Abstract

Students who had taken an object-oriented languages and systems class were surveyed, as were managers in object-oriented technology. From among the students, a “practicing o-o programmer” subgroup was identified. The aim of the survey was to determine how best to teach object technology. Both the programmers and managers agreed that o-o analysis and design need to be stressed at least as heavily as programming skills. The choice of an OOA/OOD methodology was thought not to be critical. Nor were OOA/OOD tools thought to be necessary. The most desired o-o languages by far were Smalltalk and C++; the choice of language was considered quite important. Respondents favored integrating the teaching of OOA/OOD with programming, instead of teaching them in different courses.

1. Introduction

When incorporating object-oriented technology (OOT) into a curriculum, the instructor must choose among many options. Emphasis must be divided between general concepts of analysis design, and programming, and specific training in methodologies, languages and environments. The ultimate goal of a curriculum is not to teach students specific tools, but rather to educate them to appreciate many aspects of this challenging technology. Curriculum design must also take into account that industry is demanding certain OOT skills which help to make a student more employable.

This paper attempts to identify these skills through surveying two groups of individuals: (1) OOT managers in various industries and, (2) former students from the first author’s OOT course. Each group responded to a different questionnaire; then representatives of both groups answered followup questions to explore the reasons behind their answers.

The questionnaires probe issues such as which OOA&D methodologies and languages should be taught, whether OOA/OOD software tools are essential in class, how much time should be spent on teaching class libraries, and how a multi-course sequence in OOT should be organized. Comments from the respondents as well as the authors are used to place the results in context. It is expected that the data can be of value in planning curriculum and course content at both the undergraduate and graduate levels.

Section 2 considers the groups involved in the surveys. It is followed by Section 3, which deals with o-o analysis and design, and Section 4, which covers language issues. Section 5 considers course organization, and Section 6 summarizes the findings.

2. The Surveys

This paper is based on two surveys: a survey of students who have taken a course in object-oriented languages and systems, and a survey of managers who direct object-oriented programming projects. The students surveyed were those who have taken the first author’s course since 1989. The student survey is a continuation of the project reported in [Gehr 93]. Altogether, 604 students have taken this course over that period. Responses were re-
ceived from 187 of them, a response rate of 31%. Because the responses were received over a 3-year period from 1992 to 1995, care must be taken in interpreting some of the results; in particular, students' perceptions of the usefulness of knowing various languages may have changed significantly over that time. Other results should not depend significantly on when the questionnaires were returned; these are questions such as the difficulty of learning the languages and the helpfulness of seeing the Smalltalk environment displayed on a screen during lectures.

After the questionnaires were returned, various followup questions were e-mailed to selected groups of students. One such group were former students who are now working in various areas of object technology (henceforth known as the “practicing o-o programmers”). This group was identified by taking the 52 students who answered “strongly agree” to the question, “I was immediately able to apply what I had learned in the course to my work or studies outside the course,” and eliminating those who are still full-time students. Of the 52, we had valid e-mail addresses for 37.

Since only about one-third of the students who have taken the first author’s course responded to the survey, a question arises as to whether a self-selection bias may exist. We do not have a complete answer for this, but some evidence can be obtained from the following. Students were permitted to respond to the survey anonymously, by sending e-mail to an account that forwarded it after removing the sender’s address. Only 12 of the 187 chose to respond anonymously. On ten of the sixteen questions with numerical answers answered by more than one anonymous respondent, the average response of the anonymous students was within 0.31 of the average of all students’ responses. The anonymous respondents thought it was harder to learn Smalltalk and C++, and thought the pace of the course was faster. On all questions relating to the usefulness of the course and the live lectures, their responses closely matched those of the entire sample.

The managers included in the other survey are mostly either contacts made by the second author in seeking feedback from industry for her courses or managers of students who have taken the first author’s course. In three cases, they were students in the course who later moved into management roles. A few of the respondents were met at trade shows. In all, 52 managers responded, from 28 companies in 17 states. One response was received from another country (Argentina). Twelve of the surveys came from managers in different IBM departments.

No other company contributed more than three responses.

To test whether the sample might be perturbed by including so many respondents from one survey, we excluded the IBM personnel from the list and looked at the resulting average response to the questions. With a scale ranging from 1 to 5, no average changed by more than 0.20 from what it was in the full sample. Twenty of the managers were from North Carolina. When their responses were removed, no average for any question changed by more than 0.33. This suggests that the results have not been seriously perturbed by the influence of a nonrepresentative subgroup.

3. The Importance of Object-Oriented Analysis and Design

Until now, object-oriented analysis and design has not been a major topic of the first author’s course. Only the CRC card method has been taught.1 One question on the students’ survey asked if the course would have been more useful if OOA and OOD had been covered in greater depth.2 The thirty-eight students who responded agreed with the statement, giving it a composite score of 4.06. A followup question was then posed to the managers and practicing o-o programmers asking how much time should be devoted to OOA/OOD relative to OOP in the “ideal” course. Both groups thought that the two should be equally stressed: the 17 students who responded on average said that OOA/OOD should be given 50.3% of the time; the 22 managers’ responses averaged 51.5%.

