Structured Interviews on the Object-Oriented Paradigm

Abstract

The method of structured interviewing is an appropriate means of conducting a primary investigation when beginning a programme of research involving different empirical techniques because (i) interviewing experienced users from academia and industry should determine if academics are exploring issues that industrialists deem important, (ii) interviewing users from industry helps provide external validity, (iii) users of object-oriented systems have their own views on the benefits and drawbacks of the paradigm. These opinions could then be used as evidence to decide whether an empirically unsupported practice is justified and warrants further, more controlled investigation, and (iv) these views can identify issues that have not been previously considered.

Structured interviews, each based on a template of 23 questions, were carried out with 13 experienced object-oriented users to explore problems discussed in the object-oriented literature concerned with high level system understanding, inheritance, software maintenance, and other issues. Interviews lasted between 30 minutes and 2 hours.

Analysis, undertaken by transcripting and summarizing each interview and tabulating subject's answers to each question, discovered opinions similar to those expressed in the literature. These include the steepness of the learning curve, the difficulties of understanding created by inheritance, missing documentation, and inappropriate design, the tendency of systems to degrade as a consequence of software maintenance, and the view that C++ is inferior to purer OO languages.

1 Introduction

Performing empirical work is a way of building a cohesive body of knowledge within software engineering. Fenton et al [8], for example, recently argued the importance of experimentation to provide empirical evidence for proposed new software development and maintenance practices. Brooks et al [2], Jones [12], and Henry et al [11] make similar arguments. Evidence may be derived from any empirical technique, for example, through controlled laboratory experiments,

introspection, questionnaires, structured interviews, or
thinking-aloud protocol analysis. Although each technique provides
different empirical data (for example, the structured interview
promotes detailed expert knowledge elicitation whilst a laboratory
based experiment yields raw performance data concerned with time and
accuracy) it is argued, that these techniques can complement one
another. If an effect has been demonstrated by two or more different
empirical techniques it is more likely that the findings (a) are
reliable and (b) will be accepted by the software engineering
community. When the results are consistent, the empirical techniques
are said to have confirmatory power. Another approach views a series
of different empirical techniques as evolutionary: the important
issues discovered by the initial exploratory study are investigated
further by the next study, and so forth. Eventually, hypotheses are
tested by more focused empirical inquiry. It is the intention of our
programme of research to conduct an exploratory investigation through
structured interviews, followed by questionnaires to confirm the
findings across a wider user group, followed by laboratory experiments
to test the findings in a more controlled setting. Of course, in an
evolutionary programme of research, the results from each technique
may turn out to confirm one another.

Confirmatory power can also be achieved through external replication:
a researcher independently attempts to repeat the results of an
empirical study. If successful, the replication provides supporting
evidence, if unsuccessful it may have the ability to challenge the
original findings (see [3] and [2] for details).

Advocates of the object-oriented paradigm have made many claims about
the benefits of the paradigm, most of which remain empirically
unsubstantiated. Jones [12], for example, details a visible lack of
empirical data to support the assertions of substantial gains in
software productivity and quality, reduction in defect potential and
improving defect removal efficiency, and software design and component
reuse. Evidence is slowly beginning to filter through in certain
areas, but is by no means thoroughly conclusive. For example, Wilde
and Huitt [24] quote Mancl and Havanas [16] as providing evidence

\for extensive reuse of software objects and easier maintenance
through

better data encapsulation"

but their study lacks the reported detail to convince the strongest skeptics. In contrast, problems with object-oriented software are now beginning to be realised. For example, an empirical study by Dvorak [6] provides initial evidence of a degradation effect identified as conceptual entropy of class hierarchies, something which requires further investigation. So there is a need to provide stronger empirical evidence than currently exists. Structured interviewing is an appropriate method to initiate evidence gathering for several reasons. First, interviewing experienced users from academia and industry should determine if academics are exploring issues that industrialists deem important, i.e., do academics and industrialists have similar or differing opinions on object-oriented issues? Second, interviewing users from industry provides external validity in the form of real world data, something which in more controlled experimentation is difficult to provide. Third, object-oriented users have their own opinions on the advantages and disadvantages of the paradigm; these opinions could be used as evidence to decide whether an empirically unsupported practice is justified and warrants further, more controlled investigation. And fourth, these views can identify issues that have not been previously considered.

The structured interview requires a template from which the interview can be directed. The subject may give an answer which is deemed worthy of pursuing, and the interviewer can then ask supplementary questions (though care must be taken not to bias the the interview direction). Once satisfied, the interviewer returns to the next question on the template.

