The Effects of Layering and Encapsulation on Software Development Cost and Quality

Stuart H. Zweben, Stephen H. Edwards, Bruce W. Weide, and Joseph E. Hollingsworth

Abstract— **Software engineers often espouse the importance of using abstraction and encapsulation in developing software components. They advocate the "layering" of new components on top of existing components, using only information about the functionality and interfaces provided by the existing components. This layering approach is in contrast to a "direct implementation" of new components, utilizing unencapsulated access to the representation data structures and code present in the existing components. By increasing the reuse of existing components, the layering approach intuitively should result in reduced development costs, and in increased quality for the new components. However, there is no empirical evidence that indicates whether the layering approach improves developer productivity or component quality.**

We discuss three controlled experiments designed to gather such empirical evidence. The results support the contention that layering significantly reduces the effort required to build new components. Furthermore, the quality of the components, in terms of the number of defects introduced during their development, is at least as good using the layered approach.

Experiments such as these illustrate a number of interesting and important issues in statistical analysis. We discuss these issues because, in our experience, they are not well-known to software engineers.

Index Terms—**Empirical study, encapsulation, software components, abstract data types, software development, software reuse.**

I. Introduction

IT IS well-known that software systems typically exceed their expected development and maintenance costs. While there are many perceived reasons for this, one reason is that the components of these systems tend to be developed nearly from scratch, instead of being predominantly reuses of existing components [15].

The ability to successfully reuse components in new systems depends on the components being properly encapsulated. That way, new components can be "layered" on top of them, taking advantage of the abstract notions already encapsulated and avoiding reimplementation of these abstractions. For many years, programming languages have provided various means of encapsulating data structures and their associated operations, and the means of layering new components using these encapsulated components. Supposedly, there are productivity and quality gains to be had by following this layering technique [20].

However, currently we do not see in software systems the widespread use of encapsulation to layer the system's components. This is true even in popular component libraries. For example, Booch [2] represents a map abstraction as a hash table using chaining for collision resolution. But he codes from scratch the lists that represent chains. He does not reuse his list package. Studies of object-oriented class hierarchies often have found that these "hierarchies" in fact are very flat [4], indicating that most components are not really built on top of existing components.

One possible reason for this lack of layering is that the original components are not well-designed, so that 1) the components needed in the next application are not quite those that are currently available, and 2) the available components cannot easily be converted (by layering) into those that *are* needed. Features and standard use of programming languages, such as Ada's restriction on the mode for parameters to functions and mixed use of private and limited private types, also inhibit one's ability to compose components [11].

Another reason may be that it is not obvious that, even if there were a useful component available, it would be better to reuse it by respecting its encapsulation than it would be to develop the required new functionality by accessing the underlying representation of the component. Performance is most often mentioned as the basis for this belief, although in our experience the performance penalty typically is minimal if the proper abstract functionality is encapsulated in the component. Language features such as code inheritance actually encourage violation of encapsulation while appearing to support layering [12].

We know of no controlled studies showing that the use of layering and encapsulation improves the cost of component development or the quality of the components, based on such measures as time to design/develop or number of defects in the resulting product. While abstract and anecdotal arguments have some place in technology assessment, it is vital that sound empirical support be obtained for the use of emerging software technologies.

Manuscript received April 1994; revised September 1994 and November 1994. Recommended by J. Gannon. This work was supported in part by the National Science Foundation under Grants CCR-9111892 and CCR-9311702, and by ARPA under Contract F30602-93-C-0243, monitored by the U.S. Air Force Material Command, Rome Laboratories, ARPA Order A714.

S. H. Zweben, S. H. Edwards, and B. W. Weide are with Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210 USA (e-mail: zweben@cis.ohio-state.edu); (e-mail: edwards@cis.ohio-state.edu); (e-mail: weide@cis.ohio-state.edu).

J. E. Hollingsworth is with the Department of Computer Science, Indiana University Southeast, New Albany, IN 47150 USA (e-mail: jholly@ius.indiana.edu).

IEEE Log Number 9409037.

Lewis *et al.* [13], did study differences in productivity and quality when subjects were allowed to reuse existing components versus when they were not. They found that subjects who were allowed to reuse performed significantly better than those who were not. However, for their experiments, "reuse" included alteration of existing components' code. Therefore, these experiments really could not show the effect of layering. The Lewis experiments also studied the effects due to the language in which the components were written, using an object-oriented language (C++) or a procedural language (Pascal). While there undoubtedly were some potential differences in encapsulation between these alternatives, the encapsulation did not need to be respected by the subjects in reusing components, since the subjects were allowed to see inside the existing encapsulated units and modify them as they saw fit.

