A Survey of Experiences amongst Object-Oriented Practitioners

J Daly, J Miller, A Brooks, M Roper, and M Wood
Department of Computer Science, University of Strathclyde, U.K.

Abstract

The object-oriented paradigm is becoming increasingly popular as a result of expert opinion and anecdotal evidence and not on the basis of sound empirical data. This questionnaire survey was undertaken as part of a programme of research to validate unsupported claims about the paradigm. The questionnaire follows structured interviews of experienced object-oriented developers [7] with the intention to confirm the findings on a wider practitioner group. It was posted to relevant electronic newsgroups and to members of an object-oriented (postal) mailing list. The survey received 167 responses to the electronic questionnaire and 119 responses (30% response rate) to the postal version.

Results show that respondents are of the view that:
(i) The object-oriented paradigm has advantages over other paradigms in terms of ease of analysis and design, programmer productivity, software reuse, and ease of maintenance.
(ii) Inheritance can introduce difficulties when trying to understand object-oriented software.
(iii) Missing design documentation and poor or inappropriate design are prevalent problems.
(iv) Maintenance causes degradation of object-oriented software, but less frequently than conventional software.
(v) C++ has many deficiencies in comparison to other purer object-oriented languages.

KEYWORDS: empirical, object-oriented, questionnaire survey

1 Introduction

Object-oriented has become the buzz-word of the 90's in the software engineering industry as a result of expert opinion and anecdotal evidence from individual developers. This has lead to speculation about the advantages that the object-oriented paradigm offers. Relying only on expert opinion and anecdotal evidence to support such claims, however, can induce biased and misleading conclusions. This is something which Henry et al. [15] have demonstrated. In their study, subjects expressed the opinion that maintenance tasks were more difficult to complete when using object-oriented software in comparison to using procedural software. This is despite the fact that the data collected supported the premise that object-oriented techniques resulted in fewer maintenance changes and reduced maintenance effort. To avoid such dangers there is a need for a cohesive body of knowledge built from the solid foundations of empirical study. Fenton et al. [11] have recently argued the importance of experimentation to provide well founded evidence for proposed new software development and maintenance practices. Brooks et al. [1], Glass [13], and Lewis et al. [20] make similar arguments. Evidence must be derived from a variety of empirical techniques, the data collected being used to substantiate findings, identify discrepancies, and act as a platform for further investigation.

Unfortunately, little evidence exists to support many of the claims made about the object-oriented paradigm. Jones [17], 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 reuse of software components. Henry et al. [15] provide a list of references which they state have made claims having qualitative appeal, but which have little supporting quantitative data. Evidence is slowly beginning to filter through in certain areas, but is by no means conclusive, e.g., Mancl and Havanas [22] provide some evidence of reuse of software objects and easier maintenance. Lewis et al. [20] have also provided some evidence through experimentation using students as subjects which suggests that the object-oriented paradigm has a particular affinity to the reuse process. In addition, the experimental results showed the object-oriented paradigm substantially improved productivity over the procedural paradigm. On the other hand, problems with object-oriented software are beginning to be realised, e.g., Dvorak [9] reports initial evidence of a class hierarchy degradation effect, identified as conceptual entropy; Wilde and Huitt [26] report understanding and maintenance difficulties that inheritance, polymorphism, and dynamic binding can cause.
So there is a need to provide stronger empirical evidence than currently exists. As a consequence, a programme of research, beginning with an exploratory investigation interviewing experienced object-oriented developers [7], was embarked upon. The questionnaire survey reported here followed these interviews in an attempt to confirm their findings on: (i) the perceived advantages of the paradigm, (ii) inheritance, (iii) the difficulties object-oriented code can cause, (iv) software maintenance and its consequences, (v) use of in-house (local) class libraries, and (vi) the C++ programming language. Using questionnaires allows quantitative data to be gathered relatively cheaply and quickly from many different practitioners. The collected data can also help identify hypotheses to be tested by more focused empirical enquiry. It is the intention of this programme of research to test important findings from the interviews and questionnaires with controlled laboratory experimentation.

This paper presents the findings of an object-oriented questionnaire posted to appropriate world distributed electronic newsgroups and to members of an object-oriented (U.K. postal) mailing list.