The remainder of this paper is partitioned as follows: Section 2 discusses some of the advantages and disadvantages of structured interviewing, Section 3 presents the detailed motivation for carrying out this work, with Section 4 detailing the actual interview method. Section 5 provides analysis and discussion of the data and Section 6 draws conclusions.

2 Interviewing as an empirical technique

Interviewing is an appropriate means for conducting a primary investigation which offers many positive aspects: First, there is the issue of external validity. Interviewing experienced subjects allows gathering of real world data which is of extreme importance for drawing real world conclusions. For example, if a number of experienced users hold the same opinion on a particular issue, the likelihood is an effect of some kind exists; further investigation is warranted. Second, as noted by Sinclair [21], interviewing is a flexible technique, for example, it does not have the rigidness of a questionnaire survey, and it allows data to be gathered relatively quickly and unhindered. Interesting points made by the subject can be explored in depth by efficient question probing. Furthermore, feedback from initial interviews can be used to refine the template to concentrate on issues that were not previously considered. Conversely, refinement can also remove issues which were initially considered to be important, but which on discussion appear less significant, i.e., the interview template can be re-oriented if necessary without major disruption to the study, as long as the detail is reported in the findings. Third, again noted by Sinclair [21], there is ability of the interviewer to motivate the subject to give more information about a topic as well as being able to direct and accelerate the information flow. Finally, there is the advantage of being able to re-interview the subject to further investigate points made previously. This allows clarification of opinions and, as such, is an advantage that other empirical techniques, e.g., questionnaires, do not readily allow.

As important, however, is to realise that there exists problems in collecting and analysing data from structured interviews. First, there is the danger of subjects saying what they think is appropriate to say. For example, if the subject being interviewed thinks that the listener is superior then, according to Bainbridge [1] there may be pressures to appear rational, knowledgeable and correct, or inversely the subject may become uncooperative or present a particular attitude to the topic. Moreover, there is also the problem that while an experienced subject may be opinionated and talk freely on the topic, they may not mention what they think should appear obvious. Second, most people think more quickly than they talk (especially if time pressures exist on the subject, e.g., the subject is being asked to speak their thoughts while performing a task of some description): this underlies the general point that verbally communicated data may only give a limited sample of the total knowledge of the subject under study, that is only a subset of their cognitive activity may be reported. Third, there might be social influences: if the subject thinks what they say can be held against them then they may report what they think the interviewer wishes to hear. And fourth, the interviewer must have a good working knowledge of the domain being questioned: without it, important aspects of the interview may be left as ambiguous, unresolved or, simply, unnoticed.

Unfortunately, there are no objective independent techniques for doing analyses on such data. Consequently, this may lead to problems for the analyser who may have to make assumptions when interpreting what the subject has mentioned, or use their own knowledge of the domain to make particular inferences. For these reasons, again according to Bainbridge [1], many academic psychologists feel strongly that verbal data

has limited importance: interviewer's bias can creep in. Also, there are high costs involved in terms of time and effort to fully analyse verbal data, for example, Neilson [18] cites Marchioni [17] as quoting the need to spend about 25 hours for each hour of subject observation on tasks such as transcribing video tapes and coding detailed protocol analysis. Neilson refutes this however and reckons in realistic development situations each hour of thinking-aloud observation only needs half an hour of combined analysis and report writing. In our experiences of transcripting and analysing structured interviews we found Marchioni's estimate reasonably accurate.

As a consequence, although conclusions drawn are unlikely to stand firm unless large numbers of users are interviewed, it is important to emphasize that this technique is helpful (i) for gaining insight into others' knowledge, (ii) for gaining insight into users' opinions about published problems and advocates claims, and (iii) as initial evidence for justifying further empirical investigation. Papers in the literature have used similar techniques, such as thinking-aloud protocols and knowledge acquisition through video recording, apparently with success (see for example [23] and [5]). However, as Lashley, quoted in Ericsson and Simon [7], states,

\introspection may make the preliminary survey, but it must be followed by the chain and transit of objective measurement."

We strongly agree with this statement and believe it can also be applied to the concept of interviewing; we therefore discuss future work from this primary research in the conclusions. For a more comprehensive discussion of the pros and cons of verbal data see [7], [9], and [20].

3 Motivation and related work

Many unsupported claims have been made for the object-oriented paradigm, for example, gains in software productivity and quality, reduction of development time, enables better prototyping and iterative development, ease of software design and component reuse, facilitates change, ease of understanding, ease of maintenance, and reduction in defect potential and improving defect removal efficiency. Henry et al [11] provide a list of references which they state have made claims as having qualitative appeal, but little supporting quantitative data.