In this paper, we report on three experiments designed to assess the effect, on effort and quality, of designing and developing software components by taking full advantage of existing abstractions, compared with an approach that allows seeing and using information about the implementation of the existing abstractions when designing and developing new components. Each of the experiments is a controlled study [5], and in each study there are interesting aspects to the statistical analysis required to determine if there is a significant difference between the two approaches.

The next section gives some additional background to motivate the experiments, while Section III describes each experiment in detail. Section IV discusses important issues that affect the statistical analyses appropriate for these experiments. Section V discusses the results.

II. Background

Controlled studies are relatively rare in software engineering, and this often is attributed to the prohibitive cost of repeating large systems development tasks so that the effect of using a particular method can be examined relative to a baseline. Most attempts at doing a controlled analysis have been done with small sets of programmers (frequently students) on very small projects. While the applicability of the results of such studies to "real" systems is not obvious, studies of this nature have proven useful in the past. In some cases, they have shown that a particular method may not have the desired effect, or they have helped us to understand better the various factors that may influence the effect of a method [19]. If the tasks performed by subjects in these experiments indeed are a subset of the tasks performed in the development of larger systems, and if the results of these experiments can be replicated, then the experiments can offer useful information about pieces of the complex process of software development.

What kinds of tasks might be typical of large systems development, and therefore worth investigating via controlled experimental studies? Two possibilities are 1) adding functionality to an existing component, and 2) building a component that has *similar* functionality to that of an existing component. In the former case, all that is desired is a new set of operations, but the existing component is of the right kind to provide these operations; the existing component's set of capabilities just needs to be enhanced. In the latter case, a closely related component might be available, but the desired component must provide somewhat different, rather than merely additional, functionality.

TABLE I
Comparison of Component Development Approaches

Component Devel. Approach	Type of Reuse	Information Used	Understanding Required
Direct Implementation	White box (open, transparent)	Coding details, representation data structures	Purpose/functionality and implementation of existing components
Layered	Black box (closed, opaque)	Functional specification, interface description	Purpose/functionality of existing components

In both types of tasks, one can imagine developing the new components by reusing existing ones in at least two ways. One reuse approach is to "directly implement" the desired functionality using the coding details of existing components. This involves making use of information about the data structures used to represent the objects encapsulated by the existing components. Some people refer to this type of reuse as "white box," "open," or "transparent" reuse. A second reuse approach to developing the desired functionality is to use only the specifications of the functionality of, and the interfaces to, the existing components. We call this the "layering" approach because the new functionality is provided by components built on top of the existing encapsulated units. Other names for this approach might be "black box," "closed," or "opaque" reuse (see Table I).

Both approaches require that the implementor understand the purpose of and functionality provided by the existing components. But the first approach also requires an understanding of the *implementation* of the existing components, while the second approach does not. Intuitively, this suggests that black-box reuse demands of the implementor a lower cognitive load, and hence should reduce the *effort* required, at least for the initial design and coding of the new components.

The expected effect on *quality* of layering versus direct implementation is somewhat less clear. In this paper, we use the term "quality" in the sense of "correctness," though we know that quality has other dimensions, too. Since the need to work with, and possibly misuse, the existing implementation is eliminated with black box strategies, one might expect better quality (i.e., fewer defects) from components developed using layering. On the other hand, since with layering one has available only the operations provided by the existing components and not the underlying representation data structures, the algorithms used in the layering approach might differ from those used in the direct implementation approach. This use of different algorithmic approaches might give rise to different distributions of defects. Even if the same algorithmic approaches are used in the two methods, most of the defects might occur in the attempt to synthesize the algorithm from the existing operations, rather than in the manipulation of representations of existing data structures.

Both of the tasks described above, that of adding functionality and that of modifying functionality, and both reuse

approaches, were studied in our experiments. The experiments are described in detail in the following section.

III. Overview of the Experiments

Our experiments were conducted as part of a course on the subject of "Software Components in Ada." This course was offered in two separate quarters, once in the summer of 1991 and again in the fall of 1992. In each case, the students in the course were graduate and upper-division undergraduate students majoring in computer science. Several of the students were full-time employees in the computing field.

The lectures heavily emphasized the trade-offs evident with principles of encapsulation, abstraction and layering, and presented a detailed engineering discipline for designing, formally specifying, and correctly and efficiently implementing Ada generic packages exporting abstract data types. Several programming assignments illustrated main points from the lectures, and the experiments were conducted in conjunction with some of these assignments.

Our main independent variable for the experiments was the reuse approach used by the subjects, and the dependent variables of interest were effort and component quality. Therefore, the primary null hypothesis to be tested is that the reuse approach used has no significant effect on the development effort or the quality of the resulting component.

To assess effort, we had the subjects keep careful records of the time spent in the initial designing/coding, testing, and debugging/recoding phases of the task. These times were recorded individually for each operation exported by the component under investigation. Both the initial designing/coding time and the total time to complete the task (including the testing and debugging/recoding time) were of interest to this study.

