A Survey of the Usage of Encapsulation in Object-Oriented Programming

Mats Skoglund
Stockholm University/Royal Institute of Technology
Department of Computer and Systems Sciences
Forum 100, S-164 40 KISTA, SWEDEN
Phone: +46 8 161536, Fax: +46 8 7039025
`matte@dsv.su.se`

July 22, 2003

Abstract

In object-oriented programming the concept of encapsulation is used to create abstract datatypes that should be possible to modify only through their external interface. One way to enforce this kind of encapsulation is to declare variables with different access modifiers, such as private or public. However, when using private variables in programming languages with reference semantics, such as *e.g.* Java, it is only the names of the variables that are protected, not the real objects pointed to by the variables. This problem is sometimes referred to as the *representation exposure* problem and many proposals addressing this problem have been presented.

There is, however, a lack of empirical data on how widespread this problem is in the software industry and its effect on software quality.

In this paper we report our finding from a questionnaire survey conducted among software engineers on their view on encapsulation and information hiding issues, their use of OO programming languages, their way of working and their opinions on tools and techniques supporting encapsulation, information hiding and representation exposure.

1 Introduction

In object-oriented programming the concept of encapsulation is used to create abstract datatypes that should be possible to modify only through their external interface. This helps to create programs that are easier to maintain, debug and reason about [2]. One way to enforce this kind of encapsulation is to declare variables with different access modifiers, such as private or public. However, when using private variables in programming languages with reference semantics, such as *e.g.* Java, it is only the names of the variables that are protected, not the real objects pointed to by the variables. Since references to objects can be returned from method calls, it is possible for an object that is external to *e.g.* a compound object to obtain references to objects held in the private variables of the compound object. By invoking methods directly on the received reference, invariants of the compound object that are preserved by the interface methods may be broken and this may lead to inconsistencies in the system or to a system crash [10, 13]. This problem is sometimes referred to as the *representation exposure* problem [17] and many proposals have been presented addressing this problem [9, 1, 13, 8, 2, 11, 16, 21, 22, 18].

There is, however, a lack of empirical data on how widespread this problem is in the software industry and its effect on software quality.

In this paper we report our finding from a questionnaire survey conducted among software engineers on their view on encapsulation and information hiding issues, the representation exposure problem, their use of object-oriented programming languages, their way of working and their opinions on tools and techniques supporting object-oriented development with respect to encapsulation and representation exposure. This questionnaire survey was a follow-up study on a previously conducted series of structured interviews among software engineers on the same subject [20].

The outline of this paper is as follows. Section 2 presents related empirical studies that have been conducted regarding the use of encapsulation in object-oriented programming languages. In Section 3 the design of the study is detailed. The analysis of the responses is reported in Section 4. The validity of the survey is discussed in Section 5 and Section 6 contains concluding remarks.

2 Motivation and Related Work

The use of encapsulation and information hiding is emphasized in the object-oriented literature because it is said to cut development time, improve the quality of the software produced and facilitate maintenance [25]. There are several ways that may be taken to enhance the level of encapsulation and information hiding in programs. Guidelines and coding standards that advise the programmer to code in certain ways may for example have a positive effect on the code produced if the programmers have the discipline to follow them.

There also exist tools and techniques supporting the concept of encapsulation and information hiding. For example, in Smalltalk the member variables cannot be accessed outside the class by default and thus the programmer is forced to write setter and getter methods to give external access to the objects referred by them, thus reducing the risk of breaking encapsulation by mistake.

Automatic code generation tools may also support the programmer by generating code where member variables are declared with restricted access, *e.g.* as private, and also generate getter and setters methods for those variables. This should also reduce the risk of variables becoming more accessible than necessary just by mistake.

Static analysis tools may also be used to analyse programs and for example point out those member variables that are declared as public.

The existing tools, methods and techniques for encapsulation, information hiding and representation exposure need not, however, necessarily be used by the programmer. For example, declaring a member variable as private must in Java explicitly be stated by the programmer.

Unfortunately, there is a lack of empirical indications on whether encapsulation and information hiding concepts are actually enforced by the programmers in the software industry.