In contrast, recent research has started to discover problems with the object-oriented approach particularly in the areas of software maintenance, inheritance, polymorphism and dynamic binding. These problems are now discussed.

Wilde and Huitt [24] present a comprehensive report on what they see as several failings of object-oriented programming: First, object-oriented languages complicate the tracing of dependencies by making use of dynamic binding of messages to specific methods. Second, inheritance, it is argued, can make understanding a single line of code a difficult task because it may require tracing a line of method invocations. Furthermore, inheritance can cause modification problems, e.g., if modifying a class X then a maintainer must be concerned with possible side effects in subclass Y. Third, the dispersion of functionality into different object classes may make the process of software reuse and maintenance inefficient because of difficulty in locating all the required code. Fourth, polymorphism can introduce subtle errors through inconsistent naming, e.g., if a system has several implementations of a method which have significantly different effects then a maintainer may be mislead when interpreting the code, and then introduce errors when changes are being made. Finally, it is argued that high level system understanding can be hampered because the traditional module calling hierarchy is replaced by a hierarchy of methods. A hierarchy of methods offers several difficulties because of the dynamic binding problem mentioned above; there may be no main or controlling system method which is usually a good starting point; and a hierarchy of methods detracts from the grouping of methods within objects. These points assume that only the source code is available, however, there is no mention of documentation which, if availble, is likely to ease these problems.

In a similar paper, Wilde et al [25] report that while object-orientation facilitates change it can also make programs harder to understand for maintainers. The issues addressed in [24] are reexamined, and the authors state

\... [the] different appearance and organization of object-oriented code may startle many programmers, and their traditional approaches to program understanding may break down."[25]

They conclude that it will take years to identify all the problems with maintenance of object-oriented software systems and, as such, any likely problem areas should be closely monitored.

Kung et al [13] reiterate some of the problems reported by Wilde et al [24], [25] arguing that the new characteristics of object-orientation (namely inheritance, polymorphism, and dynamic binding) introduce new problems in software testing and maintenance.

Lejter et al [14] also report on the problems of inheritance and dynamic binding with respect to software maintenance. Additionally, they discuss the problems of attempting

to answer questions such as \what is the definition of this function?", or \where is the declaration of this symbol?": without the use of good tools function and operator overloading, they argue, confuse such questions because names are no longer unique in object-oriented systems.

Ponder and Bush [19] discuss the issue of polymorphism considered harmful using the Smalltalk compiler source code listings as an example polymorphism causing understanding difficulties. While the authors feel that it is not necessary to know the implementations of polymorphic methods to understand their function, they report that several problems arise: first, inferring the generic meaning of a complex and obscurely named function from just a few instances is difficult. Second, the reader can inadvertently be encouraged to infer similarities from different generic operations which share the same method name. And, third, the nature of object-oriented systems dispersing operations across a large number of relatively small methods can make it difficult to understand their collective effect. Further, polymorphism, they argue, compounds this problem because if a method performs few operations, there is little basis for inferring the types of its variables. Ponder and Bush conclude that while polymorphism can be both a flexible and powerful programming tool, it presents opportunity for abuse which can ruin program understanding.

Finally, Dvorak [6] presents the property of conceptual entropy of class hierarchies: the deeper the hierarchy the more likely a subclass does not consistently extend or specialize its parent's abstraction. The concept was demonstrated to exist by an empirical study using seven experienced software engineers, each with on average one years objectoriented experience, as subjects. Dvorak concluded that if conceptual entropy was left unchecked it would eventually reach a stage where the class hierarchy would have to be restructured.

In conclusion, there are theoretical arguments for the advantages of the objectoriented paradigm, but various disadvantages have been reported as above. In our programme of research our first step has been to conduct structured interviews to explore these.

4 The interview method

The interview questions were designed to elicit information on three aspects of the users' object-oriented knowledge: information on their background, e.g., experience, language familiarity, their opinions on the perceived advantages and disadvantages of the paradigm, and their opinions on existing object-oriented technology, e.g., software

tools, languages such as C++, and class libraries. The interview was structured in such a way that factual questions (requiring just one or two sentence answers) were asked first, followed by questions requiring answers based on opinions from experience. This approach was taken to allow the subject time to settle and relax before asking the more thought provoking questions.