To assess quality, for each operation of the component the subjects recorded the number of defects they needed to fix. Only those defects that caused run-time failures during testing were included.

We emphasized to the students the importance of being internally consistent in keeping and reporting the data, and stressed that grades in the course would have nothing to do with the reported numbers. We audited the information provided by the students through post-assignment interviews, and the programs submitted by the students were tested by the instructor to ensure (as well as possible) that no lingering defects remained. We were somewhat skeptical of the data on a per-operation basis, but were confident that the aggregate information provided by the students was accurate.

Several factors other than reuse approach might influence the results on the dependent variables. One of these, the nature of the activity (i.e., enhancement of functionality versus modification of functionality) was mentioned in the previous section. Other important factors include the component involved in the reuse activity, the subjects' familiarity with the component specification and representation, and the abilities of the subjects. In planning the experiments, we attempted to deal with each of these issues.

The first experiment was an "enhancement of functionality" exercise using a simple, well-understood unbounded queue component. In addition to the "universal" package operations of `Initialize`, `Finalize`, and `Swap` [21], [8], [11], the basic Ada queue package provided the standard `Enqueue`, `Dequeue`, and `Is_Empty` operations (specifications for an Ada queue package can be found in [11]). The enhancement task was to add operations to `Copy` a queue, `Clear` (i.e., empty) a queue, `Append` one queue to another, and `Reverse` a queue. The representation structure used for the queue was a standard linked data structure with pointers to the front and rear. Eighteen subjects participated in this experiment. Since the functionality and representation structure used in this package were so familiar to the subjects—both from classical data structures and a previous lab exercise—we felt that it would be difficult to obtain significant differences due to the approach used. Preliminary results from this experiment were reported in [10].

The second experiment was a replication of the first experiment using a more complex component that encapsulated a "partial map." This component allows a client to create an associative mapping which is a partial function from an arbitrary domain type to an arbitrary range type; it is useful in table processing applications. In addition to `Initialize`, `Finalize`, and `Swap`, the basic package provided the following operations (specifications for an Ada partial map package can be found in [18]).

- `Make_Defined` Assign a given range value to a given domain element (add an association to the map).
- `Make_Undefined` Undefine a given, presently defined domain element (i.e., remove a particular association from the map).
- `Make_Any_One_Undefined` Undefine some presently defined domain element, chosen arbitrarily and returned by the implementation (remove an arbitrary association from the map).
- `Test_If_Defined` Test if a given domain element is defined.
- `Test_If_Any_One_Defined` Test if there are *any* defined elements in the map.

The enhancement task for this experiment was to add operations that would `Display` (print) the map, `Clear` it (making every domain element undefined, according to the map), `Combine` two partial maps (assuming there were no inconsistently defined mappings in these two maps), and `Remap` all domain elements that map to one range value so they instead map to another range value. The representation chosen for the partial map was a hash table. The subjects were taught about this package in a previous lab assignment, where they were asked to implement the package using the hash table representation. Nevertheless, in comparison with the unbounded queue package, both the specification of the partial map package (i.e., the description of the functionality of the operations) and the representation of the data type were more complicated. Furthermore, we felt that the experience with the representation structure gained in the earlier lab could only *improve* the subjects' abilities to do the direct implementation

version of the enhancement, relative to their ability to do the layered version. Again, we expected that it would be difficult to obtain significant differences between the reuse approaches. Ten subjects participated in this experiment.

The third experiment was a "modification of functionality" experiment using the partial map component. Given the basic partial map package used in the second experiment, the task was to create a component that encapsulated the concept of an *almost constant map*, which maps all but a finite number of domain elements to a "default" range value. Other than `Initialize`, `Finalize`, and `Swap`, the operations required by this package are as follows.

- `Reset` Reset the map to a constant function in which all domain elements map to a specified default value.
- `Get_Default` Return the default value, to which most of the elements are mapped.
- `Swap_Range_Value` Modify the range value associated with a given domain element.
- `Remove_Any_Anomaly` Make an arbitrary domain element, presently not mapping to the default value, instead map to the default value.
- `Test_If_Anomaly` Test if a given domain value maps to the default value.
- `Test_If_Constant` Test if all domain values map to the default value.

In this experiment, the subjects would be creating a rather unfamiliar component using what, by now, would be a somewhat familiar component. The same ten subjects who participated in the second experiment also participated in this third experiment.

To mitigate the differential effects of subjects on task completion (see, e.g., [3]), in each experiment each subject performed the task using *both* the layering and direct implementation approaches. The order in which the two approaches were followed was assigned randomly, subject to a counterbalancing with respect to the experience level of the subjects (i.e., we didn't want more experienced subjects doing the tasks in one order while the less experienced subjects did the tasks in the other order). Due to the uneven distribution of subject experiences and the relatively small number of subjects available for study, we chose not to investigate experience level as a separate factor in the experimental design.

