Practical Use of Encapsulation in Object-Oriented Programming

Conference Paper · January 2003

Source: DBLP

CITATION
1

READS
1,450

1 author:

Mats Skoglund
Lund University
13 PUBLICATIONS **421** CITATIONS

SEE PROFILE

Practical Use of Encapsulation in Object-Oriented Programming

Mats Skoglund
Department of Computer and System Sciences
Stockholm University/Royal Institute of Technology
Kista, Sweden
matte@dsv.su.se

Abstract

Even though an OO program may have a high degree of encapsulation it is still sometimes possible to modify the inner representation of compound objects. In for example Java there is little to prevent references exported from a compound object to be used by its receivers. Thus it may be possible to change the states of inner objects of a compound object from the outside leading to invariants may be broken. This is often referred to as the representation exposure problem and many solutions to this problem have been proposed. There is, however, a lack of empirical evidence that this is actually a practical problem in the software industry.

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

Keywords: *object-orientation, empirical, encapsulation, representation exposure, programming*

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 somewhat control 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 and not the actual 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 such a received reference, invariants of the compound object that are preserved by the interface methods may be broken. This may lead to inconsistencies in the system or to a system crash [9, 12]. This problem is sometimes referred to as the *representation exposure* problem [16] and many proposals have been presented addressing this problem, both at the design level and at the programming language level [8, 1, 12, 7, 10, 15, 18, 19, 17].

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

In this paper we report our finding from structured interviews 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.

Section 2 presents related empirical studies that have been conducted regarding the use of encapsulation in object-oriented programming languages. In Section 3 the interview method used is detailed. The analysis of the interviews is reported in Section 4 and Section 5 concludes.

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 [21]. For example, in The Code Conventions for the Java

Programming Language [14] it states : "Don't make any instance or class variable public without good reason". Such guidelines and coding standards that advise the programmer to code in certain ways may 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. In for example Smalltalk member variables are by default inaccessible from outside their classes. Thus, the programmer is forced to write getter and setter methods to give external access to the objects referred to by the member variables. This should reduce the risk of the programmer breaking encapsulation just 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 only when needed 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 be used to analyse programs and highlight encapsulation weaknesses to the programmer, for example, by pointing out member variables declared as public.

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

There also exist empirical indications that encapsulation and information hiding are not always enforced by the programmers.

By analysing Smalltalk classes Menzies and Haynes [13] 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 examined 100 open-source Java classes and their adherence to 16 different coding practices by using static analysis tools [4]. Their results show that 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 [6]. In this study many students considered reducing both 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.

The reasons for not using techniques for controlling encapsulation and information hiding could be numerous. One reason could be that the support from the programming languages is insufficient. For example, 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 [5].

Other techniques used may also have negative impacts on the degree of encapsulation in the final program. For example, as noted by Snyder [20], 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 fot not using the existing techniques 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 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 [11].

Even though there are empirical indications that encapsulation and information hiding are not used to the full extent in object-oriented programs there is a lack of data whether this have any impact on the quality of the software produced. There also is a lack of empirical data on representation exposure and its effect on software quality. To gain further knowledge in these areas we conducted a number of structural interviews among software engineers in the software industry.

3 The Interview Method

By using interviews we could gather real world data to get a deep insight into software engineers' view on encapsulation, information hiding and representation exposure in object-oriented programming.

The study was designed primarily as a pre-study to a forthcoming questionnaire survey on the same subject where quantitative data from software engineers was to be gathered.

The interview questions were designed to elicit information on the subject's object-oriented experience and

education, their way of working with encapsulation, information hiding and representation exposure and how they believe different techniques, methods and tools for controlling representation exposure would affect their way of working and the quality of the software they produce.

The interview template was tested by conducting three test interviews with software developers to receive feedback on the questions and on the interview technique. Note that the three test interviews are not included in the analysis.

All interviews were voice recorded by using dictation software on a portable computer with a built-in microphone and then transcripted for the analysis.

Eight experienced industrial software engineers were interviewed. They originated from software developing companies, computer consultants and computer departments at a financial institution, all having experience in producing object-oriented software. The subjects were given the questions beforehand via e-mail so they would be able to collect information in advance in order to give more accurate answers during the interviews. At the beginning of every interview the subjects were informed that their responses would be treated confidentially and they were also given instructions that they should try to answer the questions based on their own experience and their own opinions and not give answers based on rumors, hearsay or on other peoples' statements. The interview was kept informal to help the subjects answer as freely as possible. The interviews lasted between 25 minutes to approximately 1 hour and 10 minutes.

