Confessions of Some Used-Program Clients *

Joseph E. Hollingsworth
Bruce W. Weide
Stuart H. Zweben

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210
Tel: 614-292-1517
Email: weide@cis.ohio-state.edu

Abstract

In the past, claims have been made that one can expect improved software quality and higher programmer productivity by faithful application of abstraction, encapsulation and layering (A/E/L). In an effort to explore the effects of A/E/L in the context of reusable software components, we conducted an empirical pilot study using a class of graduate and upper-division undergraduate students. We present some statistical results concerning the effects of A/E/L based on the data collected by the study.

Keywords: education, guideline development, metrics, reusable software components

1 The Problem

Many respected software engineers (e.g., [Parnas 72]) have long argued that potentially significant quality and productivity gains can be achieved by faithful use of abstraction, encapsulation, and layering (A/E/L). In this approach, higher-level parts of the system are layered on top of lower-level encapsulated abstractions. The claimed benefits stem largely from separation of concerns between a component's implementer and a component's client. The component implementer needs to understand only the abstract interface, not its use by the client; the client can reason abstractly about higher layers of software knowing only the abstract interface, not its implementation. If underlying representation or algorithmic details change (e.g., to improve performance) the higher layers

*This material based upon work supported by the National Science Foundation Grant No. CCR-9111892

remain stable. This modularity property is especially important when the lower-level abstractions are reusable software components [Ernst 91, Weide 91].

In our experience as "used-program clients" (apologies to [Tracz 88]), we have noticed that A/E/L principles are not faithfully observed by some used-program salesmen. There are two main problems:

1. Some component libraries do not use A/E/L principles as much as they could, within those libraries. For example, [Booch 87] represents a "map" abstraction as a hash table using chaining for collision resolution. But he codes from scratch the lists that implement chains. He does not reuse the list package.

2. The designs of components sometimes interact with each other and with certain language features to make it difficult for clients to respect abstraction while layering new functionality on top of existing components. The Booch components again provide an example. Ada's restriction on the mode for parameters to functions, mixed use of private and limited private types, and a variety of details of the component designs combine to make it surprisingly difficult for clients of these components (and all others we know of) to observe A/E/L principles [Hollingsworth 91]. It is, therefore, usually considered fortunate (although it probably should not be [Muralidharan 90b]) that many component libraries are in source form. This makes it possible for clients to extend and change component interfaces to suit their own needs—not by layering on top of encapsulated abstractions but by directly modifying them.

Why should such an apparently well-established principle of software engineering—especially one that seems to form the foundation for software reuse—still be so elusive in practice? We suggest there are two main reasons:

1. There are some obvious disadvantages to A/E/L that temper the claimed advantages. The most important is that performance suffers. *Secondary* operations implemented by layering involve extra procedure call overhead. This is usually a small constant-factor performance penalty that can be reduced with aggressive inlining and other compiler optimizations; yet it may be important in some applications. But secondary operations also may be slow because the *primary* operations provided by an underlying component do not offer the proper abstract functionality, with the right performance, to permit a layered implementation to execute as quickly as if it were permitted to access underlying representations. This can be an order-of-magnitude performance penalty that the client cannot overcome except by violating abstraction—prying open an encapsulated component and delving into the guts of its implementation.

2. While A/E/L may have many software engineering benefits *in principle*, apparently there are no controlled empirical studies that document measurable quality or productivity benefits of using these techniques. In making a trade-off between the analyzable and measurable performance penalties associated with faithful adherence to A/E/L, and the largely hypothetical and unquantified quality and productivity gains, a designer or manager is clearly tempted to opt for performance. This is particularly true where the programming language contains "features" such as code inheritance that seem to support layering, but that actually encourage violation of abstraction and encapsulation [LaLonde 89, Muralidharan 90a, Raj 90].

We are impressed by the importance of the second point in many informal discussions with software practitioners. Some claim to see the benefits of remaining completely faithful to A/E/L. They blame short-term thinking by management, inflexible deadlines, unrealistic performance objectives, and a variety of other factors, for violations of principles. The true skeptics, though, harbor sincere doubts about the claimed advantages of A/E/L in practice. They really would like to see some empirical evidence that a vigilant adherence to A/E/L actually "works."

Having contributed already to the hypothetical academic arguments for essentially complete allegiance to A/E/L principles in design of reusable software components [Harms 91, Weide 91], we considered how we might influence potential industrial collaborators to undertake a realistic empirical evaluation of the benefits of this approach. During summer 1991 we used a class of 18 graduate and upper- division undergraduate students to conduct an empirical pilot study of some productivity and quality effects of A/E/L in the context of reusable software components. The main purpose of this paper is to present preliminary results of that study, which support our position that observing A/E/L principles is an important factor in obtaining the claimed productivity and quality benefits of reuse.

2 The Study

The class in question was called "Software Components Using Ada." The lectures heavily emphasized the trade-offs evident with A/E/L principles, and presented a detailed engineering discipline for designing, formally specifying, and correctly and efficiently implementing Ada generic packages. We used [Booch 87] as a supplementary text and did one project using the Booch components. But the majority of the course used our own component designs and our own engineering discipline [Harms 91, Weide 91]. Several programming assignments illustrated main points from the lectures. On some of the assignments, we asked the students to keep track of the effort they spent on various activities (which we defined as carefully as possible), and on the number of bugs that caused run-time errors that they found and fixed.