To summarize, each experiment was designed to assess the effect of the reuse approach (layered or direct implementation) on each of three dependent variables (initial design and coding time, total time, and number of defects). Each subject in each experiment was assigned to one of two sequences in which the two reuse approaches were employed (layered first or direct implementation first). Thus, as illustrated in Table II, each of the experiments was a two-period crossover design [14], [16].

TABLE II
FACTORS IN THE EXPERIMENTAL DESIGN

Treatment	Sequence	
	Layered First	Direct First
Layered	9 subj. in Exp. 1	9 subj. in Exp. 1
Direct	5 in Exp. 2 and 3	5 in Exp. 2 and 3

IV. STATISTICAL ANALYSIS ISSUES

The analysis of variance (ANOVA) model for these experiments allows us to decide if there is a significantly different effect, on each dependent variable, of the layering approach versus the direct implementation approach (in statistical terminology, this is the "treatment effect" in the experiment). The null hypothesis is that there is no such effect. In addition, the model allows us to decide if there is a significant difference in the two possible sequences (orders) in which these two approaches were employed. Finally, it allows us to assess if the treatment and the sequence interact; that is, we can test if there is a different effect of the approach depending on the order in which the two approaches were employed. Since each subject did the same task twice (once for each of the two approaches), but did so in only one of the two possible sequences, in statistical terminology the subjects are nested within sequences. Thus, this model is sometimes called a "nested factorial" design [9]. In the model, the variance associated with subjects (nested in sequences) is used to test for the sequence effect, while the variance of the subject by treatment interaction is used to test for the treatment effect and the treatment by sequence interaction. Some texts call this latter variance the "time error term" [14].

However, the normal nested factorial analysis of variance may not be appropriate for these experiments, for two reasons.

1) An important concern is the potential that a subject's having done a task once will affect performance on the second attempt at the task (even though a different approach is being used). This possibility of a "carryover effect," as it is known in statistics, requires that care be taken when deciding if there really is an effect due to the approach used.

A common and accepted way of dealing with this issue is first to test if there is a significant sequence effect and/or a significant interaction effect. If there is neither, then the normal nested factorial analysis of variance is used to test for the effect of the treatment (i.e., the approach). If either the sequence effect or the interaction effect is significant, however, the treatment effect is tested using only the data for the first period in the sequence. That is, for those subjects who did the layering approach first, only their data for the layering approach is used; for those who did the direct implementation first, only their data for the direct implementation approach is used. The error estimate used for this test is a function of the subject error and time error terms [14].

2) The defect data typically will comprise small integer values, whose distribution does not satisfy the normality assumptions required of a standard analysis of variance. Defect data may be modeled more accurately by a Poisson-like distribution, and there are common statistical procedures to do Poisson (regression) analysis, allowing tests for the significance of the sequence, treatment, and sequence by treatment interaction effects. However, a true Poisson distribution would imply that the mean and variance are equal. The well-known vast differences in subjects makes this assumption unlikely to be met in our experiments. When the true variance is higher than that assumed by the Poisson model, a phenomenon called

"overdispersion" is present. The statistical analysis therefore must deal not only with nonnormality, but also with overdispersion. Without correcting for overdispersion, for example, the test for significance of the treatment (approach) may falsely indicate significance. Fortunately, there are statistical procedures to deal with this [1]. These procedures compute test statistics to see if a true Poisson regression is appropriate, or if a correction for overdispersion must be applied.

Some authors [5], [7] suggest, when analyzing data for which the assumptions of normality are questionable[1], that nonparametric tests may be more appropriate. However, it also is known that parametric tests on means and in experiments where cell sizes are equal (characteristics of our experiments), are fairly robust against deviations from normality [17]. Moreover, though it is easy to perform standard nonparametric tests such as the Mann-Whitney test [5], [6] to see if the primary effect due to reuse approach is present, these tests lose other information present in the experimental design and in the data (e.g., order, nesting of subjects within order, possible interactions between order and treatment, and actual values of the time and number of defects instead of their ranks). The Mann-Whitney test also is not very powerful for situations where there are many tied scores (as we have with the defect data), though it is possible to adjust for ties. Parametric models therefore allow one to get more information from the data, and are preferred if the models' assumptions are reasonably met.

In the following section, we report the detailed analyses of the parametric tests only. We did perform Mann-Whitney tests on the treatment effect for each experiment. The results of these nonparametric tests were exactly the same as their parametric counterparts, at the same significance level.