4 Analysis and Discussion

Prog Subject	Prog exp.	OO exp.	Prog meth	OO meth
A	2.5	2.5	U	U
B	2	2	U	U
C	3	3	U	U
D	6.5	6.5	U	U
E	20	5	I	I
F	8	5.5	Self	Self
G	7	7	U	U & I
H	15	9	U	U

Table 1. Subjects' professional experience in years and education in programming. U = University, I = Industry, Self = Self educated.

All subjects interviewed were experienced software engineers with between 2 and 20 years professional experience in software development and between 2 and 9 years in object-oriented software development [1]. All but two of the engineers interviewed had learned programming methodology and object-oriented development methodology from university courses. A detailed presentation of the subjects' experience and their education is shown in Table 1.

Subj.	C++	Delphi	Java	Other
A		√		
B		√		
C	√		√	
D	√		√	
E	√	√	√	Cobol, Visual Basic
F	√			Pascal, Visual Basic
G	√	√	√	Algol, C, Fortran
H	√			Object-1, Smalltalk

Table 2. Subjects' familiarity with different programming languages

The subjects were asked to list all programming languages they had more than one year of professional experience with. Six had experience in C++, 4 in Java and 4 in Delphi. Two had experience in all the three languages C++, Java and Delphi. Four had also experiences in other programming languages. The responses are summarized in Table 2.

4.1 The Use of Access Modifiers

All subjects mentioned that when declaring a member variable, their default choice of access modifiers is private or protected. Subject A changes the declaration of a variable from private to protected when it is suspected that the class holding the variable will later be subclassed and the subclass will be needing the variable.

Subjects C and E declared member variables as protected by default, subject D due to organization culture and subject E to facilitate inheritance beforehand. Subjects D, F and G use private as default unless there is an intention to create subclasses that will need the variable, and then it will be declared with protected modifier directly. Subjects C, E and F mentioned that they sometimes declare variables initially as public to do some testing in the initial development but always change the declaration when the testing is finished.

[1] The subjects were asked to calculate and give there experience in years of full-time professional work, *i.e.* 2 years half-time was equivalent to one year full-time.

Subjects A and B made the point that when a private declared variable needs to be accessed from an external object they may change the declaration to public instead of reworking the design when under time pressure. According to subject B "There is not always the time to do things exactly the way you want".

Additionally, subject H said that a class that is completely internal inside a component and not visible from the outside may have public variables. When a class works only as a container for data variables and there is no implementation in the class, subjects C and H may declare those variables as public.

Subjects C, G and H mentioned that they have never used public variables instead of private or protected for performance reasons. Subject F had used public variables for performance reasons one time when developing a graphical package.

4.2 Defects, Code Review and Quality Assurance

First, subjects A, B, C, D, E and H had never discovered bugs caused by the wrong use of access modifiers. Subjects F and G had discovered defects caused by this. Subject F said that when working with multithreading where two or more threads are accessing the same variable at the same time, it is very common to find defects. In all the cases where subject G had discovered defects the faulty code had been written by the same programmer who had misunderstood the concept of immutable objects.

Subjects A, B and C mentioned that the reason for not finding defects caused by the wrong use of access modifiers is their way of working with using private or protected as the default access modifier for member variables. Subject E said that "A reason for me not to discover such defects is due to the way we use object-oriented languages in general. We do not have very complex objects". Subject F also said that the reason is their organization's way of working, "If a function needed does not exist you often ask the person who created the class to create it".

However, subjects mentioned that they have come across, from their point of view, the wrong use of access modifiers but not in a way that caused defects. Subject D made the statement "I have often seen the wrong use of access modifier but not in a way that the functionality suffers". Subject G mentioned that it is very common to come across the wrong use of access modifiers, "Situations exist where people do not know which access modifiers they should use".