A draft interview template was written and prototyped by conducting an initial interview with an experienced industrial employee to receive reactions on the structured interview technique and the questions asked. Only a minor point was made and this was then taken into consideration, i.e., the subject felt not enough about software tools was asked, and he stressed that tools were an important part of object-oriented programming. The draft template was then revised to take account of this noted point; the other interviews were conducted with this refined template as their basis. Appendix A contains a copy of this template.

Twelve subjects, all of whom were individually approached, agreed to be interviewed using the refined template. Seven of these subjects were industrial employees and five were academics, four of whom had industrial experience of producing or working with object-oriented systems. The subjects were given no prior knowledge of the content of the interview except that it was based on the object-oriented paradigm, but were told that any replies given would be treated confidentially to reduce concern about how their answers might be used. The interview process was kept as informal as possible, e.g., the subjects were drinking coffee before and during the interview to help them relax and answer questions freely.

The interviews were carried out during the working day and lasted between 30 minutes to almost 2 hours. The entire interview, recorded using a hand held recorder, was transcripted onto paper to enable a thorough analysis. It should be noted that the initial interview was also included in this analysis as there was only a minor difference between the two templates used.
It was hoped that the interview template was comprehensive, but as a safeguard each subject was asked the final question

Has there been anything that you have not been asked that you feel is important and should be addressed?

The majority of subjects answered no to this, or mentioned something specific about their own work which they would have enjoyed talking about. In all, the subjects appeared to think the interview was a \pretty full questionnaire".

5 Analysis and discussion

Analysis was undertaken by summarizing each interview transcript, and tabulating answers (paraphrased) for each question to compare and contrast them (see Appendix B). This method of analysis has the advantage of allowing the data to be easily visualized, bearing in mind that 13 subjects were interviewed.

Table 1 summarizes the object-oriented experience of the subjects, their position (either an academic or industrialist), and their knowledge of object-oriented languages. The level of object-oriented experience varied from 6 months to 10 years, but note that the least experienced used it every working day. So we feel justified in describing all these subjects as experienced object-oriented users (only I, J, and L were not daily users).

5.1 Learning curve, documentation, time pressures and quick

fixes

First, several subjects mentioned the learning curve as an influencing feature when changing from the structured design paradigm to that of object-orientation. Subjects A, I, J, and L made the point that the learning curve was steep. Additionally, subject J commented \you really have to think differently ... I'm still thinking in structured C in some ways" (repeating the point made in [25]) and subject L reckoned to fully make the transition can take as much as two years. Regarding the transition from structured to

object-oriented programming subjects B, K, L, and M said the use of hybrid languages, e.g., C++, has a detrimental influence: while C++ allows the programmer to write C code providing several advantages, e.g., use of existing C libraries, familiarity with C, and not all problems are suited to an object-oriented solution, according to subject K \it's a language which doesn't really encourage to think in terms of objects" (subjects B, L, and M agreed). Consequently, according to subject B \someone ... might just regress into doing something in a C fashion when they should be doing it in an object-oriented fashion." Subject M said \being forced to be object-oriented I'm sure actually helped me." Lozinski [15] details many of the limitations of C++ and provides some excellent accompanying discussion by comparing it to Objective-C. (Tables 14, 25, and 26 provide individual comments.)

Second, ten subjects, including all industrial subjects, mentioned that they had had trouble with the availability of design documentation: either it did not exist or it was inadequate. Additionally, subject K mentioned when time constraints become a factor documentation is the first thing that suffers (subjects A and D made similar statements). Moreover, subject L mentioned that documentation for object-oriented systems is a more important aid to understanding than it is for other systems. Subject G qualified, \design documentation is important otherwise you are back to where you were before [with structured designed systems]". The subject made the point that object-oriented systems are not easier to understand than other systems unless the design documentation fully captures class relationships, class member functions, member variables, and so on. Subject M supported this claiming, \documentation is essential." However, according to subject H \... on the last project I worked on there wasn't any decent documentation anyway, ... you might find that, that's by far the most common case that there's no good up to date design documentation." (Tables 21 and 28 provide individual comments.)

Third, eleven subjects thought it was possible to perform quick fixes on objectoriented code, e.g., \yes, particularly in C++", \it's too easy in C++", \in C++ it's a lot easier", \it's all too easy, it's really easy, and the temptation is always there". Subject L qualified

this as \one of the dangers in these hybrid languages", while subject G stated \yes, but that's bad maintenance rather than bad design". Subjects A, C, D, F, and K mentioned a quick fix can be the result of time pressures. In a previous paper [4], the authors have discovered pragmatic maintenance (performing the minimal amount possible to complete the required task) under controlled laboratory conditions. (Tables 27 and 28 provide individual comments.)
The problems mentioned above appear indicative of Kung et al's [13] experience, who state

\Our experience indicates that it is extremely time consuming and tedious to test and maintain an OO software system. This becomes even more acute when documentation is either missing or inadequate."