What reasons underlie this emphasis on analysis and design? One factor is maintainability. One o-o programmer put it like this: “I think Lincoln once said that you should spend 80% of your time planning and 20% doing. I don’t agree exactly with his numbers, but I do believe that we programmers are much too anxious to jump in and get our hands dirty without fully thinking things out. It seems like every day someone is saying, ‘I wonder why they laid out the hierarchy like that … Why does he inherit from him … Shouldn’t they have made an Abstract Base class here… etc. etc. etc.’”

"In the o-o world, I think that some stringent up-front OODA is extremely important—much more than top-down analysis was in procedural pro-

1The second author has emphasized OOA and OOD much more in her courses.
2This question was asked for the first time in 1995.
gramming. I believe that o-o spaghetti code is much more difficult to deal with than procedural spaghetti code.”

3.1 Which methodology to teach?

The Booch methodology is the most widely used, and in our survey it was most desired by managers. On the question, “What o-o analysis/design methodology would you like a recent graduate to know?” 24 cited Booch, compared to 21 for Rumbaugh and 17 for Wirfs-Brock, the runners-up. No other methodology was named more than six times (Table 1).

<table>
<thead>
<tr>
<th>Table 1: O-o design methodologies considered important</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Booch Method (Booch) .............................. 24</td>
</tr>
<tr>
<td>Object Modeling Technique (Rumbaugh) ..................... 21</td>
</tr>
<tr>
<td>Responsibility-Driven Design(Wirfs-Brock) .............. 17</td>
</tr>
<tr>
<td>“Any is acceptable” ........................................ 13</td>
</tr>
<tr>
<td>OOSE (Jacobson) ............................................. 6</td>
</tr>
<tr>
<td>Fusion (Hewlett-Packard) ................................... 4</td>
</tr>
<tr>
<td>Information Engineering with Objects (Texas Instruments) .... 4</td>
</tr>
<tr>
<td>Class-Centered Modeling (Eric Adanow) .................... 3</td>
</tr>
<tr>
<td>Coad/Yourdon .................................................. 2</td>
</tr>
<tr>
<td>OO Software Engineering (Martin/Odell) ................... 2</td>
</tr>
<tr>
<td>(Seven other methodologies were cited once.)</td>
</tr>
</tbody>
</table>

Those citations should be kept in perspective, given that most of our managers didn’t have a solid preference for a student to know any particular method. They strongly believed (4.75 on a scale of 1 to 5, with 5 being “agree strongly”) that it is more important for employees to understand the basics of object-oriented design and analysis than any particular methodology. Their equanimity with respect to design methodologies did not extend to languages, however. The statement, “I believe it is more important for a recent graduate to understand the basics of o-o programming than any particular language” received an average score of only 3.62.

Followup questions to managers and former students working in OOP shed some light on this answer. One reason that specific languages are given greater weight than specific OOA/OOD methodologies is that the differences between languages are better known. “What about a method and corresponding CASE tool? Booch, Rumbaugh, Shlaer-Mellor? I don’t know which is the best. And frankly … it doesn’t really matter. The method is a means, not the end,” remarked one o-o programmer. Indeed, when asked what OOA/OOD method they wanted recent graduates to know, thirteen of the managers checked “any is acceptable.”

<table>
<thead>
<tr>
<th>Table 2: Managers’ perceived importance of design and programming skills (scale of 1=unimportant to 5=very important)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.75 I think it is more important for a recent graduate to understand the basics of o-o analysis and design than any particular methodology.</td>
</tr>
<tr>
<td>3.62 I think it is more important for a recent graduate to understand the basics of o-o programming than any particular language.</td>
</tr>
<tr>
<td>3.04 I think it is more important for a recent graduate to know an o-o analysis and design methodology than to know an o-o programming language.</td>
</tr>
</tbody>
</table>

Followup questions asked for suggestions on books that could fit well into a course structure. Recommendations closely paralleled the recommendations of methods; Booch’s *Object-Oriented Design with Applications* [Booc 94], Wirfs-Brock’s *Designing Object-Oriented Software* [Wirf 90], Rumbaugh’s *Object-Oriented Modeling and Design* [Rumb 91], and Jacobson’s *Object-Oriented Software Engineering* [Jaco 92] were the top four on the list. The next two are noteworthy books not associated with any particular methodology, Gamma, Helm, Johnson, and Vlissides, *Design Patterns* [GHJV 95]; and Bertrand Meyer’s *Object-Oriented Software Construction* [Meye 88].