By analysing Smalltalk classes Menzies and Haynes [14] investigated the level of encapsulation and information hiding in Smalltalk programs. Their hypothesis was that if the program adheres to the principle of encapsulation there should be sparse calls to other classes. However, the analysis of 2000 classes in 5 applications show a low usage of information hiding.

Elish and Offut have examined 100 open-source Java classes and their adherence to 16 different coding practices by using static analysis tools [5]. Their results show the third most broken coding practice is that member variables should not be public. This practice is violated in 15% of the classes examined.

Fleury interviewed 28 students learning object-oriented programming about their understanding of different object-oriented concepts [7]. In this study many students considered reducing the number of lines of code and the number of classes to be more important than encapsulation. For example, some students considered a class with no accessor methods and where all member variables were declared public to be better than the same class with private member variables and accessor methods due to the smaller code size.

There are also a lack of empirical evidence on the reasons for not using techniques for controlling encapsulation and information hiding. Evered and Menger argue that students have trouble using encapsulation and information hiding in C++ since the language does not allow separation of the declaration of the public member variables from the private [6]. Thus, one reason could be that the support from the programming languages is insufficient.

Another possible reason is that other techniques used may have negative impacts on the degree of encapsulation in the final programs. For example, as noted by Snyder [23], the inheritance mechanism in Smalltalk gives the subclass full access to the superclass' member variables and thus the encapsulation is broken and the possibility to change the superclass without affecting the clients gets more complicated.

Another reason could be that object-orientation is difficult to teach and thus those who learn object-orientation do not comprehend the encapsulation and information hiding concepts enough to use it properly. However, Klling argues in [12] that the underlying reason for this could be that the programming languages used for teaching object-orientation do not have concepts that are clean, consistent and easy to understand.

There is however a need to provide stronger empirical evidence than currently exists on software engineers' use of encapsulation, information hiding and representation exposure. We therefore conducted a questionnaire survey to gather more empirical data in this area.

3 The Design of the Study

The purpose of the questionnaire survey was to gather quantitative data regarding software engineers' way of working with encapsulation, information hiding and representation exposure. Another purpose was to get opinions on how tools and techniques would affect their way of working and the quality of the software produced.

3.1 Questionnaire Design

The structured interview template from the earlier conducted interviews was the starting point when designing the questions for the questionnaire [20]. However, for easier analysis, we introduced closed questions to avoid the open type questions that were used in those interviews. The layout of the questionnaire followed the guidelines in [4]:

An introductory part describing the motivation of the study and instructions on how to fill in the form and how to return the completed form. Then, the questions were divided into different parts depending on their focus. The parts were: questions regarding the respondent's background in programming, object-orientation and code reuse, then questions regarding their use of access modifiers when programming. The next section of the questionnaire was about encapsulation issues and the last part was about tool and technologies and their effect on the quality of software and finally there was a thank note to the respondent.

A web based form of the questionnaire was also created which the respondents could fill in via the Internet. Before distributing the questionnaire we conducted a pilot study to discover errors, ambiguities, inadequate response alternatives and confusing questions. The comments from the pilot respondents were collected verbally and via e-mail and the questionnaire was adjusted accordingly. The main changes made as result from the pilot study were changes to the definitions and explanations of terms and situations. The final questionnaire used in this survey can be found in [19].

Note that the responses resulting from the pilot study were not included in the analysis.

3.2 Population Sampling and Distribution

Our subjects of interest were experienced software engineers. The ability to generalize from a study is dependent on how well the sample drawn from a population represents the target population as a whole [24]. Sampling techniques such as *simple random sample*, *systematic sampling* or *stratified sampling* could not be applied in our study since they require a complete list of the whole population to draw the samples from, which we did not have access to.

The questionnaire was instead distributed as postings to ten different electronic newsgroups on the Internet[1]. This meant that the questionnaire could be read by a variety of people in the software industry and academia, *e.g.* software engineers, programmers, software architects.

[1] comp.lang.c++, comp.lang.eiffel, comp.lang.java.beans, comp.lang.java.programmer, comp.lang.objective-C, comp.lang.pascal.borland, comp.lang.pascal.delphi.misc, comp.lang.smalltalk, comp.object, comp.software-eng.