Second, subjects were asked how they ensure that they have, from their point of view, the appropriate level of encapsulation and information hiding in their classes. All subjects said that they strive to use the modifier that gives or preserves their desired level of information hiding when declaring a new variable. Subjects C and F said that they perform informal code walkthroughs themselves after the first initial development cycles to ensure that they have, in their opinion, an appropriate level of information hiding and encapsulation. Subject A repeated the earlier point made, that when it is assessed that it would take to long to modify the structure of the classes to get the desired level of information hiding, the informal rule of using the private modifier as the default modifier may be neglected. Subject C declared that "Sometimes it feels that some classes do not interact perfectly with each other and that is a good indication that something is wrong and must be fixed". Subject E made the point that "This is something that we probably do not put so much weight to" but also declared that "This is something that could be checked during a code review".

Third, none of the subjects had participated in formal code reviews where the use of access modifiers were reviewed in specific as a part of the review. Subjects A, B, C and F all said that this is since they have not yet participated in any formal code reviews at all. Subject D said that "Code reviews are often about looking at the code in general" and not about looking for details such as every use of access modifiers on variables. Subjects E and G had used checklists in code reviews but both stated that checking the use of access modifiers was not part of these checklists. Subjects E and G also agreed that the use of access modifiers can and should be reviewed in code reviews and walkthroughs whether or not it is included in the checklist.

4.3 Representation Exposure

During the interview the subjects were shown a Java example where it was possible from the outside of a container to obtain a reference to an object held by a private declared variable inside the container. The obtained reference could then be used to invoke methods that changed internal objects of the container. Questions were then asked about their experience regarding similar situations of representation exposure.

Regarding defects caused by representation exposures, all subjects recognized the situation in the shown example as a potential risk and almost all subjects spontaneously mentioned that they had seen these kinds of situations. However, only subject G had actually experienced defects caused by this. Subject A said "There are not so many defects due to this" and subject F stated that "It is fairly unusual".

Subjects A and D agreed that even if these kinds

of situations sometimes occur, the checks that are circumvented are usually performed at some other place in the program instead. Subjects A and F also said that these kinds of situations are rare since if the class is programmed correctly, methods that perform the operations needed should exist and the programmer would not circumvent any checks.

Subjects E and G also mentioned that this is a somewhat small issue since their programs are often rather primitive and do not have a very complex network of references. Also, while subjects D and H both agreed that this kind of programming is bad, subject B disagreed and said that this kind of programming is alright if you need to obtain an object inside a container.

Subjects A, B, D, E and G have never rewritten code similar to the example shown if it did not cause defects. Subject G recalled having rewritten similar code in cases where it caused defects. Moreover, subject G stated that similar code not containing defects has not been rewritten since "It has not been needed". Subject B recalled having inserted comments and updated documentation when similar code was found. Subject H has rewritten similar code even when it did not cause defects and remarked that "It can be very difficult sometimes... when the exported reference is used very much".

Regarding doing formal code reviews where representation exposure situations were explicitly reviewed, almost every subject mentioned that they had not participated in such code reviews. Subject D repeated the point that their code reviews are performed in a more general manner, not following detailed checklist looking for specific details in code. The subject also declared that "If I had seen something like this I would of course rework it". Subject E stated that "If our code review process works it would find such situations". Subject F said that "I do not use checklists but I check such things". Additionally, subject G has not participated in formal code reviews looking for representation exposure situations specifically, but added "You have to make sure the encapsulation works" and "You perform a general code review where also a review of the encapsulation is conducted". When probing for reasons for not having these kind of reviews, subject A said it was since "It has not been seen as a problem". Subjects B, F, H mentioned the lack of formal reviews as the major reason.

4.4 Methods, Tools and Techniques

The subjects' opinions differ regarding whether or not more information about how the classes they develop exposes their inner representation would be useful. Subjects D and E said that this kind of information would be of no use while B, C, F, G and H thought it could be somewhat useful.

Subjects mentioned that a tool giving information about representation exposure could be interesting. Subjects A and C said that they would like to have warnings from the compiler about whether or not objects' representations are exposed. Subject F does not want this kind of tool included in a compiler, but could instead imagine using it as an separate tool. Subjects A, C, D and H mentioned that it could be interesting to get information that representation exposure exists and subjects A and H also noted that it should be indicated when the exposed references were actually used for modification.