3.2. Choice of methodology not critical

Even the detailed comments on the questionnaires failed to reveal any strong partisans for a particular methodology. The only detailed rationale was provided by an experienced Smalltalk and C++ developer who favored teaching several methodologies:

> “After the students have a couple of short programming assignments under their belt (but ones that required them to use all of the above concepts) they could then tackle the issues involved in OOA&D. You could then introduce them (in the following order) to Jacobson’s OOSE and Use Cases, Wirfs-Brock’s method and CRC cards, the Booch method, the OMT method, etc. This might take up 30-40% of the course.”
When questioned on why he suggested teaching more than one methodology, he put it this way:

“I may be about the only person in your respondents who has seen the limitations of the various methods. Let me state, though, that it’s not just me — this is a common belief in the industry.

“The methods themselves are all incomplete in one way or another. For instance, the most popular, Booch, is extremely weak in the analysis side. In his most recent book (Booch94) he even admits it, more or less, and suggests that people use Use Cases and CRC cards as preliminary analysis steps to his design method. [See page 158-159 of Booch 94.]

“On the other hand, Wirfs-Brock, while very strong in analysis, is very weak in detailed design, and does not, for instance, provide for state transition diagrams as part of her design method. While Jacobson is very good at front-end analysis, some of his conclusions (Use Cases are first class objects) are a bit far-out for most practitioners.”

The nonpartisan nature of the methodology recommendations contrasts sharply with the managers’ distinct preference for certain o-o languages (see Section 4.1). Followup revealed two related reasons for this: object-oriented methodologies are much less standardized than languages, and it is possible to draw on experience with multiple methodologies in the course of a software project far more easily than one can apply experience with multiple languages. In support of the first reason, one manager cited “the toolbox analogy.”

“Once you learn the basics of OO technology, I believe you can follow a particular methodology. Once you have followed one methodology completely through the lifecycle, it becomes easier to pick up other OO techniques (e.g., design patterns). The industrial world is filled with multiple methodologies, even some homegrown in-house styles. … Where do the two above differ? I think a recent graduate needs have the toolbox of OO A&D, but should be on [the] way to becoming an expert with a language.”

However, pragmatic considerations may dictate hiring a programmer who knows a particular language, as another manager pointed out:

“I want to hire people who have had some real experience with certain languages. My company, a small startup, can’t afford to spend lots of time training graduates [in] Smalltalk when all they have had is C or C++.”

3.3 Should OOA/OOD tools be used?

One would certainly not attempt to teach an OOL without a compiler, but can one teach an OOA or OOD methodology without software tools? When we posed this as a followup question to our practicing programmers, the results were surprising: they indicated that special software might not be necessary. The question we posed was, “On a scale of 1 (= not important) to 5 (= very important), how important is it to have on-line tools available when teaching OOA/OOD?” The average was only 3.33, although caution is advised in generalizing, since there were only 12 responses. The “pro-tool” camp is typified by this response:

“In the end, if there is no tool set available then there is no way to practice what has been preached. Some of the OOA&D methodologies are extremely complex, and without a toolset, nearly impossible to produce by hand.”

The middle ground, illustrated by this response by a Ph.D. student who has done some teaching, is that tools can be a useful adjunct:

“My experience has been that undergrads have difficulty decomposing a problem into objects and grouping objects at the right level. They also have difficulty grasping the abstract nature of having a number of autonomous agents with no obvious thread of control like you have in a block-structured program. … If there are on-line tools to help that development, that’s great, but I’ve seen good results without any such tools.”

On the other hand, the difficulty of learning to use tools gave some pause: “It’s often very difficult to know how much use a project will make of a tool. Tools are often very big, maybe even clumsy when trying to use all the diagram editors, etc. They can be frustrating unless the developers have used [them] before.” The same respondent downgraded tools for being of limited use in program maintenance:

“More often than not, the diagrams drawn/generat​ed prior to programming will
not be consistent when the code is completed. The very act of programming, in my experience, provides another iteration of the design and thus the design changes! … Therefore, some/most/all of the design diagrams will not be consistent with the code. Unless the tool can generate correct diagrams from code, the diagrams should be corrected by hand or even discarded.”

One might wonder whether those who had used tools would rate them more highly than those who had not. So respondents to the tools question were asked if they had ever used tools. Seven of the ten who responded this time had used tools, but their composite ranking was 3.00, lower than the 3.67 average of the three who had not. Again, the number of respondents is far too small to draw any firm conclusions.

4. Language Issues

An old debate among o-o educators community concerns whether previous experience in a non-o-o language is a help or a hindrance. Various pundits have opined that once one’s mind is attuned to thinking about problems in a non-o-o way, it is more difficult to overcome that “interference” than it would be to learn o-o programming from scratch. We posed that question to our managers in followup to the survey. On our scale of 1 to 5, the average response was 3. The difficulty of reaching a definitive conclusion is illustrated by this response:

“Probably somewhat [of a hindrance], but I don’t think a major obstacle. Our entire project (=20 developers in ’89) went from structured design & programming to learning o-o. I don’t think anyone had a major hard time, but then you might say we are not the best o-o developers because of our background. Hard to compare that with a group that learned o-o first.”