V. Results

While there are enough data points to apply the statistical methods used in this paper, the small sample sizes used in our experiments adversely affect the power of the statistical tests. This means that, if indeed there are real differences between layering and direct implementation, our tests might conclude otherwise. One way to improve the power of the tests is to raise the alpha level (the probability of concluding that there is a significant treatment effect when in fact there is none). In each of the analyses that follow, the significance tests are done using the standard alpha level of .05, though a case might be made that a higher alpha level (e.g., .10) would be reasonable.

In the first experiment, the subjects already were very familiar with the component in question (the queue). They had seen, and likely used several times, the standard linked representation structure used in this experiment. Hence, the simplicity and familiarity of the component itself might mask true effects due to the treatment. Prior to the second and third experiments, the partial map component used in these experiments had been studied by the subjects in an exercise in which they did an implementation using hashing. A hashing implementation was, in fact, used as the representation of the partial map component provided in experiments two and three (though the actual code for the implementation used in the experiments likely differed somewhat from that developed by any of the subjects in their exercise).

[1] Our time data also could fall into this category. There is some positive skewness in the data; actual tests for normality are highly volatile on small samples such as we have in our experiments.

TABLE III
Experiment 1—Initial Design/Coding Time Data Summary

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	99.1	41.8	57.2	21.3	78.2	38.7
Direct	94.0	44.3	180.4	65.4	137.2	70.1
Overall	96.6	41.9	118.8	79.0	107.7	63.4

TABLE IV
Experiment 1—Initial Design/Coding Time Analysis

Source	df	Mean Sq.	f
Sequence	1	4466.7	1.84
Subject (within Sequence)	16	2434.7	
Sequence × Treatment	1	37056.3	20.73*
Subject × Treatment (within Sequence)	16	1787.5	
Treatment (first period only)	1	6609.7	14.09*
Error	16	469.1	

By using the standard .05 alpha level we felt that, if our experiments were biased at all, they were biased in favor of *not* getting a significant treatment effect when in fact one was present. Therefore, our judgment was that, if the experiment showed a significant treatment effect, it was not likely to be spurious.

A. Experiment 1—Queue Enhancement

The analysis of variance for this experiment, using initial design and coding time, revealed no significant effect for sequence (mean for layering first = 96.6 min., mean for direct first = 118.8 min., $f = 1.84$, critical $F_{1,16,.95} = 4.49$), but did indicate a significant sequence by treatment interaction ($f = 20.73$, $F_{1,16,.95} = 4.49$). The means and standard deviations for each of the four cells are shown in Table III, while the relevant components of the ANOVA table are shown in Table IV. In the ANOVA tables, statistically significant f values (at the .05 level) are indicated by an asterisk.

Following the approach outlined in the previous section, the treatment effect was tested using only the first period data, with the result that the layering approach required significantly less effort ($f = 14.09$, $F_{1,16,.95} = 4.49$).

When total time was used as the dependent variable, a situation similar to that for initial design and coding time was observed. There was no significant effect for sequence (mean for layering first = 163.4 min., mean for direct first = 182.6 min., $f = 0.30$, $F_{1,16,.95} = 4.49$), there was a significant sequence by treatment interaction ($f = 7.16$, $F_{1,16,.95} = 4.49$), and the first period data showed that the layering approach was significantly faster ($f = 7.81$, $F_{1,16,.95} = 4.49$). Tables V and VI contain the relevant data and ANOVA, respectively, for total time.

TABLE V
EXPERIMENT 1—TOTAL TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	145.1	81.4	102.8	47.6	123.9	68.2
Direct	181.8	111.0	262.3	102.4	222.1	111.5
Overall	163.4	96.3	182.6	112.8	173.0	103.8

TABLE VI
EXPERIMENT 1—TOTAL TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	3287.1	0.30
Subject (within Sequence)	16	11087.7	
Sequence × Treatment	1	33878.8	7.16*
Subject × Treatment (within Sequence)	16	4748.5	
Treatment (first period only)	1	13735.8	7.81*
Error	16	1759.6	

TABLE VII
EXPERIMENT 1—DEFECT DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	0.89	0.93	1.33	1.41	1.11	1.18
Direct	2.67	3.57	4.22	3.99	3.44	3.76
Overall	1.78	2.69	2.78	3.26	2.28	2.99

TABLE VIII
EXPERIMENT 1—DEFECT ANALYSIS (POISSON REGRESSION)

Source	df	Mean Sq.	f
Sequence	1	1.38	0.48
Treatment	1	7.84	2.72
Sequence × Treatment	1	<0.01	<0.01
Error	32	2.88	

The Poisson analysis of the defect data revealed neither a significant sequence effect (mean for layering first = 1.78, mean for direct first = 2.78, $f = 0.48$, $F_{1,32,.95} = 4.16$) nor a significant sequence by treatment interaction ($f < 0.01$, $F_{1,32,.95} = 4.16$). Tables VII and VIII contain the relevant statistics. There was no significant treatment effect after correcting for overdispersion ($f = 2.72$, $F_{1,32,.95} = 4.16$).