5.2 Inheritance and high level understanding

First, subjects A, C, F, G, I, and L made the point that if inheritance is designed properly (the hierarchy is structured using the appropriate abstractions) then it will not cause understanding difficulties. Additionally, 5 of these subjects agreed that if designed properly inheritance should aid understanding. On the other hand, subjects D, E, H, J, K, and M mentioned that inheritance can make understanding difficult: subject D said that it can be difficult to trace the flow of control, while subjects J, K, and M mentioned that tracing a line of method invocations to determine which method is performing the work is difficult (repeating two of the points in [24]). Subjects H and K agreed it can be difficult to understand what functionality a line of code is performing until it is realised it calls an inherited member function. Subjects H and K also agreed inheritance can cause confusion when interacting with inherited member variables - where have they been inherited from? (repeating two of the points in [14]). However, subject C's statement, made in the context of design, appears to best fit the general opinion

\I think on the whole, it comes back slightly to how well written the code is ... I think once you get into the way of thinking about inheritance it does not make it more difficult to understand and in fact often makes it a lot simpler, and if your code is badly written then it's a nightmare."

(Table 7 provides individual comments.)

Second, regarding the effect of spreading method functionality over the inheritance hierarchy on system understanding almost every subject

commented that if the inheritance hierarchy is designed properly then
it would not be detrimental to understanding. Additionally, subject A
said that it was a natural way of writing and understanding code,
subject C declared that \it's likely to aid understanding", and
subject H stated \it simplifies things; it simplifies your design".
(Table 8 provides individual comments.) Third, an issue discussed was
the depth of inheritance, and how it affects system understanding.
Subjects were asked to define how many levels of inheritance it took
before their understanding began to become constrained. These
definitions are summarised in Table 2. Subject C said that 10 levels
of inheritance (including multiple inheritance) would be a deep and
complex hierarchy. The subject added that

\you want to avoid using a depth of nesting that you don't need and
it's therefore inappropriate."

The subject made the point that the hierarchy must be properly
designed. Subject H also noted that 10 levels of inheritance is a deep
hierarchy. In contrast, subject I stated that depth was not an issue
he ever considered and, therefore, the question was not relevant. The
remaining subjects mainly noted 3-4 levels of inheritance as deep.
Comparison of Tables 1 and 2 shows no significant relationship between
experience or the language used1 and the depth of inheritance.

Fourth, subjects B, C, G, H, I, and K mentioned that multiple
inheritance is more complex than single inheritance. Additionally,
subject L was concerned that multiple inheritance allowed for a sloppy
approach (inappropriate inheritance). He stated his reluctance to use
multiple inheritance by mentioning anything designed using multiple
inheritance can be designed using single inheritance. Subjects B and H
agreed mentioning that multiple inheritance makes your design more
complex. On the other hand, subjects C, E, and K stated multiple
inheritance had benefits. Subject C reckoned it is easier to reuse
with multiple inheritance and it can also make maintenance changes
easier. Subject E mentioned it maps the reality of systems being
modelled, and subject K agreed reporting that if you have a real case
for multiple inheritance then it is advantageous to use it. Subjects
B, G, and L disagreed however. Both B and L stated multiple
inheritance is more tightly coupled than single, and therefore reuse
it harder, 1Although most subjects talked in the context of C++ for
the majority of the interview.

Advantages Disadvantages
Allows design at the highest level of abstraction Understanding is difficult if not well designed Provides modularity Overuse or inappropriate use of inheritance leads Prevents code redundancy to more complex code that's harder to follow Aid to understanding if designed well Multiple inheritance can complicate the design Information hiding through encapsulation Tracing flow of control and dependencies can be Quick prototyping difficult
Single inheritance encourages good design New concept to learn
Multiple inheritance for complex designs Deep hierarchies can cause understanding difficulties

Table 3: Summary of inheritance advantages and disadvantages

while subject G stated \single inheritance is the way to do it."
(Table 12 provides individual comments.)
Finally, Table 3 summarizes the discussed advantages and disadvantages of inheritance. (Tables 10 and 11 provide individual comments.)