The posting included the introductory text as well as the questionnaire together with a hyperlink to the web based version of the questionnaire. After eight days the questionnaire was reposted to get more responses. The web based questionnaire was closed after thirty days from the first posting and no more responses were accepted.

4 Analysis

A total of 65 responses were received, 2 via mail and 63 via the web based form. We excluded 4 blank responses and the remaining 61 responses were then analysed. This section presents the analysis of the collected data from the survey. The frequency statistics for each questions can be found in [19]

All respondents had 1 year or more of professional experience in programming and more than 60% of the respondents had more than 6 years of experience. None of the respondents had less than 1 year of experience of object-orientation and 64% had more than 4 years of experience.

Table 1 presents the respondents' experience in programming, object-orientation and code reuse. The most familiar language among the respondents was C++ followed by Java and Smalltalk. The respondents' experience in different programming languages is detailed in Table 2. The percentages presented in the table are calculated relative to the number of respondents, *i.e.* 61. The "Other" column includes the languages CLOS, Common Lisp, Objective-C, Perl, Smallscript and PostScript in NeWS.

	Respondents' experience				
	< 1 year	1-2 years	3-4 years	5-6 years	> 6 years
Prog.	0 (0%)	4 (7%)	7 (11%)	12 (20%)	38 (62%)
OO	0 (0%)	10 (16%)	12 (20%)	39 (64%)	n/a
Reuse	9 (15%)	19 (31%)	10 (16%)	23 (38%)	n/a

Table 1: Respondent's experiences in programming, OO and reuse

Object-Oriented language experience						
C++	C#	Eiffel	Java	Pascal (Delphi)	Object Smalltalk	Other
43 (70%)	4 (7%)	4 (7%)	31 (51%)	7 (11%)	18 (30%)	11 (18%)

Table 2: Language familiarity

Of the 61 respondents 52 (85%) responded that their code is reused. All of the respondents that said their code is reused also responded that one way their code is reused is by instantiation of their classes. The second most common form of reuse of the respondents' code is subclassing (65%), followed by the use of class methods (58%).

4.1 The Use of Access Modifiers

The respondents were asked to select the two access levels they mostly assign to member variables when programming.[2]

56 of the respondents (92%) use private or protected as their first or second choice of access level. Diagram showing the distribution of the use of access modifiers is presented in Figure 1.

Of the 61 respondents 26 (43%) responded that they had not declared a member variable with an access modifier other than private or protected in the last 12 months.

[2]21 of the 61 respondents selected only one of the alternatives given but from their comments provided we could infer a second alternative for some of the respondents and the data were adjusted accordingly

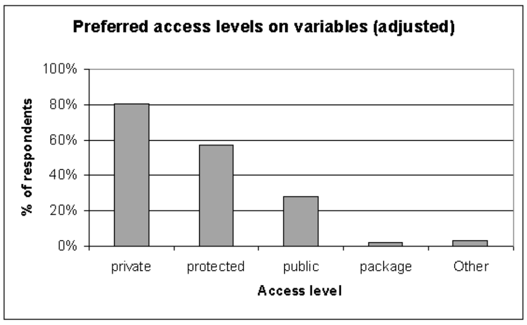

Figure 1: The respondents' use of access modifiers

We were also interested in the reasons behind the programmers' choice of access levels. From our previously conducted interviews we derived three closed questions and one open question regarding this. The three closed questions included performance, use of design patterns and avoiding writing getter and setter methods as possible reasons for choosing an access modifier other than private or protected when declaring member variables. The most frequent reason was to avoid writing getter and setter methods. One third of the respondents (33%) said this has been the reason in the last 12 months. The second most frequent reasons were performance reasons (20%) and the use of design patterns (20%).

4.2 Defects, Code Review and Quality Assurance

On the question of whether the respondents in some way assure that they have declared member variables with, in their opinion, an appropriate access modifier, 45 of the respondents (75%) responded that they do. They were also asked to specify how they assure that they have the appropriate access level. Four alternatives were given: (i) Constantly using the (in their own opinion) "correct" access modifier when declaring member variables, (ii) explicitly reviewing their own code in this respect and (iii) having the code reviewed in organized code inspections and (iiii) other.