B. Experiment 2—Partial Map Enhancement

The analysis of initial design and coding time for the partial map enhancement was similar to that for the queue enhancement. There was no significant sequence effect, but there was a significant sequence by treatment interaction (Tables IX and X). The test for the treatment effect, using only the first period data, revealed that the layering approach required significantly less effort ($f = 18.87$, $F_{1,8,.95} = 5.32$).

The analysis of total time gave results similar to that of initial design and coding time. Tables XI and XII show the data and analysis; the treatment test using only the first period data was significant in favor of layering ($f = 14.35$, $F_{1,8,.95} = 5.32$). Again, this replicated the results of Experiment 1.

The Poisson analysis of the defect data showed no significant effect either for sequence or sequence by treatment interaction. The treatment test, after correcting for overdispersion, showed a significant difference in favor of layering (i.e., the layering approach gave rise to significantly fewer defects). This result differed from that in Experiment 1, where no significant effect was observed (see Tables XIII and XIV for details).

TABLE IX
EXPERIMENT 2—INITIAL DESIGN/CODING TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	69.8	16.6	119.0	103.8	94.4	74.7
Direct	127.6	29.6	273.2	131.1	200.4	118.0
Overall	98.7	37.9	196.1	138.0	147.4	110.4

TABLE X
EXPERIMENT 2—INITIAL DESIGN/CODING TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	47433.8	3.63
Subject (within Sequence)	8	13082.6	
Sequence × Treatment	1	11616.2	7.86*
Subject × Treatment (within Sequence)	8	1477.5	
Treatment (first period only)	1	41371.6	18.87*
Error	8	2912.0	

TABLE XI
EXPERIMENT 2—TOTAL TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	113.0	62.9	160.6	147.3	136.8	109.7
Direct	191.0	40.5	458.2	236.1	324.6	212.9
Overall	152.0	64.6	309.4	242.9	230.7	190.9

TABLE XII
EXPERIMENT 2—TOTAL TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	123873.8	3.37
Subject (within Sequence)	8	36725.2	
Sequence × Treatment	1	60280.2	12.59*
Subject × Treatment (within Sequence)	8	4786.8	
Treatment (first period only)	1	119163.0	14.35*
Error	8	8302.4	

TABLE XIII
EXPERIMENT 2—DEFECT DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	0.80	1.79	0.80	0.84	0.80	1.32
Direct	2.60	1.67	6.60	2.61	4.60	2.95
Overall	1.70	1.89	3.70	3.56	2.70	2.96

TABLE XIV
EXPERIMENT 2—DEFECT ANALYSIS (POISSON REGRESSION)

Source	df	Mean Sq.	f
Sequence	1	4.24	2.37
Treatment	1	16.52	9.23*
Sequence × Treatment	1	0.79	0.44
Error	16	1.79	

TABLE XV
EXPERIMENT 3—INITIAL DESIGN/CODING TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	65.4	28.8	50.6	33.5	58.0	30.5
Direct	112.4	65.1	133.8	52.6	123.1	56.9
Overall	88.9	53.5	92.2	60.4	90.6	55.6

TABLE XVI
EXPERIMENT 3—INITIAL DESIGN/CODING TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	54.5	0.02
Subject (within Sequence)	8	3609.0	
Treatment	1	21190.1	24.42*
Sequence × Treatment	1	1638.1	1.89
Subject × Treatment (within Sequence)	8	867.8	

TABLE XVII
EXPERIMENT 3—TOTAL TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	157.8	99.4	112.0	83.6	134.9	89.9
Direct	193.2	112.9	311.8	327.4	252.5	239.2
Overall	175.5	102.0	211.9	248.7	193.7	185.9

TABLE XVIII
EXPERIMENT 3—TOTAL TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	6624.8	0.13
Subject (within Sequence)	8	49544.4	
Treatment	1	69148.8	3.67
Sequence × Treatment	1	33784.2	1.79
Subject × Treatment (within Sequence)	8	18859.9	

C. Experiment 3—Partial Map Modification

In the analysis of initial design and coding time for the partial map modification experiment, there was no significant effect for sequence nor for the sequence by treatment interaction. The analysis of the treatment effect, using all of the data, revealed a significant treatment effect in favor of layering (Tables XV and XVI).

The analysis of total time also revealed no significant sequence effect, nor a significant sequence by treatment interaction. The treatment effect was not significant (Tables XVII and XVIII).