Among the difficulties of moving to o-o, our managers rated “retraining employees” (3.90) as more of a challenge than “few qualified recent graduates” (3.56; difference significant at the 90% confidence level). This raises the following question, which was asked in followup: “Are recent graduates better prepared to tackle o-o technology than programmers with several years’ experience?” Again, the results were noncommittal, with the composite of six responses netting a 3.2 average. Not surprisingly, the answer depends very much on the skills of the programmers in question:

“Yes and no. We’ve found that yes, recent graduates have more preparation but once we give training to the people who have been our good contributors for years prior to oo that it doesn’t take them long to get on equal footing (1-2 years) with the recent grads. Then they pass the recent grads because they have the maturity level which only experience in the ever-changing industry brings.

“Now, average to below-average contributors are a different story. For people who were struggling to do well in the non-o-o world but surviving, it seems that no amount of training helps them over the hump to even compete with the recent grads. I can’t really explain why that is but I know it happens.”

Another respondent said that his organization had “had better luck with MS degree graduates” than with bachelors’ graduates.

4.1 Which language or languages?

Among languages that managers would like recent graduates to know, not surprisingly, C++ led the pack, with a composite score of 4.41 on the scale of 1 to 5. Smalltalk, rated at 4.10, was not significantly different. The survey explicitly queried managers about three other languages: CLOS, Objective C, and Eiffel. The managers rated these significantly lower (at the 95% confidence level) than the first two. Eiffel and Objective C were next, with scores of 2.41 and 2.33 respectively. CLOS finished last at 2.09, significantly behind the others at the 90% confidence level. Eiffel’s margin over CLOS was unexpected, given the much larger number of programmers who have access to CLOS. Other than the explicitly mentioned languages, the only write-in votes went to Ada 95 (3) and Visual Basic (2).

Table 3: Managers’ perceived importance of o-o languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>4.41</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>4.10</td>
</tr>
<tr>
<td>CLOS</td>
<td>2.09</td>
</tr>
</tbody>
</table>

3All differences reported as significant in this paper are at the 95% confidence level unless otherwise noted.
The “trendiness” of various o-o languages has varied widely during the ten or so years since o-o programming has come into vogue. Objective C enjoyed an initial flurry of support, but was supplant ed in the late ’80s by C++ as the most popular C derivative. Smalltalk has been making a strong comeback in the last two years. One measure of this is how our students’ responses have varied when asked about the value of learning Smalltalk, C++, and Eiffel, the three languages that have been taught since 1992 in the first author’s course. The students responding to the survey in 1992 ranked the value of learning Smalltalk at 3.64 on a scale of 1 to 5; this rose to 4.21 in 1993, but fell to 3.76 in 1994. Covering C++ in more detail was considered desirable at a 3.69 clip in by those responding in 1992. This fell to 3.07 in 1993 after instruction expanded from one week to three. In 1994, it increased to 3.96. The usefulness of learning Eiffel was ranked at 3.17 by 1992 respondents, but this fell to 2.61 in 1993 and 2.64 in 1994, reflecting installation difficulties with the graphical Eiffel environment in both Fall 1993 and Fall 1994.

Another reason is the ambivalent attitude of many computer scientists toward C++. From a purist’s perspective, it’s probably not the language one would want to use to teach the object-oriented paradigm. But from a pragmatic point of view, the demand for C++ programmers makes it difficult for a curriculum to ignore.

If two languages are to be taught, then which should be first? A followup question posited Smalltalk and C++ as the two languages. The six who responded gave a unanimous verdict: Smalltalk. As one manager put it,

“Smalltalk without a question! C++ has too many pitfalls and constructs that will get in the way and confuse what is OO and what is not. Once they learn OO via Smalltalk they will be better C++ programmers. By the way, I asked this same question to a bunch of our own project managers on our big client projects and they came up with this response that I’m giving you.”

Another continued, “C++ grammar, pitfalls, etc., greatly get in the way of developing good OO systems. … [C++] is the only [language] I know of where you can go to a conference and take a course on the ‘gotchas’ of the language.”

For all its shortcomings, the students in the survey rated C++ the easiest language to learn. When asked how easy it was to learn to use each of these languages, with 1 meaning very hard and 5 very easy, they rated C++ 3.78, Smalltalk 3.14, and Eiffel 2.79. All of the differences are significant.
It is widely suspected that C++ may be easier because it allows programmers to lapse into familiar non-o-o coding patterns. In followup, we asked the managers how significant the problem was. They were given three choices: (a) not very bad, (b) moderately bad, and (c) a severe problem. Eight of the ten who responded chose (c), while the other two chose (b). “They copy pieces of existing C code into new C++ code,” said one manager, “and leave it as is rather than redesigning in an OO fashion.”