2 Empirical framework

Evidence may be derived from any empirical technique, e.g., through controlled laboratory experiments, introspection, questionnaires, structured interviews, or thinking-aloud protocol analysis. It is argued that although each technique produces different empirical data, data collected from one empirical technique can complement data collected from another different empirical technique. This approach is termed 'multi-method' [6]. If an effect is 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 agree, 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 an initial exploratory study are investigated further by the next study, and so forth, with each study in turn focusing more closely on the phenomenon under investigation. Of course, in an evolutionary programme of research, the results from each technique may turn out to confirm one another. As indicated, the plan of our evolutionary programme of research has been to conduct an exploratory investigation through structured interviews, followed by questionnaires with the intention to confirm the findings across a wider practitioner group, followed by laboratory experiments to test the findings in a more controlled setting.

3 Questionnaire construction

The questionnaire was divided into two sections. Section 1 collected details on the respondents position at work, experience with the object-oriented paradigm, their familiarity with object-oriented languages, and information on respondents experience with inheritance, maintenance, and class libraries. Section 2 asked questions to which the answers were more based on opinions derived from “experience, reading, or conferring with colleagues.” All questionnaires were distributed and returned in the first half of 1994.

3.1 Derivation of the questions

As stated, the questionnaire was the second technique used in our programme of research. The aim was to explore practitioners’ attitudes to some of the findings of the structured interviews [7]. Analysis of these attitudes can be used to form hypotheses to be tested by more specific forms of experimentation, for example, a questionnaire devoted to object-oriented software maintenance, or a controlled laboratory experiment to test a smaller, but more focused hypothesis.

Issues discovered in our interview study lead to the following survey objectives

1. Explore attitudes to the perceived advantages of ease of analysis and design, programmer productivity, software reuse, and ease of maintenance.
2. Explore attitudes towards inheritance (including depth of inheritance and multiple inheritance) and measure how often reported difficulties in the literature occur in reality.
3. Catalogue the difficulties understanding object-oriented code.
4. Explore attitudes to object-oriented software maintenance and its consequences.
5. Discover if the promise of software reuse is being met only through commercial class libraries or if in-house (local) class libraries are being used.
6. Explore practitioners attitudes towards the C++ programming language.

Nineteen questions (several with 2 or more parts) were derived from these objectives (this includes 3 initial questions on experience, job classification, and
Table 1: The frequency of respondents' positions and the percentage of the survey each category accounts for

<table>
<thead>
<tr>
<th>Position</th>
<th>Frequency</th>
<th>% of Survey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>37</td>
<td>13.5</td>
</tr>
<tr>
<td>Academic</td>
<td>34</td>
<td>12.4</td>
</tr>
<tr>
<td>Software engineer</td>
<td>127</td>
<td>46.2</td>
</tr>
<tr>
<td>Project manager</td>
<td>35</td>
<td>12.7</td>
</tr>
<tr>
<td>Others</td>
<td>42</td>
<td>15.3</td>
</tr>
</tbody>
</table>

Table 2: Respondents' experience with the object-oriented paradigm

<table>
<thead>
<tr>
<th>Time</th>
<th>Frequency</th>
<th>% of Survey</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1 year</td>
<td>25</td>
<td>9.2</td>
</tr>
<tr>
<td>1 - 2 years</td>
<td>69</td>
<td>25.3</td>
</tr>
<tr>
<td>3 - 4 years</td>
<td>85</td>
<td>31.1</td>
</tr>
<tr>
<td>&gt; 4 years</td>
<td>94</td>
<td>34.4</td>
</tr>
</tbody>
</table>

An initial pilot study was conducted to uncover any unnoticed assumptions in the questions. Feedback from respondents was used to remove identified faults and to produce the final version of the questionnaire. Note that these responses were not included in the analysis. More detail on the questionnaire construction, administration, and distribution can be found in [5].

4 Responses received

A total of 286 responses were received: the electronic survey produced 167 respondents to the questionnaire, and the postal survey produced 119 returns from 400 posted forms (30% response rate), although 11 of which were incomplete to such an extent that they were excluded from the analysis. Edwards [10] remarks that a response rate of 20-30% is considered to be adequate. In their well referenced software maintenance survey, Lientz and Swanson [21] 'only' produced a 24.6% response rate and that included sending a reminder.