5.3 Maintenance of object-oriented programs

First, regarding the effect of making changes to object-oriented programs in comparison to equivalent structured programs, subjects A, B, C, D, E, F, G, H, K, and L stressed that if well designed changes are made to a well designed object-oriented system, the effects of these changes are likely to be more localized with object-oriented code than with structured code. In contrast, if the changes are not well designed changes then they are just as likely to propagate through an object-oriented system as much as any other. Although subjects D, H, J, and K agreed that changes are more localized in an objectoriented system, these 4 subjects also noted that changes can also have a bigger global effect because any changes made will affect subclasses which inherit them (repeating the point in [13] and [24]).
On the other hand, subjects D, H, and K noted that this concept has the benefit of being able to \fix it for one, fix it for all": if class X has a corrective maintenance change made then any subclass Y has that change made automatically through inheritance (a benefit not mentioned in [13] or [25]). Subjects B, D, G, J, and L raised the issue of tracing of method dependencies through the class hierarchy as \problematic" (repeating the points in [24] and [14]). According to subject K \just seeing where a piece of code does that work ... from the point of view of understandability can be quite tricky." The subjects agreed this is compounded when the hierarchy is deep. (Tables 5, 7 and 9 provide individual comments.)

Second, regarding object-oriented programs causing problems for software maintainers. Subject A related it to the learning curve: understanding the way an object-oriented program works is not easy to begin with and takes time to acquire the correct method of thinking, \learning by osmosis" (repeating the point made in [25]). Subject J repeated that missing documentation could cause maintainers difficulty. Subjects B and K mentioned the one class per file style promoted by Stroustrup [22] as undesirable: it can make tracing of dependencies extremely difficult without good tool support (repeating the point in [14]). The general opinion was, however, that there is more structure to a well designed object-oriented system and, as such, it is easier to maintain than a non object-oriented system, although subject L felt that few object-oriented systems have had sufficient maintenance to be able to say. (Tables 13, 14, and 19 provide individual comments.)

Third, subjects were asked if they thought that continual maintenance would eventually lead to unmaintainability. The majority of subjects commented that this would be the case although in comparison to an equivalent structured program there is a lesser tendency for this to happen, i.e., it would still happen but over a longer of period time. Subject A stated this would happen \without a doubt": the rate of entropy increases over time and everything tends to disorder. Subject D also discussed the rate of entropy and conceded it is reduced for object-oriented systems. Subject E agreed, but only on the condition that programmers \maintain the object mindset for making changes" to the system. Subject J made a similar statement. Subject H concluded that as with anything, if ad hoc changes are made then unmaintainability will ensue. (Table 15 provides individual comments.)

Fourth, subjects mentioned that tools are helpful for understanding and maintaining object-oriented programs. Differences of opinion did arise about whether they are a necessity or not: subjects E, F, H, J, K, L, and M all declaring they are, the remainder declaring they are not. All the subjects, however, did agree that tools do alleviate some of the understanding problems that can occur when performing object-oriented software maintenance. Subject I reckoned that more object-oriented tools are needed. (Table 6 provides individual comments.)

Finally, subjects A, B, C, D, G, K, and M mentioned that it would be difficult for a maintainer to tell if any changes made had degraded the system code quality (repeating the point made in [13]). Code reviews and walk-throughs performed by a panel of programmers are recommended practice by these subjects as a reliable safety net for catching any introduced bugs. (Table 20 provides individual comments.)

5.4 Other issues

First, regarding small methods as good object-oriented programming practice, subject A concluded that methods \can be one line ... but as much as 50 lines for `real' methods" depending on their function. Subject F stated, \I try to keep them small." However, subjects B, C, D, and K all mentioned methods of size in excess of 100 lines of code (subject C had worked with a method of 2500 lines) although this was not regarded as good programming practice. In general, the subjects felt that small methods are good programming practice. (Table 4 provides individual comments.)

Second, subjects were asked if they had ideas on good and bad object-oriented programming practices. Once more the importance of design was stressed. Subjects A, C, F, I, K, and M mentioned that good design usually means good code; conversely bad design produces bad code. According to subject C \it is almost less the coding style than the design style that is important". Additionally, subjects C and D agreed that not too deep an inheritance tree was good style (although from Table 2 it can be seen their definitions of deep differ). Subject H added \there seems to be few points of reference for object-oriented code quality". The rules and recommendations produced by Henricson and Nyquist [10] is one programming standard for object-oriented programming which may help to reduce any inconsistencies that exist. (Table 19 provides individual comments.)