The most common form of assurance among the respondents is to consistently use the "correct" access modifiers when declaring member variables. Not surprisingly, the vast majority (82%) use this method of assuring that they have declared the member variables appropriately.

Only 4 of the respondents who said they perform some assurance activity use all three of the alternatives given. Furthermore, one third of them (33%) use both of the first two alternatives for assuring their code in this respect. The majority of the respondents (73%) had never participated in any formal code review where the access modifiers were explicitly reviewed. The distribution

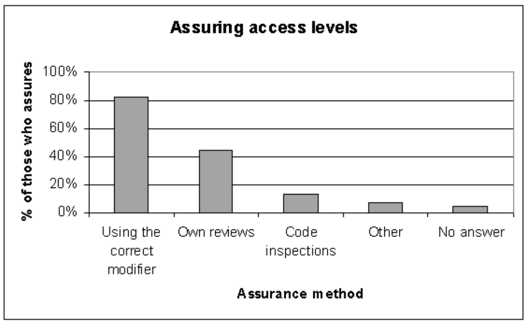

Figure 2: Methods for assuring appropriate access level

of the different methods used is presented in Figure 2.

4.3 Representation Exposure Problems

From the structured interviews we received indications that situations where objects' internal representations are exposed and checks bypassed are common in programming and the results from this survey give the same indication. More than half of the respondents (59%) responded that they had discovered this kind of situation in the last 12 months, 12% of the respondents more than 10 times. Also, one third of the respondents who had discovered these kinds of situations had also discovered defects due to this. Furthermore, 41% of those who had discovered such situations had also rewritten the code. Respondents were asked whether they had participated in organized code inspections regarding representation exposure in the last 12 months. The majority of the respondents (81%) responded that they had not participated in such reviews.

4.4 The Opinions on the Usefulness of Methods, Tools and Techniques

To examine the attitude regarding the usefulness of methods, tools and techniques for preventing representation exposure, respondents were asked to grade their opinion on three hypothetical aids on a scale from 0 (not useful) to 5 (very useful) :

1. A measure of how much classes expose internal representation
2. A tool showing where in the code internal representations are exposed
3. A programming language construct that can prevent internal objects from being exposed

While a small subset of the respondents (13%) answered that none of the aids could be useful at all, one fourth of the respondents gave a usefulness of 3 or higher. More than half of the respondents (62%) could find at least some usefulness in all the aids, *i.e.* gave a grade above 0. The opinions on the usefulness of tools and techniques could perhaps be dependent on the

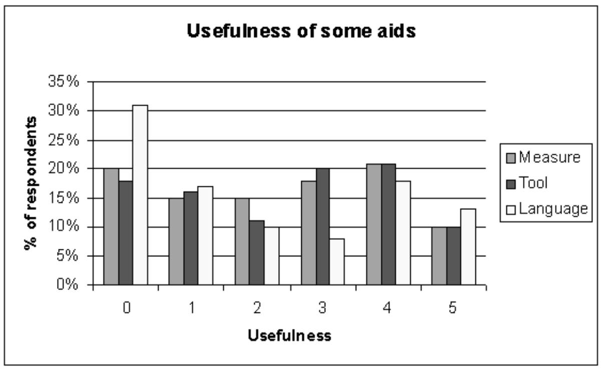

Figure 3: Respondents' opinion on the usefulness of different aids

amount of experience. Perhaps programmers with less experience believe aids are more useful than those with longer experience or perhaps experienced programmers can see an advantage of an aid for certain issues that less experienced programmers do not see. Chi-square tests were calculated to check if such relations exist. We tested those respondents with more than 6 years of experience in programming with those who gave at least the value 3 on the scale of usefulness for measure, tool and programming language. None of the tests provided any significant result. (measure:) $df = 1$, $\mathcal{X}^2 = 1.148$, $\rho \leq 1$, (tool:) $df = 1$, $\mathcal{X}^2 = 0.707$, $\rho \leq 1$, (language:) $df = 1$, $\mathcal{X}^2 = 0.012$, $\rho \leq 1$.