Before discussing the analysis of the 275 completed questionnaires we present the information on our respondents' attributes. Table 1 presents a break down of the different positions held by respondents, the largest proportion being software engineers (46.2%). The 'Others' category consists of class librarians, consultants, research assistants, system managers, and technical directors. Table 2 presents a break down of the respondents' experience of the object-oriented paradigm, the largest proportion (34.4%) having greater than 4 years experience.

Further, approximately 90% of the respondents claimed to be using object-oriented technology more than twice every working week, with only 2% using it less than once a week. Finally, a break down of the languages respondents are familiar with is presented in Table 3. The table displays the number of respondents familiar with C++, Objective-C, and so on, and the percentage this accounts for of the total respondents. The 'Others' row contained the languages C-Flavors, Dylan, object-oriented COBOL, object-oriented Pascal, Sather, and Simula. As expected, C++ is the most familiar language although the percentage of respondents familiar with it might have been higher. Further examination of the dataset found that software engineers (67%) were more familiar with C++ than were academics (50%) or project managers (40%).

Table 3: Respondents' familiarity with different object-oriented languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Frequency</th>
<th>% of Survey</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>159</td>
<td>57.8</td>
</tr>
<tr>
<td>Objective-C</td>
<td>98</td>
<td>36.6</td>
</tr>
<tr>
<td>Eiffel</td>
<td>17</td>
<td>6.2</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>93</td>
<td>33.8</td>
</tr>
<tr>
<td>CLOS</td>
<td>15</td>
<td>5.5</td>
</tr>
<tr>
<td>Others</td>
<td>49</td>
<td>17.8</td>
</tr>
</tbody>
</table>

5 Analysis and discussion

Analysis was performed by tabulating the responses for each questionnaire into the format of the statistics package SPSS which was then used to calculate frequencies and percentages. Statistical tests were applied where appropriate.
The object-oriented paradigm more beneficial in terms of ...

Figure 1: Is the object-oriented paradigm more beneficial than other paradigms in terms of A: ease of analysis and design, B: programmer productivity, C: software reuse, and D: ease of maintenance?

5.1 The object-oriented paradigm in comparison to other paradigms

Many unsupported claims about the benefits of the object-oriented paradigm over other paradigms have been made, particularly with respect to ease of analysis and design, programmer productivity, software reuse, and ease of maintenance. Figure 1 displays the distribution of responses to a closed question on these issues. The results show a huge positive response in favour of the object-oriented paradigm in each of these categories. A subset of respondents, however, qualified their response. For example, the problem: not all problems are best suited to an object-oriented solution, the design: if the design is poor then the resulting object-oriented software system is likely to be more difficult to maintain than a poorly designed conventional equivalent, the quality of abstraction: if hidden assumptions are made during the design and implementation phases, the resulting software will be harder to reuse, and all things being equal between the paradigms: e.g., a skilled non object-oriented programmer will be more productive in their paradigm than a non skilled object-oriented programmer. While these caveats are reasonable, the fact remains the large majority of respondents view the object-oriented paradigm as offering benefits in these four categories. (Remember that the sample are experienced object-oriented practitioners who use the technology regularly.)

