the number of concerns divided by the number of classes. The CPPack measurements have the greatest amount of variation. System designers may cause the variation in CPPack measurements. They define, fairly early in the development process, the packages that will be included in the software. There is a similar variation in the CPC measurements; classes tend to also be defined early and completed as the software is developed. The most consistent measurement seems to be CPLOC. The data shows that it remains fairly constant during the development process. In the rest of this paper, we will use the CPLOC as a means of comparison of the concern density in programs.

Netbeans has the greatest variation in concern density of the three systems. Its final version has the highest concern density, 1.068. If considered by itself, the first version, at 0.955, would rank Netbeans second among the three systems. Because the first version of Netbeans is better than System A and the final version is worse than System A, the two systems are ranked as equivalent. However, when we compare System B is clearly better, in terms of concern density, than either System A or Netbeans. System B was developed by experienced OO programmers. The data suggests that such programmers produce code with fewer concerns.

To better understand the impact of concerns, we placed a subset of the concerns into four groups:

- Calling Concerns: *empty catch blocks, empty class, empty blocks, and empty method bodies.*
- Empty Concerns: *the method finalize() failing to call super.finalize(), and call of a non-final/non-private method in a constructor.*
- Hiding Concerns: *hiding of names, local variable hides a field, hiding inherited attributes, and multiple visible declarations with same name.*
- Not Used Concerns: *unused local variable and formal parameters, unused private method, variable assigned a value successively, without being used, local variable/private field assigned a value and not used*

We assigned relevant concerns into groups, eliminated duplicate concerns, and added the number of concerns of each category in a package. We developed bar charts and box plots for all four concern groups. For brevity we only examine the System A hiding concerns group. Figure 1 shows the bar chart of hiding concerns for System A, while Figure 2 shows the box plots. Note that package 7 has approximately 30 hiding concerns in each of the four versions examined. All the dots representing outliers in the box plots in Figure 2 are packages 4 and 7. Outliers show those packages that have the greatest number of concerns. The outliers represent an appropriate place to begin checking concerns to see if they indicate program errors. However, errors could be found at any of the concerns developed System B.