Two of the assignments were particularly relevant to the point of this paper. In the first, we provided an implementation of a generic "unbounded queue" package that exported the standard primary operations: enqueue, dequeue, and a test for emptiness. We asked the students to add four secondary operations: copy, clear, append (concatenate two queues), and reverse. We formally specified all the operations and discussed them in class so there would be no doubt as to their intended semantics. We asked the students to implement these secondary operations using two different methods: (a) by layering them in a new generic package on top of the provided queue abstraction, and (b) without layering, i.e., by directly modifying the underlying generic package to export the four additional operations. Half the class (nine students chosen at random) did part (a) first; the other half did part (b) first. Below we call these Group A and Group B, respectively.

For the next assignment, we provided a standard solution to part (b) of the above assignment, and asked the students to change the underlying representation of queues. This required that they redesign and recode the implementations of the original primary operations as well as the four secondary operations. We asked that all the students first reimplement the primary operations, then the secondary operations.

For both assignments we asked the students to keep careful records of the time they spent in

designing/coding, testing, and debugging/recoding each operation. We also asked them to report how many bugs they fixed in each operation. We emphasized the importance of being internally consistent in keeping and reporting this data, and stressed that grades in the course would have nothing to do with the reported numbers. After discussing the study with each of the students before and after the assignments, we found no reason to believe that the results were significantly affected by variations in reporting methods, by collaboration, by severe outliers, or by latent fears that honest effort/bug data would influence course grades.

3 The Results

We have just started to analyze the data and cannot yet report everything that might be lurking in them. We plan to document statistical details of the following (and other results) in a future paper.

Assignment 1

Examining average total effort data for the two parts of the assignment (Table 1), we noted that the students overall spent less than half as much total time on the layered implementation as on the one without layering. Even those who did part (a) first spent less total time on the layered implementation than on the non-layered one. Looking at just design/coding effort gave a similar picture.

Table 1:
Average Total Times for Assignment 1

	Group A (Layered First)	Group B (Direct First)	All Students
Layered	145	57	101
Direct	182	261	222
Total	327	318	323

To test the statistical significance of these observations, we performed an analysis of variance [Hicks 73], looking for the significance of three primary effects on the total effort required for the assignment: (1) the effect due to the treatment, i.e., the difference in times to implement the secondary operations with layering and without layering; (2) the effect due to the group, i.e., the effect, on total time to do the two implementations, of the order in which layering and non-layering were done; and (3) the interaction effect between treatment and order, i.e., the potential "learning" effect that completing the first implementation had on the time to do the other implementation. Our nested-factorial model also included the effect due to students within groups and the interaction effect between students and treatments, but these effects were untestable because we had only one point per student for each level of treatment. In this model, effects (1) and (3) are tested against the interaction between students and treatments, while effect (2) is tested against the student effect. We looked for F values that were significant at the 5% level; with 1 and 16 degrees of freedom, the minimum significant F is 4.49.

We found (Table 2) that effects (1) and (3) were statistically significant, and that effect (2) was not significant. That is, non-layering took significantly more total time than layering. Furthermore,

there was an apparent learning effect in the sense that the total time spent on the first treatment condition was significantly greater than that for the second treatment condition. We found no significant difference between the two groups in the total time to do both parts of the assignment.

Table 2:
Analysis of Variance for Total Time for Assignment 1

Source/Effect	*df*	*Sum of Squares*	*Mean Square*	*F*
Treatment (layering)	1	130,321	130,321	23.70*
Group (order)	1	160	160	0.02
Treatment X Group (learning)	1	63,001	63,001	11.46*
Student (within Group)	16	149,566	9,348	
Treatment X Student	16	87,970	5,498	

$*$ Significant at the 5% level, i.e., $F > F_{1,16} = 4.49$.

These data indicate a measurable productivity advantage when secondary operations are implemented without violating A/E/L principles. Several students noted in their lab reports that it was far easier to think abstractly about queues when designing and coding the secondary operations than it was to worry about the nodes and pointers of the underlying representation. This seems to be the most reasonable explanation of the observed data—exactly what A/E/L advocates might have predicted.

The lack of a significant effect due to order is also plausible from common sense. While there is reason to expect that something about the task will be learned from the first treatment condition, in fact the mode of thinking, algorithms, and code for layering and non-layering are quite different. Therefore, the total time to complete both parts of the assignment should (intuitively) be independent of which one was done first. Indeed, this is what we observed.

We also found a significant difference in the quality of the code, as measured by the number of bugs causing run-time errors that were found and fixed before testing revealed no more. The layered implementations had significantly fewer bugs than the non-layered ones. Based on the Mann-Whitney U Test [Downie 65], we were able to reject the hypothesis of no difference between the number of bugs in the layered and non-layered implementations, at the 5% level.