The opinions regarding a programming language that could be used to control representation exposure were somewhat different from the opinions on the other aids. The majority of the respondents (60%) graded the usefulness of a language below 3 and the most frequently selected alternative for this question, selected by more than one fourth of the respondents (27%), was a grade of 0, *i.e.* not useful. However, the second most chosen alternatives were 1 and 4 (11%) and the third most selected was 5 (8%) so the respondents disagree regarding the usefulness of a language. A diagram of the distribution of the responses for the aids is presented in Figure 3.

The last closed question on the questionnaire was how much impact an aid for representation exposure would have on the quality of software produced. Not surprisingly, none of the respondents thought that an aid would lead to the number of defects being increased. The greatest proportion of the respondents (52%) said that the number of defects would not be affected at all, while 43% said that it would slightly decrease the number of defects in their code. Perhaps experience affects the opinion in this respect. A programmer with not so much experience may perhaps believe that the use of an aid has a decreasing impact on the number of defects than do more experienced programmers. However, Chi-square tests did not indicate any significance for the relation between those who had more than 6 years of experience in programming or more than 4 years experience in object-orientation and the opinion regarding the impact on the number of defects ($df = 1,\ \mathcal{X}^2 = 0.35,\ \rho \leq 1$ and, $df = 1,\ \mathcal{X}^2 = 0.11,\ \rho \leq 1$ respectively).

5 Validity of the Survey

A small number of respondents (4) mentioned that the question regarding their use of access modifiers was not relevant for them since they were Smalltalk programmers. Fortunately, we could from those respondents' comments infer a closed alternative according to Smalltalk's default access levels and adjust the data accordingly. However, we are not able to determine whether respondents other than those who provided comments made the same judgment regarding this question and thus perhaps affected the outcome.

Also, the questions regarding different reasons for not declaring variables as private or protected could perhaps be misunderstood. The reason for avoiding writing getters and setters could perhaps be to gain performance or because a design pattern requires it. This could perhaps lead to those who had experienced that performance or design patterns had been reasons for declaring variables as non-private or non-protected also regarding avoiding getters/setters as reasons automatically. However, since none of the respondents commented on this we believe that the number of misunderstandings due to this was rather few.

A threat to the external validity, *i.e.* the potential to draw general conclusions from a small sample of a population, is dependent on how representative the sample is. We selected newsgroups that have a large proportion of the individuals who are part of our target population. However nothing can be said about how representative the readers of the specific newsgroups are of the software engineering community as a whole. We still believe the responses from 61 experienced software engineers should be considered important.

Also, only the readers of the newsgroups received the questionnaire and only those motivated enough responded, thus the method suffers from self selection. However, some degree of self-selection exists for all distribution methods [3]. A study conducted by Daly compared the difference between newsgroup distributed questionnaires and mail distributed questionnaires [4]. This study did not show any significant difference in opinions of the respondents across the media used. The problem of self selection in the newsgroup distributed questionnaire appeared to be of a similar order to that in the mail distributed questionnaire. Our study had the same target group as the study conducted by Daly, we posted our questionnaire to partly the same newsgroups as in his study and our area of interest was partly the same as his, *i.e.* object-orientation. Thus, we believe that the level of self-selection in our study is comparable to his, *i.e.* similar to the degree of self-selection for a similar mail distributed questionnaire.

6 Concluding Remarks

The results from our survey show that some programmers use public variables as their first two choices of access modifiers when programming in object-oriented programming languages.

In The Code Conventions for the Java Programming Language [15] it states: “Don’t make any instance or class variable public without good reason” and a “good reason” could, according to [15], be the use of classes as records or structs with no behavior. We found that performance, using patterns and avoiding writing getter and setters are three reasons why publicly accessible variables are used. Whether these reasons are “good reasons” in some respects has however not been investigated.

Representation exposure situations are common in object-oriented programs according to the respondents. They also agree that defects in programs exist due to representation exposure. This could justify the use of tools and techniques for minimizing representation exposure in programs. The kinds of tools or techniques that would be most appropriate are however unclear. There does not seem to exist a silver bullet for the representation exposure issues either. The software engineers’ opinions on the usefulness of a tool, a measure or a programming language for controlling, measuring or providing information about representation exposure differ. The reasons for this disagreement were not thoroughly investigated in this study but it does not seem to be dependent on the software engineers’ experience in either programming or object-orientation.