Subjects A, B and F said that if this functionality was built into the compiler it should be possible to turn off the feature if wanted. Subjects A, F and H agreed that it should be possible to instruct the tool not to point out representation exposures in cases where the programmer has inserted them deliberately for some reason. A similar point was made by subject C, that the tool should not give information about representation exposure for classes which are only containers for two or more variables and do not have any implementation. Additionally, subject F said that "Since programming is pretty flexible it should be possible to construct your own rule base that the tool could use" in order to give some classes special treatment when needed.

Regarding measures for information hiding, encapsulation and representation exposure, subjects B, C and H mentioned that a measure could be of some interest but they also agreed that this kind of information is of interest only as curiosity or to get a hint on the level of "quality" of the code in this respect. Subjects A, D, E, F and G were all doubtful about the usefulness of such measures. Subject A declared that "It is not so important to get a measure", and according to subject B "It could be a reason for every variable to be public". Subject D stated "No, I do not see anything, no usefulness at all actually" and subject E remarked "To have a measure would not add anything".

Fourth, almost all subjects agreed that a programming language that could control representation exposure would be useful. Only subject F disagree with the following motivation:

"If a programmer has constructed a class using this kind of feature and someone else needs to change something or fix a bug in the future you have to rewrite the whole class instead of just cheating a little bit by accessing the variable and give the variable the

correct value. I would like to keep this possibility."

Subject B said that it would be nice to have the possibility to allow some object to manipulate some data while others may not. Subject D said that it could be used for "Keeping stupid programmers from doing stupid things". Subject E mentioned that such a programming language would not add so much currently but "Could see the need under different circumstances". Subject G said that "Making an object immutable on the reference level would be interesting". Subject H noted that "This can be addressed via the design too, but it could of course be introduced into the language also. It could be of interest".

4.5 Impact on Software Quality and on the Way of Working

All subjects except subject E said that access to more information about their own classes' level of encapsulation, information hiding and representation exposure would affect their way of working in some way. Subject A, C and D agreed that they would change the code to increase the level of encapsulation and information hiding and reduce representation exposures if they got information about it. Subject B would change the code if there turned out to be very much representation exposures in the code. Subject E mentioned that under current circumstances it would not affect the way of working but said that under different circumstances it perhaps would. Subjects F and H both mentioned that they would become more careless in this respect since they would receive information about encapsulation, information hiding and representation exposure problems when they occur and could thus fix them afterwards.

Regarding whether the number of defects would decrease, almost all subjects mentioned that they would either not decrease or decrease very little. Subject A remarked that the reason was the size of their organization, "We can ask the others what they have done instead of just relying on the documentation". Subject G declared that "In my own programs it would probably not have any effect at all since I am aware of the problem, but I can imagine that in the programs I am working with it would definitely have a effect". Subject G remarked that "I do not think the number of defects would decrease. However, I think it could lead to better designs."

5 Conclusions and Future Work

Even though the conclusions drawn from a limited number of subjects interviewed are unlikely to be generalizable we share the view of Daly et al. that structured interviews are helpful [3]:

1. "For gaining insight into others' knowledge"
2. "For gaining insight into users' opinions about published problems and advocates claims", and
3. "As initial evidence for justifying further empirical investigation"

Analysis of the collected interview data revealed some interesting aspects. All subjects had also come across situations where the inner representation was exposed but only one had discovered bugs due to this and only one rewrite such code when it is discovered. Moreover, none of the subject had participated in formal code reviews where representation exposure had been reviewed as a part of the review.

Most of the subjects mentioned that they could make use of information obtained from tools, techniques or methods for representation exposure. A programming language for controlling representation exposure seemed to be the most useful tool while a measure was seen as less useful. However, it should be noted that the interviewer had worked on a programming language addressing the representation exposure problem and thus there is a great risk of interviewer bias regarding the responses on the usefulness of such a programming language.

In conclusion, the representation exposure problem does not seem to be a big problem among the software engineers. Nevertheless, even though access to more information regarding representation exposure is considered to give only a small decrease in the number of defects in software the software engineers believe that some tool or programming language for controlling representation exposure could be interesting.

We believe that further investigation is needed to clear out the issues in the practical use of encapsulation, information hiding and representation exposure. Our next step in this research is a questionnaire survey among software engineers to obtain quantitative data on a wider user group regarding these issues.

References

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–??, 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. Structured interviews on the object-oriented paradigm. Technical report, Department of Computer Sciencee, University of Strathclyde, Glasgow, 1995.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.