5.2 Inheritance

Three questions were devoted to the topic of inheritance: First, depth of inheritance was considered. A closed question asked at which depth does inheritance begin to cause problems (in [7] many interviewed subjects considered depth of inheritance an issue). The data, illustrated in Figure 2, shows, that of those respondents that felt depth can cause problems (approximately 55%), the largest proportion marked 4-6 levels of inheritance as the region they begin to have difficulties. A Chi-square test performed on the respondents who reported having problems with depth of inheritance against those who did not, provided a statistical significant result \( p < 0.05 \) (one-tailed, \( df = 3, X^2 = 6.79 \)) for an association between experience and having a problem with depth of inheritance. That is, the more experienced the developer the less likely they are to have a problem with depth of inheritance. Yet the frequency of experienced respondents who reported a problem with depth is still high: 43 respondents (50.6%) with 3 - 4 years experience and 43 respondents (45.7%) with > 4 years experience. Further, a Chi-square test was calculated to check if having a problem with depth of inheritance was language dependent, but no statistical result was found (two-tailed, \( df = 2, X^2 = 2.54 \)). Regardless, the response distribution shown in Figure 2 indicates that inheritance depth can cause understanding problems. Chidamber and Kemerer [2] discuss depth of inheritance as a metric and present data on this from libraries of two different sites: at site A (C++ library) only approximately 75 from 634 classes have a depth of 4 or more (median=1, max=8). At site B (Smalltalk library) approximately 550 from 1459 classes have a depth of 4 or more (median=3, max=10). Similarly, Miller et al. [23] present a mean depth of 1 and a maximum of 4 for their inspected C++ code. Viewing these figures with the presented data raises the question are...
programmers are deliberately avoiding the creation of deeper hierarchies or are shallow hierarchies are just a more natural model?

Second, respondents were asked to grade how often inheritance caused them understanding difficulty. It was generally agreed that inheritance can cause understanding difficulties: only 19% of respondents said it never caused any difficulty. The largest proportion of respondents (48.5%), however, said it caused difficulty only occasionally. Of greater significance is that when asked what has caused the most difficulty when trying to understand object-oriented software, inheritance was the second most popular answer after missing or inadequate design documentation (see Section 5.3). Related research has reported that inheritance causes:

- Distributed class descriptions through the hierarchy, i.e., a complete description for a class can only be obtained by examining the class and all of its superclasses [19], [27].
- Class-to-class dependencies. Consequently, a change to class X means a programmer must be concerned with possible side effects in subclass Y [3], [18], [26].
- Understanding a single line of code can become a difficult task, e.g., it may mean tracing a chain of method invocations through inheritance hierarchies to find the code performing the work [26], [27].
- The ability to misuse or inappropriately use inheritance can cause an increase in code complexity: subclasses which do not extend or specialize the concept of their superclass (identified as conceptual entropy) are less intuitive and, therefore, more difficult to understand [3], [9].

All of these inheritance related problems were described by respondents as reasons for causing understanding difficulties.

Third, respondents were asked to grade the usefulness of multiple inheritance (although a subset of the sample may not have been familiar with this concept because not all languages implement it, e.g., Objective-C). The distribution of responses was spread relatively evenly across the middle categories, with minorities of 11.7% and 10.5% reporting it was never and always of use respectively. The utility (and concept) of multiple inheritance may be language dependent, but a Chi-square test did not indicate significance of this (two-tailed, \(df = 8, \chi^2 = 9.00\)). According to Perry and Kaiser [24] multiple inheritance is widely recognized as both a blessing and a curse and the response distribution supports this position. The argument of multiple inheritance versus single inheritance will continue: some arguing that it produces a more complex design, is more difficult to test, is more difficult to reuse, and is easy to abuse; others arguing it maps the reality of the domain being modeled producing a more appropriate design, and it facilitates software reuse and maintenance [7]. While multiple inheritance may be more complex than single inheritance the questionnaire data supports the theory that multiple inheritance is a useful concept. A source of concern, however, is that multiple inheritance is implemented when it is inappropriate. As a consequence, object-oriented software can become more complex than is necessary. One method of preventing unnecessary complexity may be through use of patterns, e.g., see Gamma et al. [12].
5.2.1 Typical method size

Wilde et al. [27] report that maintainers of object-oriented code must often trace through chains of dependencies created by inheritance, a problem compounded by the proliferation of small methods. Small methods, however, have been advocated by much of the work on good object-oriented programming style, e.g., [14], [16]. The data summary in Table 4 displays the frequencies to respondents' typical method size (in executable lines of code) and the upper and lower limits of methods that programmers have written. The response distribution is something of a bell curve: a typical method appears to fall within 12 ± 7 lines of code. While a typical method is less than 20 lines of code long for nearly 90% of the respondents, 44% of respondents reported their largest methods exceeded 50 lines of code.