Another related two anecdotes: “How do I implement this flowchart in Smalltalk?” and a printout of a program with the following characteristics:

“1. Class — program
2. Methods — main();
Lines of code for the method main: 11,211.”

4.2 Class libraries

Except for C++, most object-oriented languages are reasonably “small.” It is possible to teach the syntax of Smalltalk in a single 75-minute lecture. But not until one assimilates the Smalltalk class library does one become an effective Smalltalk programmer. How much time should be spent on teaching class libraries? The first author’s course spends about one week covering the high points of the Smalltalk class library. Is this enough? The student survey asked about agreement with the statement, “The course would have been more useful if we had studied standard class libraries in greater depth.” The average response was 2.9 on a scale of 1 to 5, indicating that the students are less enthusiastic about devoting greater time to this topic than to OOA/OOD or C++.

One programmer offered a rationale for counting OOA/OOD as more basic:

“Even if you teach good ways to use good class libraries, that’s half the battle. You’re likely to lose ground when people use non-o-o techniques (or misuse o-o techniques) for the rest of their code because they haven’t got the skills for doing OOA/OOD first.”

The managers were asked which areas recent graduates needed more training in. OOA/OOD, language syntax and semantics, OOL environments, or class libraries. Only the score for language syntax and semantics differs significantly from the others, suggesting that graduates are no worse off in their knowledge of class libraries than in other major areas of object technology.

<table>
<thead>
<tr>
<th>Table 4: Managers’ perceived weaknesses of graduates</th>
</tr>
</thead>
<tbody>
<tr>
<td>After we hire them, recent graduates seem to need training in—</td>
</tr>
<tr>
<td>4.28 O-o analysis/design methodology.</td>
</tr>
<tr>
<td>3.73 O-o programming language(s) syntax and semantics.</td>
</tr>
<tr>
<td>4.04 O-o programming language(s) environment.</td>
</tr>
<tr>
<td>4.18 O-o class libraries.</td>
</tr>
</tbody>
</table>

5. Course Organization

It is not obvious how courses in OOA/D and OOP should be organized. Having an OOA/D course and an OOP course does not seem to be a good idea. Teaching programming before design will lead to bad design practices, which will take effort to “unlearn.” Teaching design before programming could result in a sterile course where students were unable to fully apply what they had learned. Nonetheless, one of our managers did suggest teaching an OOA/D course first, and defended it this way:

“[T]he reason I suggest this (and maybe it’s not practical) is that I’ve seen so many OOP courses without any kind of OOA/D at any point that I’d like to see the other extreme tried. The concepts are so much more important than the language. However, with any language comes idioms which are equally important and equally ignored by instructors. …

“Just like you can do a lot of database work with pseudo-languages and entity-relationship diagrams, you can do a lot of OO work without any OOP language at all. Just learning and applying the models of any of the methodologies would be plenty of work. Being able to express a model in code later on would almost be trivial if the A/D work was properly done.”

Support for this position came from one of the programmers:

“1. I believe students come into an OOA class with less “baggage” then they do an OOP class. Generally, I believe the student is less prepared to do modeling and that is an advantage. It gives the instructor a chance to present it right the first
time and not have to spend time "unlearning".

"2. We want software engineers to be good abstract thinkers and OOA/OOD will help accomplish this. It also helps the developer think about the problem at a higher level. This helps them to organize the problem to better handle complexity.

"3. Mixing OOA and OOP tends to dilute the OOA part. Often, especially with C++, you have to spend so much time describing the syntax that you cannot adequately cover the OOA part.

"4. The ideas of OOA can be used with any language. Providing the student with a language-independent view is helpful in that the student may not be able to use the language covered in class [in a work environment]."

Almost all of the other respondents preferred a mix of OOA, OOD, and OOP in each course. This was one of the recommendations:

"I would spend 50% of the course on OOA and Design and the remaining 50% on programming. However, I would not do a linear progression. I would cover design in theory while the students complete a design for a project and then I would have them program, ... When the program was complete, I would review the design a second time and offer an opportunity to reimplement. You can only talk abstract for so long and then you have to let them code, but continue to oscillate between the two. If I were to change those percentages, I would spend more time with design because once they can design they will find a way to program. There are enough differences between Smalltalk, Eiffel, C++, etc. that learning one language thoroughly is unimportant in the classroom. In fact, if they learn to design based on one language, rather than based in theory, their designs may not be as good."