Assignment 2

In the second assignment the students undertook a typical maintenance task: change the representation of an abstraction and all the code that depends on it. Using layering, as in part (a) of the first assignment, means that the code for the secondary operations can be written once and certified to be correct. A change to the underlying representation costs only as much as changing the primary operations. The students, however, also had to change the secondary operations, because they were implemented without layering. It was this extra—and with A/E/L principles, unnecessary—effort that the assignment was intended to help us measure.

We found that the students averaged spending about half their total redesign and recoding effort on the four secondary operations. However, they had to find and fix an average of two-thirds of all their bugs in these operations. These data have such large confidence intervals that

we hesitate to draw any serious conclusions from a small sample and one example. Nonetheless, it is entirely plausible that secondary operations generally should be more difficult to get right than primary operations. Secondary operations perform more complicated manipulations than the primary operations, which are chosen precisely because they are "primitive."

4 Status and Recommendations

We plan to examine more carefully the effort and bug data obtained in this small study. We also hope to refine it and run a study again next year with different students. But the preliminary results suggest that a commercial software developer might do well to adhere carefully to A/E/L principles on a realistically large software project, collecting as much similar and related data as possible, in an attempt to document a convincing empirical case for the productivity and quality advantages of A/E/L. By knowing the cost of design and coding time, maintenance activities, etc., and having estimates of the different amounts of time involved in these tasks, a manager should be able to make a more informed trade-off between software engineering costs and run-time performance costs of design decisions.

References

[Booch 87] Booch, G., *Software Components with Ada*, Benjamin/Cummings, Menlo Park, CA, 1987.

[Downie 65] Downie, N.M., and Heath, R.W., *Basic Statistical Methods, 2nd ed.*, Harper & Row, New York, 1965.

[Ernst 91] Ernst, G.W., Hookway, R.J., Menegay, J.A., and Ogden, W.F.,"Modular Verification of Ada Generics," *Comp. Lang. 16*, 3/4 (1991), 259-280.

[Harms 91] Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Trans. on Software Eng. 17*, 5 (May 1991), 424-435.

[Hicks 73] Hicks, C.R., *Fundamental Concepts in the Design of Experiments*, Holt, Rinehart and Winston, New York, 1973.

[Hollingsworth 91] Hollingsworth, J.E., Weide, B.W., and Zweben, S.H., "Abstraction Leaks in Ada," *Proc. 14th Minnowbrook Workshop on Software Eng.*, Blue Mountain Lake, NY, July 1991.

[LaLonde 89] LaLonde, W.R., "Designing Families of Data Types Using Exemplars," *ACM Trans. on Prog. Lang. and Syst. 11*, 2 (1989), 212-248.

[Muralidharan 90a] Muralidharan, S., and Weide, B.W., "Should Data Abstraction Be Violated to Enhance Software Reuse?," *Proc. 8th Ann. Natl. Conf. on Ada Tech.*, ANCOST, Inc., Atlanta, GA, Mar. 1990, 515-524.

[Muralidharan 90b] Muralidharan, S., and Weide, B.W. "Reusable Software Components = Formal Specifications + Object Code: Some Implications," *3rd Annual Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center, Syracuse, NY, July 1990.

[Parnas 72] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *CACM 15*, 12 (Dec. 1972), 1053-1058.

[Raj 90] Raj, R.K., "Code Inheritance Considered Harmful," *3rd Annual Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center, Syracuse, NY, July 1990.

[Tracz 88] Tracz, W.J., ed., *Tutorial: Software Reuse: Emerging Technology*, IEEE Computer Society Press, Washington, DC, 1988, 92-95.

[Weide 91] Weide, B.W., Ogden, W.F., and Zweben, S.H., "Reusable Software Components," in *Advances in Computers*, v. 33, M.C.Yovits, ed., Academic Press, 1991, 1-65.

Biographical Data

Joe Hollingsworth holds an undergraduate degree from Indiana University and a master's degree from Purdue University. Before returning to school as a Ph.D. candidate at The Ohio State University, he worked at Texas Instruments. He has also consulted for Battelle Memorial Institute on issues of software design in Ada. In his recent research at OSU he has developed a compiler, linker, and run-time system for RESOLVE, and has worked on a set of engineering principles that can be used to develop generic reusable software components in Ada.

Bruce W. Weide is Associate Professor of Computer and Information Science at The Ohio State University. He received his B.S.E.E. degree from the University of Toledo in 1974 and the Ph.D. in Computer Science from Carnegie Mellon University in 1978. He has been at Ohio State since 1978. His research interests include various aspects of reusable software components and software engineering in general: software design-for-reuse, formal specification and verification, data structures and algorithms, and programming language issues. He has also published recently in the area of software support for real-time and embedded systems.

Stuart H. Zweben, a 1974 Ph.D. graduate of Purdue University, is Associate Professor of Computer and Information Science at The Ohio State University. He is a leader in the field of software metrics and has published extensively in the areas of software testing, software engineering methodology, software understandability, and design of efficient data structures. His current research interests include software reuse and software testing issues, as well as software engineering tools and methods, and evaluation techniques for measuring the effects of such tools and methods.