Table 5 presents the frequencies of a typical method size for the most popular languages (any respondent who circled familiarity with more than one language could not be included in any of these columns owing to inability to distinguish which language was being described). Wilde et al. [27] present an analysis of three software systems, reporting that 50% or more methods are fewer than 4 Smalltalk lines or 2 C++ lines independent of the application domain. The data presented above does not seem to support this finding, but is more supportive of the data presented in Miller et al. [23] who report a mean method size of C++ code for experienced programmers as 9 lines of code (range 1 - 35).

Table 4: Frequency of typical method sizes and method ranges

<table>
<thead>
<tr>
<th>Typical method size</th>
<th>Frequency</th>
<th>%</th>
<th>Method range</th>
<th>Frequency</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 4 lines</td>
<td>20</td>
<td>8.0</td>
<td>1 - 50 lines</td>
<td>116</td>
<td>55.8</td>
</tr>
<tr>
<td>5 - 10 lines</td>
<td>101</td>
<td>40.2</td>
<td>1 - 100 lines</td>
<td>49</td>
<td>23.6</td>
</tr>
<tr>
<td>11 - 20 lines</td>
<td>104</td>
<td>41.4</td>
<td>1 - 150 lines</td>
<td>8</td>
<td>3.8</td>
</tr>
<tr>
<td>&gt; 20 lines</td>
<td>26</td>
<td>10.4</td>
<td>1 - 200 lines</td>
<td>22</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - 250+ lines</td>
<td>13</td>
<td>6.3</td>
</tr>
</tbody>
</table>

5.3 Difficulties in understanding an OO program

An open-ended question asked what causes the most difficulty when trying to understand an object-oriented program. The most frequently appearing answers were:

1. Missing or inadequate design documentation (16.8%).
2. Inheritance (15.5%).
3. Poor or inappropriate design (including inappropriate use of OO concepts) (12.9%).
4. Tracing a line of method invocations to find the method which performs the work (8.6%).
5. The C++ programming language (including syntax, languages obscurities, and the friend function) (7.3%).
6. Method naming confusions (including obscure and inconsistent naming) (4.7%).
7. Experience with the procedural paradigm before learning the object-oriented paradigm (3.9%).
8. Relationships between classes and how objects communicate (3.9%).
9. Polymorphism (3.4%).
10. Understanding 'clever' coding styles (3.4%).

Other less popular responses were dynamic binding, hybrid code (mixture of both object-oriented and conventional code), knowing what code to reuse from a class library, and the splitting up of a system into many small files. There were, however, 5 subjects that stated that nothing specific had caused them understanding difficulties.

The survey data supports the premise that missing design documentation is a major source of heartache to developers attempting to understand what a software system is doing, and how it is doing it. As Davis [8] states,
Design without documentation is not design. I have often heard software engineers say “I have finished the design. All that’s left is its documentation.” Can you imagine a building architect saying “I have completed the design of your new home. All that’s left is to draw a picture of it”? Moreover, the data supports the premise that object-oriented systems are equally susceptible to missing design documentation and the difficulties it causes.

Inheritance is reported as the second most popular reason for causing understanding difficulties in object-oriented systems. Possible reasons were discussed in Section 5.2.

Finally, the data suggests that poor or inappropriate design is a major source of understanding difficulty. (In the context of object-oriented systems, poor design includes inappropriate use of OO concepts, inappropriate abstractions, and unnecessary complexity.) Further, design must be considered with reference to inheritance. A poorly designed hierarchy will compound the problems discussed in Section 5.2. This is not a surprising result, but it does strengthen the argument that the object-oriented paradigm is not a panacea: object-oriented systems must still be appropriately designed. Failure to do so severely affects their understandability.

Missing or inadequate design documentation and poor or inappropriate design (two of the first three in the above list) are not paradigm specific and are concerned with the deficiencies of current software engineering practice. Perhaps an improvement of current practice would offer more advantages than simply making the transition to the object-oriented paradigm.

5.4 Maintenance of conventional and object-oriented programs

Three closed questions were asked about software maintenance. First, respondents were asked if continual maintenance of conventionally (i.e., structured) designed software would lead to unmaintainability. The largest proportion (43.7%) of respondents circled category 4, i.e., they thought this would usually happen. Second, respondents were asked the same question, but in the context of object-oriented designed software. The largest proportion (48.5%) of respondents circled category 2, i.e., they thought this would occasionally happen. A Wilcoxon matched-pairs signed-ranked test (related) was calculated to test for ordinal level differences between the two responses. The results show significance at \( p < 0.01 \). Thus we conclude that respondents regard object-oriented software less likely to lead to unmaintainability.