One of the managers suggested an adaptation of Kent Beck’s "boot camp" approach:

"Kent Beck was describing his boot camp to me — it sounded very interesting. A team solves a complete problem in three days: 1/2 day OOA/D, 1 1/2 day implementation, 1/2 day building test suites, documenting and 1/2 day packaging. I like this approach because it concerns itself with all aspects of building systems — not just the obvious A&D and implementation. It is these other aspects — documentation, testing, and packaging/deployment, that causes the most problems in real projects. I think it would be worthwhile to extend this concept to a 2-semester coverage of the topic, where all aspects were covered in each semester — but the first semester concerned itself with a much simpler project, and the second ramped up."

As a followup question to both programmers and managers, we proposed two possible organizations for mixed OOA/OODD/OOP courses (with their average ratings by 19 respondents in brackets):

a. A one-semester course in OOP and OOA/OOD (one language covered) followed by another course in which additional languages are taught and a larger project is done. [3.8]

b. A one-semester course in OOP and OOA/OOD (two languages covered) followed by a project course in which a team of students designs and implements a software project of substantial size. [3.2]

Both of these proposals beat a "programming first, then analysis and design" curriculum [2.8].

Some of the respondents encouraged making a software-engineering course a prerequisite for the sequence. This would be quite difficult to implement, whether in a graduate or advanced undergraduate environment, because a three-course sequence does not give the student much scheduling flexibility. A workable alternative would be to make the SE course and the first o-o course prerequisites for the second o-o course.

5.1 Programming teams

Applying o-o analysis and design techniques in a project of reasonable size will require a team approach. How large should student programming teams be? This was posed as a followup question to both the programmers and managers. Recommendations varied from two to eight, with a mean of about three. Those who favored smaller teams cited these reasons:

- Students will have an easier time getting together. This is especially a problem if some of the students are part-timers. One of our respondents offered, "This may become less..."
of an issue in the near future as more and more of our students gain access to the Internet and can meet electronically.”

- Teamwork can be learned in other courses. This, however, does not speak to the issue of applying OOA/OOD techniques in a large project.

Those who favored larger teams made these points:

- Teams of 3–4 are what they will find in the job market.
- There is less of a chance that one programmer will carry the team.
- Communication problems are more realistic in larger teams. “[C]ommunications problems increase factorially with the size of the team,” an experienced instructor noted.
- To get a feel for advantages of o-o for interfacing between different programmers. “[I]t would expose the students to one important use of classes: as interfaces between different programmers.”

Nonetheless, larger teams make it more of a challenge to keep everyone involved. As one of our respondents observed, “This is why it’s real easy for a team of three to become a team of two with one hanger-on. The two have found themselves to be mutually compatible, but have enough problems communicating with each other, never mind the odd one out.”

6. Summary

Both programmers and managers recognize that the importance of object-oriented analysis and design is growing as software systems increase in complexity. Knowing the principles of OOA and OOD is considered more important than being familiar with any particular methodology. In the language field, however, knowledge of C++ and/or Smalltalk is essential.

On several issues, no consensus emerges. There is disagreement over whether learning OOA/OOD is easier with software tools than with pencil and paper. There is no good evidence on whether a non-o-o background makes it more difficult to learn o-o well. Some believe that recent graduates learn more quickly, while others think that veteran programmers’ experience gives them the advantage in the long run.

Our respondents believed that the o-o programming paradigm is best learned if more than one language is taught. If both Smalltalk and C++ are taught, Smalltalk should be taught first. It tends to build o-o skills that are necessary if the tendency to write non-o-o code in C++ is to be avoided. While class libraries are an important part of object-oriented programming, our sample thinks it would be better to emphasize OOA/OOD. Most of them think that OOA/OOD should be integrated into the same courses as OOP, but there are widely varying ideas on how to do this. If the two are not integrated, there seems to be more support for teaching OOA/OOD last.

References


Appendix A: Results of Student Survey

*Note: Part I (Questions 1 to 5) involved identifying information (e.g., did you take the course for credit, were you in industry, etc.).*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>41</td>
<td>36</td>
<td>36</td>
<td>33</td>
<td>14</td>
<td>25</td>
<td>187</td>
</tr>
</tbody>
</table>

**Part II. Facilities**  *(Note: Two students from 1988 also filled out survey.)*

6. Rate on a scale of 1 (= very hard) to 5 (= very easy): How easy was it for you to get access to the software you needed for the course?
   - 1989: 3.45
   - 1990: 3.60
   - 1991: 2.99
   - 1992: 3.47
   - 1993: 4.00
   - 1994: 3.74
   - All years: 3.45

7. Did you find the course unusually expensive, either because of the need to purchase two textbooks, or because you needed to buy software?
   - 13% Yes
   - 21% No
   - 39% Unsure
   - 52% Did not purchase
   - 0% Yes, purchased
   - 33% No, purchased
   - 31% Unsure

8. What version of Smalltalk did you use? [Many different versions; results not tabulated here.]
   - [Many different versions; results not tabulated here.]