Third, every subject interviewed thought that polymorphism is a useful concept in object-oriented programming and they agreed it can produce generic code. However, subjects B, C, E, G, I, and J mentioned that polymorphism, especially if semantic consistency is not maintained, can cause confusion and make it difficult to understand how the system works. In particular, subject B stated,

\Yes, it's a great benefit in some cases ... In some cases, it makes things really difficult to understand ... Used properly and used reasonably sparingly, it can be really advantageous."

(This reiterates the conclusions drawn in [19].) (Tables 23 and 24 provide individual comments.)

Similarly, regarding dynamic binding, while every subject thought that it a clearly defined benefit (subject A remarked \without it there is no object-oriented programming") subjects C, E, L, and M felt that there can be understanding difficulties if the programmer was \careless" when using it. Subjects C and L mentioned that one of the big problems is the calling of non-existent methods. However, if managed carefully then according to subject D \it simplifies the code ... It's a more intuitive way of working."

(these points do not directly support the points made in [24] or [14]). (Table 22 provides individual comments.)

Fourth, regarding software reuse only subject E had not made any use of class libraries. The remaining subjects all made use of commercially produced libraries, however, subjects A, D, I, K and L were the only frequent users of local class libraries. No generalization can be made, but it does support asking the question is object-orientation meeting its reputation for ease of software reuse? (Tables 16, 17, and 18 provide individual comments)

Finally, subjects were asked to complete the interview with a conclusion on the utility of the object-oriented paradigm (is it beneficial?). Every subject answered yes, for example \no doubt about it", \the only way", \the right thing to do", \where it's appropriate it's the way forward". Again, no generalization can be made, but it does deserve recognition as of some importance.

6 Conclusions and future work

Interviewing users on the advantages and disadvantages of aspects of software engineering can be rewarding: it can help to identify (i) problems not yet considered, (ii) conditions under which existing problems are compounded or alleviated or not applicable, and (iii) claims which appear worthy of further empirical inquiry and those that do not. Furthermore, the structured interview can be an effective method of exploratory or primary investigation when undertaking a multi-method programme of empirical research. We warn, however, against generalising the results of this study due to its limited sample size.

It is believed that the weaknesses of the structured interview technique were not a major influence on this study because (a) anonymity of subjects and the interview setting encouraged them to speak freely, (b) the interviewer's experience in the domain meant

further investigation of important points, and (c) clarification of
elicitated knowledge could be made after the interview transcript was
available. The analysis was performed by transcripting and summarizing
each interview and tabulating subjects' answers to each question.
Interview transcripts were read several times and it is unlikely that
any important points were overlooked.

Analysis of the collected interview data uncovered several aspects of
marked interest. First, the learning curve was mentioned as being
steep when switching to the objectoriented paradigm from another,
different paradigm; the figure of almost two years was mentioned to
make the transition. Second, depth of inheritance was discussed:
excessive

inheritance depth or inappropriate use of inheritance can cause
understanding difficulties and is therefore something to be avoided
(any initial time saved will be outweighed by the difficulties the
inappropriate inheritance causes). In contrast, inheritance, when
designed and implemented appropriately, is more likely to aid
understanding than compound it. Understanding difficulties are created
by dependencies caused by inheritance, however. Tracing a line of
dependencies through inheritance hierarchies to find where the work is
being performed can be a time-consuming task. Third, object-oriented
systems are as prone, if not more so, to the problems of missing
documentation. Several subjects mentioned that time constraints can
cause documentation to suffer. The drawbacks of missing documentation
should be considered carefully from a maintenance perspective. If the
documentation captures the correct information it could reduce future
time pressures: subsequent changes may be properly designed, reducing
system degradation; if not then the reverse might be true. Fourth, the
design of an object-oriented system appears to be extremely important:
subjects continually mentioned design when discussing the problems of
inheritance, software maintenance, polymorphism, and dynamic binding,
stating that if the design was appropriate then understanding was not
constrained. In contrast, however, if not well designed then the
system could be a \nightmare" to understand. Fifth, as a consequence
of software maintenance, object-oriented systems are still prone to
degradation. The consensus, however, was that object-oriented software
would resist software degradation through maintenance better than
other software systems, but only if the object mindset is kept when
performing maintenance: `quick fixes' are likely to increase software
degradation (or the rate of entropy, as referred to by 2 subjects) to
that of any other software. Trainees, therefore, are not the best
people to perform maintenance. Finally, subjects indicated a lack of

enthusiasm towards C++ and viewed it inferior in many aspects to `purer' object-oriented languages.