Further, to examine the attitude of practitioners regarding object-oriented software facilitating maintenance, respondents were asked directly whether object-oriented software was generally more maintainable than the equivalent conventionally designed software. The largest proportion of respondents (58.4%) circled category 4, i.e., they thought this would usually be true. This statistic, however, does not paint a complete picture: disclaimers explained that this would be true if the software was designed well; if not then object-oriented software will be more difficult to maintain because of more complex and less intuitive inter-relationships.

5.5 Software reuse through in house (local) class libraries

It is argued that object-oriented software facilitates reuse. As a consequence, frequent use of in house class libraries might be expected. Respondents were asked to grade their use of such libraries; responses received were divided between infrequent users of in house class libraries (those that said they never or only occasionally used them 43.3%) and frequent users (those that stated they usually or always used them 42.2%). A Chi-square test found no significance difference (two-tailed, \( df = 2, \chi^2 = 3.1 \)) between the language known and frequency of use of in house class libraries. In this survey, therefore, we conclude that use of in house class libraries is language independent. Interpretation of the almost equal split, however, is difficult without comparative figures for in house software libraries for the structured paradigm. Examining the complete dataset, however, shows 75.4% of respondents make use of in house class libraries at least occasionally: although almost one quarter of respondents never make use of in house class libraries, it does appear that local software reuse is becoming widespread.

A small subset of respondents, however, warned that time can be wasted trying to find existing code to reuse, or understanding what code there is in an attempt to reuse it; Wilde and Huitt [26] note “if the reuse benefits of OOP are to be achieved it must be possible to locate the code to be reused fairly efficiently.” An area of growing interest is Christopher

\[ 12 \text{ respondents circled a number greater for the second question, 148 respondents circled a number less for the second question, and 70 respondents circled the same number for both questions; two-tailed, } N = 230, Z = -9.458. \]
Alexander’s notion of patterns and pattern languages, developed for describing architectural constructs and now borrowed by OOA and OOD, which it is argued may reduce these problems. Alexander states,

> each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. [12]

Patterns are discovered by experience, and are described in a manner that emphasizes potential reuse: the pattern can be used as a higher level building block for OOA and OOD. More information on Alexander’s pattern concept with respect to OOD is provided in the excellent book by Gamma et al. [12].

5.6 C++ as the de facto standard object-oriented language

Three questions were asked about the C++ language. Respondents were told only to answer these questions if they had relevant experience. First, a closed question measured attitudes towards C++ becoming the industry de facto standard object-oriented programming language. Table 6 provides the response distribution and shows the largest proportion of respondents (60.9%) regarded this as ‘bad’ and a much smaller proportion (17.9%) regarding this as ‘good’.

A subset of respondents explained the reasons for their answers:

**Bad:** all the problems of C exist in C++; very obscure syntax; many of its features have been ‘hacked’ together, e.g., multiple inheritance; it allows the writing of straight C, i.e., not forced to write OO code; easy to override data security and break encapsulation, e.g., friends; almost blackened the object-oriented paradigm’s reputation.

**Good:** efficient in comparison to purer OO languages; can make use of existing C libraries; many developers know C so the transition to C++ is easier and less costly; C++ has given many exposure to the object-oriented paradigm and as a consequence object-oriented programming has become mainstream.

Of interest is the difference of opinion between the academics, software engineers, and project managers. The data supports the belief that academics hold a more purist view of the object-oriented paradigm. (Note from the table that no academics responded it is good that C++ has become the de facto standard object-oriented language). Software engineers and project managers appeared more pragmatic in their responses.