9. Did you have problems installing Smalltalk (if applicable)? % answering yes:
   - 9% Yes
   - 6% No
   - 23% Unsure
   - 4% Did not use Smalltalk
   - 0% Yes, purchased
   - 0% No, purchased
   - 10% Unsure

10. Did you have problems installing Eiffel (if applicable)?
    - 44% Yes
    - 33% No
    - 45% Unsure
    - 41% Did not use Eiffel
    - 0% Yes, purchased
    - 50% No, purchased
    - 41% Unsure

11. Rate on a scale of 1 (= very hard) to 5 (= very easy): How easy was it to learn to use Smalltalk?
    - 3.38
    - 3.18
    - 2.91
    - 2.97
    - 3.64
    - 2.92
    - 3.14

11a. How easy was it to learn C++? (Asked starting in Fall 1992.)
    - 3.63
    - 4.23
    - 3.72
    - 3.78

12. How easy was it to learn to use Eiffel?
    - 2.50
    - 2.69
    - 3.11
    - 3.10
    - 2.61
    - 2.52
    - 2.79

**Part III. Delivery** *(1=strongly disagree to 5=strongly agree.)*

13. The lectures by the regular instructor were helpful in understanding the material.
    - 4.13
    - 3.89
    - 3.83
    - 3.97
    - 4.00
    - 4.04
    - 3.98

14. Showing an on-screen Smalltalk display during lectures was helpful in understanding the material.
    - 3.81
    - 3.86
    - 3.29
    - 3.61
    - 3.36
    - 3.38
    - 3.64

15. The live guest lectures were helpful in understanding the material.
    - 3.92
    - 3.83
    - 3.68
    - 3.60
    - 3.88
    - 3.76
    - 3.77

16. The videotaped guest lectures were helpful in understanding the material.
    - 3.75
    - 3.35
    - 3.54
    - 3.40
    - 3.69
    - 3.28
    - 3.49

17. It was helpful for the instructor to include in his lecture excerpts from the pre-taped Meyer lecture on Eiffel.
    - 4.00
    - 3.87
    - 3.97
    - 3.96
    - 3.91
    - 3.52
    - 3.87

18. It was (or would have been [whichever applies]) helpful to be working on a Smalltalk system at the same time you were viewing the lectures.
    - 3.61
    - 3.89
    - 3.80
    - 3.91
    - 3.93
    - 3.96
    - 3.83

19. This question for video students only: My educational experience would have been better if I had been able to take this course on campus.
    - 2.72
    - 3.38
    - 3.32
    - 3.15
    - 3.38
    - 2.89
    - 3.07

20. Was the pace of the course too fast or too slow? *(1 = way too slow, 2 = too slow, 3 = about right, 4 = too fast, 5 = way too fast.)*
    - 3.08
    - 3.21
    - 3.51
    - 3.16
    - 2.93
    - 3.27
    - 3.21

**Part IV. Usefulness** *(Same scale as Part III.)*

21. I was immediately able to apply what I had learned in the course to my work or studies outside the course.
    - 3.61
    - 3.89
    - 3.43
    - 3.06
    - 4.27
    - 3.76
    - 3.60

22. This course was more useful in my work (or studies) than most graduate courses I have taken.
    - 3.28
    - 3.89
    - 3.46
    - 3.27
    - 3.96
    - 3.88
    - 3.56

23. It was valuable to learn Smalltalk.
    - 3.95
    - 4.08
    - 4.22
    - 3.64
    - 4.21
    - 3.76
    - 3.96

24. It was valuable to learn Eiffel.
    - 3.09
    - 3.21
    - 3.20
    - 3.17
    - 2.61
    - 2.64
    - 3.03
<table>
<thead>
<tr>
<th>25. The course would have been more useful if C++ had been covered in greater depth.</th>
<th>3.44</th>
<th>3.51</th>
<th>4.00</th>
<th>3.69</th>
<th>3.07</th>
<th>3.96</th>
<th>3.66</th>
</tr>
</thead>
</table>

**Appendix A (cont.)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.34</td>
<td>3.69</td>
<td>3.28</td>
<td>3.50</td>
<td>3.46</td>
<td>3.64</td>
<td>3.46</td>
<td></td>
</tr>
</tbody>
</table>

| 27. The course would have been more useful if OOA/OOD had been covered in more depth. (Asked only since 1993.) | – | – | – | – | 4.04 | 4.22 | 4.14 |

| 28. The course would have been more useful if we had studied standard class libraries in more depth. (Asked only since 1993.) | – | – | – | – | 3.07 | 2.87 | 2.92 |

---

**Appendix B: Results of Managers’ Survey**

1. How many employees are working on your current o-o project(s)?
   - (a) Less than 5 ………… 7
   - (b) 6 – 25 ………… 26
   - (c) 26 – 50 ………… 3
   - (d) 51 – 100 ………… 7
   - (e) Over 100 ………… 8