Finally, as discussed in the introduction, a programme of research using different empirical techniques is more likely to allow conclusions which hold weight if an effect has been consistently demonstrated. Conducting the structured interviews was the first technique in our planned multi-method approach. The next stage in this research was designing a questionnaire from the structured interview findings to gather quantitative data on a wider user group. Analysis of this survey data is currently being undertaken.

References

[1] L Bainbridge. Verbal protocol analysis. In J Wilson and E Corlett, editors, Evaluation of Human Work, A practical ergonomics methodology, pages 161{179. Taylor and Francis, 1990.

[2] A Brooks, J Daly, J Miller, M Roper, and M Wood. Replication of experimental results in software engineering. submitted to IEEE Transactions on Software Engineering, 1995.

[3] H Collins. CHANGING ORDER Replication and Induction in Scientific Practice. SAGE Publications, 1985.

[4] J Daly, A Brooks, J Miller, M Roper, and M Wood. Verification of results in software maintenance through external replication. In Proceedings of the IEEE International Conference on Software Maintenance, pages 50{57, September 1994.

[5] S Denning, D Hoiem, M Simpson, and K Sullivan. The value of thinking aloud protocols in industry: A case study at Microsoft Corporation. In Proceedings of the Human Factors Society 34th Annual Meeting, pages 1285{1289, 1990.

[6] J Dvorak. Conceptual entropy and its effect on class hierarchies. IEEE Computer, 27(6):59{63, June 1994.

[7] A Ericsson and H Simon. Protocol Analysis. The MIT Press, 1st edition, 1984.

[8] N Fenton, S Pfleeger, and R Glass. Science and substance: A challenge to software engineers. IEEE Software, 11(4):86{95, July

1994.

[9] J Goguen and C Linde. Techniques for requirements elicitation. In Proceedings of the IEEE International Symposium on Requirements Engineering, pages 152{164, 1993.

[10] M Henricson and E Nyquist. Programming in C++ Rules and Recommendations. Ellemtel Telecommunications Systems Laboratories, 1992.

[11] S Henry, M Humphrey, and J Lewis. Evaluation of the maintainability of object oriented software. In IEEE Region 10 Conference on Computer and Communication Systems, pages 404{409, September 1990.

[12] C Jones. Gaps in the object-oriented paradigm. IEEE Computer, 27(6):90{91, June 1994.

[13] D Kung, J Gao, P Hsia, F Wen, Y Toyoshima, and C Chen. Change impact identification in object-oriented software maintenance. In Proceedings of the IEEE International Conference on Software Maintenance, pages 202{211, September 1994.

[14] M Lejter, S Meyers, and S Reiss. Support for maintaining object-oriented programs. IEEE Transactions on Software Engineering, SE-18(12):1045{1052, December 1992.

[15] C Lozinski. Why I need Objective-C. Journal of Object-Oriented Programming, pages 21{28, September 1991.

[16] D Mancl and W Havanas. A study of the impact of C++ on software maintenance. In Proceedings of the IEEE Conference on Software Maintenance, pages 63{69, 1990.

[17] G Marchionini. Evaluating hypermedia-based learning. In D Jonassen and H Mandel, editors, Designing hypertext/hypermedia for learning, pages 355{373. Springer-Verlag, 1990.

[18] J Neilson. Evaluating the thinking-aloud technique for use by computer scientists. In R Hartson, editor, Advances in Human Computer Interaction, volume 3, pages 69{82. Ablex Publishing Corporation, 1992.

[19] C Ponder and B Bush. Polymorphism considered harmful. ACM

SIGSOFT, Software Engineering Notes, 19(2):35{37, April 1994.

[20] N Shadbolt and M Burton. Knowledge elicitation. In J Wilson and E Corlett, editors, Evaluation of Human Work, A practical ergonomics methodology, pages 321{345. Taylor and Francis, 1990.

[21] M Sinclair. Subjective assessment. In J Wilson and E Corlett, editors, Evaluation of Human Work: A practical ergonomics methodology, pages 58{88. Taylor and Francis, 1990.

[22] B Stroustrup. The C++ Programming Language. Addison-Wesley, 2nd edition, 1991.

[23] D Walz, J Elam, and B Curtis. Inside a software design team: Knowledge acquisition, sharing and integration. Communications of the ACM, 36(10):63{76, 1993.

[24] N Wilde and R Huitt. Maintenance support for object-oriented programs. IEEE Transactions on Software Engineering, SE-18(12):1038{1044, December 1992.

[25] N Wilde, P Mathews, and R Huitt. Maintaining object-oriented software. IEEE Software, 10(1):75{80, 1993.