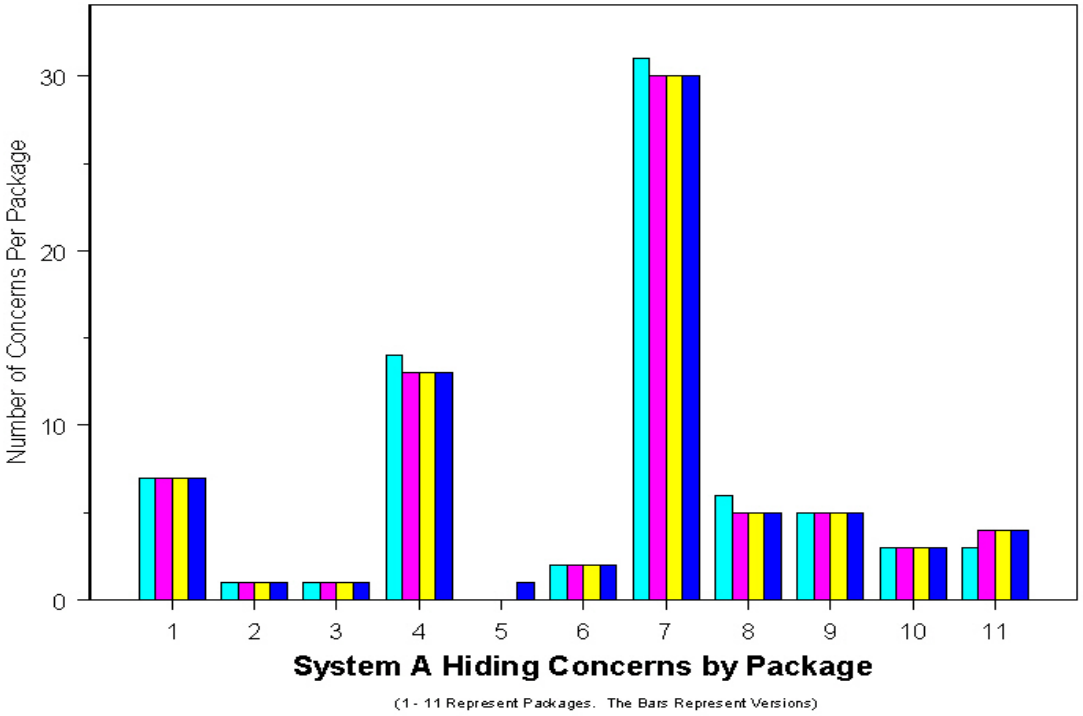

Figure 1 Bar Chart of Hiding Concerns Found in System A

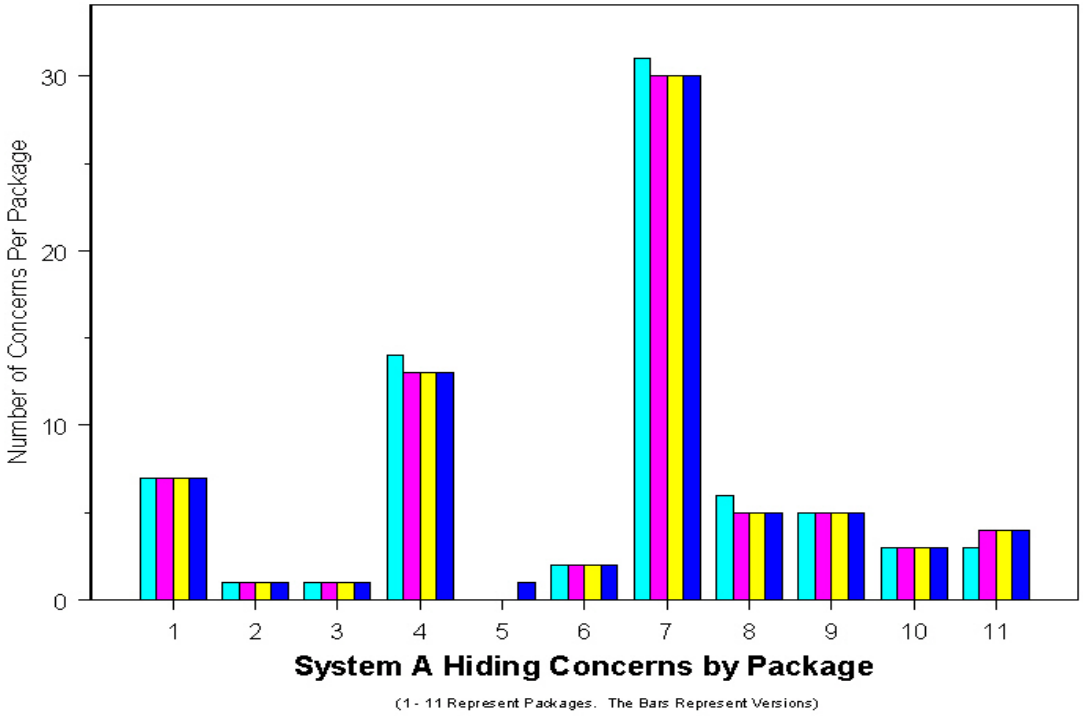

Figure 2 Box Plots of the Number of Hiding Concerns by Package in System A

The total number of hiding concerns for system A is 73 for version one and 72 for the last version. One package has nearly half of all the hiding concerns. In discussion with the primary software architect, we learned that this particular package had post release problems. NetBeans 3.31 had 4439 hiding concerns and 1.0x had 1993 or a CPLOC of 0.00641 and 0.00638 respectively. The final version of System A had a hiding concern CPLOC of 0.00319 and System B a hiding concern CPLOC of 0.000648. If we compare the Open Source and commercial systems purely on hiding concerns, this data would show that the commercial systems have a fifty percent advantage over the Open Source system. System B is an order of magnitude better than Netbeans, indicating that the experienced commercial OO programmers who developed System B wrote better code.