However, software engineers do not believe that the number of defects would change very much if they had access to more information regarding the degree of representation exposure in their programs when programming by using some tool or technology. The vast majority of software engineers believe that the number of defects would not change at all or just decrease slightly. This indicates that the use of tools or techniques for representation exposure is *not* justified. We conclude that further research is needed to clear up these somewhat ambiguous findings.

References

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP ’97 Proceedings*, 1997.

[2] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA ’98 Proceedings*, 1998.

[3] J. Daly, J. Miller, A. Brooks, M. Roper, and M. Wood. Issues on the object-oriented paradigm: A questionnaire survey. Technical report, Department of Computer Sciencee, University of Strathclyde, Glasgow, 1995.

[4] J. W. Daly. *Replication and a Multi-Method Approach to Empirical Software Engineering Research.* PhD thesis, Department of Computer Science, University of Strathclyde, Glasgow, 1996.

[5] M. O. Elish and J. Offutt. The adherence of open source java programmers to standard coding practices. In *6:th IASTED International Conference on Software Engineering and Applications*, November 2002.

[6] M. Evered and G. Menger. Using and teaching information hiding in single-semester software engineering projects. In *Proceedings of the Australasian computing education conference*, pages 103–108. ACM Press, 2000.

[7] A. E. Fleury. Encapsulation and reuse as viewed by java students. In *32nd SIGCSE Technical Symposium on Computer Science Education*, February 2001.

[8] H. Hakonen, V. Leppnen, T. Raita, T. Salakoski, and J. Teuhola. Improving object integrity and preventing side effects via deeply immutable references. In *Proceedings of Sixth Fenno-Ugric Symposium on Software Technology, FUSST*, 1999.

[9] J. Hogg. Islands: Aliasing protection in object-oriented languages. In A. Paepcke, editor, *OOPSLA '91 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285. ACM Press, 1991.

[10] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. In *OOPS Messenger 3(2), April 1992*, 1992.

[11] K. Izuru. An object-oriented analysis and design approach for safe object sharing. In *Proceedings of The Seventh International Conference on Engineering of Complex Computer Systems*, 2001.

[12] M. Klling. The problem of teaching object-oriented programming. *Journal of Object-oriented programming*, 8(11):8–15, 1999.

[13] G. Kniesel and D. Theisen. Java with transitive access control. In *IWAOOS'99 – Intercontinental Workshop on Aliasing in Object-Oriented Systems. In association with the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, June 1999.

[14] T. Menzies and P. Haynes. Empirical observations of class-level encapsulation and inheritance. Technical report, Department of Software Development, Monash University, 1996.

[15] S. Microsystem. Code conventions for the java programming language, 1999. Available at $http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html$.

[16] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[17] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998.

[18] M. Skoglund. Sharing objects by readonly references. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology*, number 2422 in Lecture Notes in Computer Science, pages 457–472, Berlin, Heidelberg, New York, 2002. Springer-Verlag.

[19] M. Skoglund. *Investigating Object-Oriented Encapsulation in Theory and Practice*. Licentiate thesis, Stockholm University/Royal Institute of Technology, March 2003.

[20] M. Skoglund. Practical use of encapsulation in object-oriented programming. In *Proceedings of The 2003 International Conference on Software Engineering Research and Practice*, 2003.

[21] M. Skoglund and T. Wrigstad. A mode system for readonly references. In kos Frohner, editor, *Formal Techniques for Java Programs*, number 2323 in Object-Oriented Technology, ECOOP 2001 Workshop Reader, pages 30–, Berlin, Heidelberg, New York, 2001. Springer-Verlag.

[22] M. Skoglund and T. Wrigstad. Alias control with read-only references. In *Proceedings of 6th International Conference on Computer Science and Informatics, CSI*, pages 457–472, 2002.

[23] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 38–45, New York, NY, 1986. ACM Press.

[24] C. Wohlin, P. Runesson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering, An Introduction*. Kluwer Academic Publishers, 2000.

[25] S. H. Zweben, S. H. Edwards, B. W. Weide, and J. E. Hollingsworth. The effects of layering and encapsulation on software development cost and quality. *IEEE Transactions on Software Engineering*, 21(3), March 1995.