A second question measured attitudes towards C++ allowing a mixture of object-oriented and conventional programming. The majority, 108 respondents (45.8%) regarded this a disadvantage, 86 respondents (36.4%) regarded it an advantage, and 25 respondents (10.6%) decided it was both. Again, a subset of respondents explained their reasons:

**Disadvantage:** leads to a mess of mixed metaphors which is harder to read and understand; for full encapsulation and ease of maintenance, software should be consistently designed in one paradigm, not two; allows for the sloppy approach because it enables reverting to conventional programming if the programmer runs into difficulties (especially novice OO programmers); complexity is increased; enables a gradual migration to OO which impedes the thought process; can claim that OO code is being written when it is not.

**Advantage:** OO is more appropriate at the design level - at the implementation level it helps to be able to perform conventional decomposition of methods; it allows freedom to choose the most appropriate technique; not all problems facilitate an OO solution, but the environment can be used regardless; transition for programmers with a conventional background can be made gradually, making it easier and maintaining programmer productivity.

Third, in the context of performing maintenance, respondents were asked about making use of the Friend function\(^3\) to prevent redesigning the inheritance hierarchy. 69 respondents (38.1%) said they would make use of a Friend function in this manner.

\(^3\)Friends of a class are trusted with access to the private and protected members of that class.
only occasionally; further 45 respondents (24.9%) said they would never make use of a Friend function to do this. Of concern, however, was the frequency of respondents who said they would use a Friend function to prevent this more than just occasionally: 37 respondents (20.4%) said sometimes, 24 respondents (13.3%) said usually, and remarkably, 6 respondents (3.3%) said always. Stroustrup [25] states that friends should only be used to avoid (a) global data, (b) global (non member) functions, and (c) public data members. Unfortunately, the Friend function is easily abused; apparently programmers are using it to cut corners by breaking encapsulation enabling another class to directly manipulate the private data. Subsequent testing and maintenance becomes more difficult because class relationships become more obscure and more complex.

6 Conclusions

The study has concentrated on the following issues: the perceived benefits of the object-oriented paradigm over conventional paradigms, inheritance, the understanding difficulties of object-oriented code, software maintenance and its consequences, use of local class libraries, and the C++ programming language. While there is difficulty in generalizing from a subset of the population to the actual population itself, the responses of 275 object-oriented practitioners on these issues should be considered important. There are, however, many issues about the use of questionnaires that affect the results of a survey, e.g., the problems of self-selection, questionnaire distribution, question biases; although these have not been discussed here, their effects on our results have been carefully considered. Full details can be found in our technical report [5] where we conclude that such issues have had a minimal effect on this survey's results.

Results have shown: First, respondents are of the view that the object-oriented paradigm is more advantageous than conventional paradigms in terms of ease of analysis and design, software reuse, programmer productivity, and ease of maintenance. Second, inheritance can cause difficulties when trying to understand object-oriented software: only 25% of respondents reported it had never caused them difficulty. More significant was that inheritance was catalogued as the second largest reason for understanding difficulties. We hypothesize that understanding becomes more constrained with a deeper hierarchy (55% of respondents indicated depth of inheritance is a concept which can introduce difficulties). Third, from the list of catalogued reasons for understanding difficulties, two of the first three are not paradigm specific: missing or inadequate design documentation and poor or inappropriate design and are still prevalent problems. The advantages that an improvement of current software engineering practice would bring, regardless of paradigm, should be considered. Fourth, maintenance is still perceived to cause software degradation, but respondents viewed (with statistical significance) this occurring less frequently provided the system and the change are well designed. Further, well designed object-oriented software is regarded to be more maintainable than equivalent conventional software. Fifth, respondents indicated that the C++ language has many deficiencies in comparison to 'purer' object-oriented languages. Consequently, the majority viewed the fact that C++ has become the industry de facto standard as detrimental.

This survey has been the second stage in our programme of research, initiated by structured interviews. Questions were based on the interviews findings and have shown consistency across a wider practitioner group. We have also recently conducted a series of controlled experiments investigating our hypothesis regarding depth of inheritance, the results of which can be found in [4].

Acknowledgments

The authors wish to thank those who took the time to participate in the survey. Acknowledgment must also be given to the Computer Science Department at the University of Strathclyde for meeting the cost of the mail shot, and to Mr Peter Day, Managing Director of Cocking and Drury (London) who gave one-off access to the company address list.

References


---

All self references are available via the World Wide Web: http://www.cs.strath.ac.uk/Contrib/efocs/index.html