TABLE XIX
EXPERIMENT 3—DEFECT DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	3.00	2.45	1.00	1.22	2.00	2.11
Direct	5.00	4.85	2.80	2.49	3.90	3.81
Overall	4.00	3.77	1.90	2.08	2.95	3.15

TABLE XX
EXPERIMENT 3—DEFECT ANALYSIS (POISSON REGRESSION)

Source	df	Mean Sq.	f
Sequence	1	2.57	0.84
Treatment	1	2.09	0.70
Sequence × Treatment	1	0.25	0.09
Error	16	2.98	

The Poisson analysis of the defect data revealed no significant sequence effect nor a significant sequence by treatment interaction. After correcting for overdispersion, the test on treatment was not significant (Tables XIX and XX).

D. Discussion

For this set of experiments, we found that the use of layering consistently resulted in significantly faster time to complete the initial design and coding of the new component. This result held as the component changed from one with which the subjects were quite familiar to one with which the subjects were less familiar (and which was more complex as measured by the number of operations encapsulated in it and total lines of code). The result also held as the task changed from a component enhancement to a component modification.

Total time, including that for testing, debugging and recoding, also was significantly better using layering when the component was an enhancement. This result held for both the simple and familiar queue, and the more complicated and less familiar partial map. The modification task provided no such significant effect, though the mean total time for layering was much less than that for direct implementation. One possible explanation for this is that some defects made in the modification task tended to be nastier than those made in the enhancement task. It turned out that the average number of defects per subject was slightly higher for the enhancement task than for the modification task. If some of the defects for the modification task were, indeed, trickier, the time to debug and repair these defects would occupy a greater fraction of total time. Apparently, the *nature* of the defects was such that this debugging and repair time was not a function of the treatment, so the gain for layering in initial design and coding time is ameliorated when the debugging and repair time is added. Note that this could mean that trivial defects were just as trivial and nasty defects were just as nasty, whether layering or direct implementation was used.

No consistent effects in favor of layering were found for the defect data, though the mean number of defects was less

for layering in each experiment. Here the small scale of the experiment may influence the results. Almost every subject had fewer than five defects. Some of these defects likely were trivial and these relatively trivial defects might be just as likely to be made when using direct implementation as they are when using layering. If the fraction of relatively trivial defects was high for many of the subjects, it then will be difficult to obtain statistical significance. With the small number of defects observed in these data, it is somewhat remarkable that we obtained *any* significant effects for this dependent variable.

A final item worth noting is that, had the analysis of the defect data for experiment 1 *not* corrected for overdispersion, it would falsely have concluded that there *was* a significant effect favoring layering. This illustrates how using the wrong statistical analysis in software engineering experiments such as these can mistakenly support the use of a particular technology even when the data do not really indicate such support.

VI. Conclusion

The results of our experiments support the contention that, by using only a description of the functionality of and interfaces to existing components, new components can be developed with less effort than that required if the source code and representation data structures of the existing components are also used. In addition, it appears from our experiments that there certainly is no loss in the quality of the development process, at least in terms of the number of defects made during development, when the layering approach is used.

The empirical studies described in this paper illustrate interesting issues in the statistical analysis procedure, issues which, based on the authors' experiences, are not well-known to software engineering researchers. It is important that the proper analysis is used, lest the wrong conclusions be reached regarding the benefits of a particular method and/or best use is not made of the information contained in the experimental design and the data collected.

The subjects used in our experiments, while mature students many of whom had full-time jobs involving software development, might not be representative of the typical programmer. Generally, they had only a couple of years' experience in commercial software development. Subjects with different backgrounds might perform differently on our experimental tasks; this is a potential avenue for future research.

It also might be interesting to compare the actual amount of code written by the subjects when using layering with that written when using direct implementation, to see if layering required less "work." If so, then the amount of work required (measured by required changes to the code) would be an alternative explanation of our results for the time and defect data. We did not collect this "code change" data. Of course, from an abstract point of view, the amount of change required when using layering was identical to that required when using direct implementation, since the functionality required was the same in each case. Moreover, we believe that a software engineer, when faced with the choice of using layering or direct implementation, would find it difficult to estimate in advance which approach would require less work, even when (as in our experiments) the engineer is quite familiar with the representations used in the direct implementation.

We are planning further experiments to more carefully analyze the defects made in the development of components such as those studied herein. It is important not only to characterize the kinds of defects observed, but also to provide if possible some cognitive explanation of these observations. We also are planning other studies to try to replicate the results reported herein. Studies such as these serve to provide a more sound and scientific basis for using (or not using) various software engineering methods.

Acknowledgment

The authors thank R. Leighty of Ohio State University's Department of Statistics for his assistance with the statistical analyses, and the referees and editor for their helpful comments.