The Together concern ACEV, *Avoid Constants with Equal Values* appeared 16 times in System A in three different packages. Between version nine and ten, the programmer making changes to one of the packages realized that there were duplicate constants and remove one of them. Thus, the concern did not appear in version ten. If the team had checked the code for concerns in the first version, the duplicate constants could have been eliminate when it first appeared. The final version value of CPLOC for the ACEV concern for Netbeans was 0.00109, for System A 0,000620, and System B 0.0000. Again giving the commercial system B the edge.

Conclusion

The high number of concerns generated for all system raises several questions. When there are too many concerns generated, developers are likely to ignore them (like compiler warnings). Still, some of the concerns may represent errors that could latter cause problems. The data does not support the idea that having code available for all to see will reduce the concern density. If fact, it seems to indicate the opposite. However, the data does tend to support the concept that experienced OO programmers will write code containing fewer concerns. The Primary Architect noted that there were more problems with System A than System B. If greater concern density indicates more problems with the system, then the data does indicate that there should be more post release problems with System A.

Since the three systems examined in this paper are not necessarily representative of commercial and open source programs, it is impossible to generalize from this initial study. Empirical studies need to be undertaken that link concerns with errors and problems in a given system.

Before concerns can be a viable tool in the hand of the programmer the time required to find the concerns needs to be reduced and only those concerns that are known to be of value should be checked.

Future Research

Probably the greatest value of this paper is that it shows the need for more research about coding concerns and their relationship to errors. We found a very large number of concerns in the systems that we studied. Attempting to check out 650,000 concerns in 650,000 lines of code would be a very daunting task indeed! This is very discouraging to the programmer attempting to confirm that the concerns do not contain errors. A system needs to be developed that would allow the programmer to check a concern and as long as there are not changes to that part of the code the concern will not be displayed again. Thus the programmer will only have to check the new concerns or concerns where changes were made to the program.

Furthermore, a list of valid concerns needs to be developed and ranked. Such a list could be checked using an OO lint-like program. This would allow individual programmers to look for concerns that he or she feels are important. The list and ranking needs to be based on an empirical studies, and not just intuition or hearsay.

Procedures defining when and how to use concerns need to be developed. Code should be checked often enough to catch possible errors at an appropriate time, while at the same time not be time consuming nor overwhelming in number. The amount of time needed to run the software to find the concerns needs to be reduced and the software needs to be able to be run on large packages. Both Together and JStyle fail on these two points.

Acknowledgements

This work is partially supported by U.S. National Science Foundation grant CCR-0098202, and by a grant from the Colorado Advanced Software Institute (CASI). CASI is sponsored in part by the Colorado Commission on Higher Education, an agency of the State of Colorado. Storage Technology Corporation provided software, tools, and computer resources for this study. We also thank Frank Molnar for his generous equipment donation to this project.

References

[1] Alexander, Roger T. Testing the Polymorphic Relationships of Object-Oriented Programs. Department of Information and Software Engineering. Ph. D. Dissertation, George Mason University, Fairfax, Virginia, 2001.

[2] Bieman, J., R. Alexander, W. Munger, and E. Meunier. Software Design Quality: Style and Substance. *ICSE 2002 Workshop on Software Quality, 2002.*

[3] Bieman, J. and V. Murdock Finding Code on the World Wide Web: A Preliminary Investigation. *Proceedings First International Workshop on Source Code Analysis and Manipulation* (SCAM 2001), Nov. 2001.

[4] ButterField, Jim. C, Lint and Confusion. *ICPUG Journal* September/October 1992.

[5] Offut, J., R. Alexander, et al. A Fault Model for Subtype Inheritance and Polymorphism. *Twelth IEEE International Symposium on Software reliability Engineering(ISSRE '01)*, Hong Kong, PRC.

[6] Spencer, Henry. "Henry Spencer's 10 Commandments for C Programmers". Http://www.isthe.com/chongo/tech/comp/c/10com.html.