2. How long has your company been using o-o technology?
   - (a) Less than 1 year …… 3
   - (b) 1 year – 2 years …… 16
   - (c) 3 years – 4 years …… 17
   - (d) 5 years – 6 years …… 5
   - (e) Over 6 years ………… 11

*Rate the next twelve statements on a scale of 1 (=strongly disagree) to 5 (=strongly agree):*

3. O-o technology has been of benefit to us by—
   - (a) Facilitating software reuse. ………… 3.56
   - (b) Making maintenance easier. ………… 3.76
   - (c) Decreasing development time (faster time-to-market) … 3.26
   - (d) Increasing software quality. ………… 3.58
   - (e) Allowing the development of more complex systems … 3.67

4. A roadblock we’ve faced in using o-o technology has been:
   - (a) Retraining employees. ………… 3.90
   - (b) Few qualified recent graduates ………… 3.56
   - (c) A lack of quality class libraries. ………… 3.44
   - (d) The poor quality of o-o development environments ………… 3.02
   - (e) Poor quality or an absence of needed tools ………… 3.31

5. What o-o analysis/design methodology would you like a recent graduate to know?
   - The Booch Method (Booch) ………… 24
   - Object Modeling Technique (Rumbaugh) ………… 21
   - Responsibility-Driven Design (Wirsfs-Brock) ………… 17
   - “Any is acceptable” ………… 13
   - OOSE (Jacobson) ………… 6
   - Fusion (Hewlett-Packard) ………… 4
   - Information Engineering with Objects (Texas Instruments) … 4
   - Class-Centered Modeling (Eric Adanow) ………… 3
   - Coad/Yourdon ………… 2
   - OO Software Engineering (Martin/Odell) ………… 2
   (Seven other methodologies were cited once each.)
### Appendix B (cont.)

**Rate the next nine statements on a scale of 1 (=strongly disagree) to 5 (=strongly agree):**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. I think it is more important for a recent graduate to understand the basics of o-o analysis and design than any particular methodology.</td>
<td>4.75</td>
</tr>
<tr>
<td>7. I would like a recent graduate to know the following o-o programming language(s):</td>
<td></td>
</tr>
<tr>
<td>(a) C++</td>
<td>4.41</td>
</tr>
<tr>
<td>(b) Smalltalk</td>
<td>4.10</td>
</tr>
<tr>
<td>(c) CLOS</td>
<td>2.09</td>
</tr>
<tr>
<td>(d) Objective C</td>
<td>2.33</td>
</tr>
<tr>
<td>(e) Eiffel</td>
<td>2.41</td>
</tr>
<tr>
<td>8. I think it is more important for a recent graduate to understand the basics of o-o programming than any particular language.</td>
<td>3.62</td>
</tr>
<tr>
<td>9. I think it is more important for a recent graduate to know an o-o analysis and design methodology than to know an o-o programming language.</td>
<td>3.04</td>
</tr>
</tbody>
</table>

**Answer the next two questions “yes” or “no”:**

<table>
<thead>
<tr>
<th>Question</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>10. Does your company have an organized program to encourage software reuse?</td>
<td>33</td>
<td>19</td>
</tr>
<tr>
<td>11. If you answered yes to #10, is the reuse:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Class libraries purchased from vendors?</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>(b) Class libraries developed in-house?</td>
<td>29</td>
<td>1</td>
</tr>
</tbody>
</table>

**Rate the next eleven statements on a scale of 1 (=strongly disagree) to 5 (=strongly agree):**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>12. After we hire them, recent graduates seem to need training in:</td>
<td></td>
</tr>
<tr>
<td>(a) O-o analysis/design methodology.</td>
<td>4.28</td>
</tr>
<tr>
<td>(b) O-o programming language(s) syntax and semantics.</td>
<td>3.73</td>
</tr>
<tr>
<td>(c) O-o programming language(s) environment.</td>
<td>4.04</td>
</tr>
<tr>
<td>(d) O-o class libraries.</td>
<td>4.18</td>
</tr>
<tr>
<td>13. The most effective type of &quot;after-hire&quot; o-o training we've used is:</td>
<td></td>
</tr>
<tr>
<td>(a) On-the-job (learn as you go, perhaps with a mentor)</td>
<td>4.04</td>
</tr>
<tr>
<td>(b) Computer based training/computer assisted instruction</td>
<td>2.49</td>
</tr>
<tr>
<td>(c) Videocourses (bought from a vendor or developed in-house).</td>
<td>2.06</td>
</tr>
<tr>
<td>(d) Lecture/seminar (stand-up lectures and/or seminars)</td>
<td>3.31</td>
</tr>
<tr>
<td>(e) Lecture/lab (stand-up lecture with hands-on lab).</td>
<td>4.24</td>
</tr>
</tbody>
</table>