References

[1] M. Aitkin *et al.*, *Statistical Modeling in GLIM*. New York: Oxford, 1989.
[2] G. Booch, *Software Components in Ada*. Menlo Park, CA: Benjamin Cummings, 1987.
[3] R. Brooks, "Studying programmer behavior experimentally: The problems of proper methodology," *Commun. ACM*, vol. 23, no. 4, pp. 207–213, Apr. 1980.
[4] S. Chidamber and C. Kemerer,"A metrics suite for object-oriented design," *IEEE Trans. Software Eng.*, vol. 20, pp. 476–493, June 1994.
[5] S. Conte, V. Shen, and H. Dunsmore, *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin Cummings, 1986.
[6] W. Daniel, *Applied Nonparametric Statistics*, 2nd ed. Boston, MA: PWS-Kent, 1990.
[7] N. Fenton, *Software Metrics: A Rigorous Approach*. London, U.K.: Chapman and Hall, 1991.
[8] D. Harms and B. Weide, "Copying and swapping: Influences on the design of reusable software components," *IEEE Trans. Software Eng.*, vol. 17, pp. 424–435, May 1991.
[9] C. Hicks, *Fundamental Concepts in the Design of Experiments*, 2nd ed. New York: Holt, Rinehart and Winston, 1973.
[10] J. Hollingsworth, B. Weide, and S. Zweben, "Confessions of some used-program clients," in *Proc. Fourth Annu. Workshop Software Reuse* (Herndon, VA), Nov. 1991.
[11] J. Hollingsworth, "Software component design-for-reuse: A language-independent discipline applied to Ada," Ph.D. dissertation, Dept. Comput., Info. Sci., Ohio State University, Columbus, OH, Aug. 1992.
[12] W. R. LaLonde, "Designing families of data types using exemplars," *ACM Trans. Programming Languages, Syst.*, vol. 11, no. 2, pp. 212–248, 1989.
[13] J. A. Lewis *et al.*, "An empirical study of the object-oriented paradigm and software reuse," in *Proc. 1991 OOPSLA Conf.*, pp. 184–196.
[14] G. Milliken and D. Johnson, *Analysis of Messy Data, Vol. 1: Designed Experiments*. Princeton, NJ: Van Nostrand Reinhold, 1984.
[15] R. Pressman, *Software Engineering: A Practitioner's Approach*, 3rd ed. New York: McGraw-Hill, 1992.
[16] D. Ratkowski, M. Evans, and J. R. Alldredge, *Cross-Over Experiments*. New York: Marcel Dekker, 1993.
[17] H. Scheffe, *The Analysis of Variance*. New York: Wiley, 1959.
[18] M. Sitaraman and B. Weide, Eds., "Special feature: Component-based software using RESOLVE," *ACM SIGSOFT Software Eng. Notes*, vol. 19, no. 4, pp. 21–67, Oct. 1994.
[19] E. Soloway and S. Iyengar, Eds., *Empirical Studies of Programmers*. Norwood, NJ: Ablex, 1986.
[20] I. Sommerville, *Software Engineering*, 4th ed. Reading, MA: Addison-Wesley, 1992.

[21] B. Weide, W. Ogden, and S. Zweben, "Reusable software components," in *Advances in Computers*, vol. 33, M. C. Yovits, Ed. New York: Academic, 1991, pp. 1–65.

Stuart H. Zweben received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN.

He is Professor and Chair of the Department of Computer and Information Science at The Ohio State University, Columbus. His research interests are in the areas of software quality evaluation and software reuse, and he codirects the Reusable Software Research Group at Ohio State.

Dr. Zweben is current President of the ACM, is former President of the Computing Sciences Accreditation Board, and is a member of the editorial board of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. He also is a member of the IEEE Computer Society, ACM, Upsilon Pi Epsilon, and the American Association of University Professors.

Stephen H. Edwards received the B.S.E.E. degree from the California Institute of Technology, Pasadena, and the M.S. degree in computer and information science from The Ohio State University, Columbus, where he is working toward the Ph.D. degree in computer and information science.

Prior to attending Ohio State, he was a Member of the Research Staff at the Institute for Defense Analyses. His research interests are in software engineering and reuse, formal models of software structure, programming languages, and information retrieval technology.

Mr. Edwards is a member of the IEEE Computer Society and the ACM.

Bruce W. Weide received the B.S.E.E. degree from the University of Toledo, Toledo, OH, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA.

He is Associate Professor of Computer and Information Science at The Ohio State University, Columbus, and Codirector of the Reusable Software Research Group with B. Ogden and S. Zweben. His research interests include all aspects of software component engineering, especially in applying RSRG work to Ada and C++ practice.

Dr. Weide is a member of the ACM and CPSR.

Joseph E. Hollingsworth received the B.S.C.S. degree from Indiana University, the M.S.C.S. from Purdue University, West Lafayette, IN, and the Ph.D. degree in computer and information science from The Ohio State University, Columbus.

He is an Assistant Professor of Computer Science at Indiana University Southeast, New Albany. His research interests include software component design disciplines for C++ and Ada and the application of those disciplines to real-world computing problems.

Dr. Hollingsworth is a member of the IEEE Computer Society and the ACM.