Pervasiveness of a Programming Paradigm: Questions Concerning an Object-oriented Approach

MARK WOODMAN, SIMON HOLLAND, BLAINE PRICE
Computing Department,
Faculty of Mathematics and Computing,
The Open University, Milton Keynes

Email ids: m.woodman@open.ac.uk, s.holland@open.ac.uk, b.a.price@open.ac.uk

Abstract

This paper outlines the way in which a radical syllabus is being designed for the new introductory computing course being offered by the Open University from 1997. It describes how a decision to teach object-oriented programming has resulted in the associated concepts and paradigm pervading the syllabus. The result is a novel pedagogy by which students take considerable time to begin conventional programming. The context for this innovatory approach is a very large student population (3,500 per year), a long lead time for developing courses, and a need to remain current six or seven years after conception. The background and the emerging syllabus are both summarized and questions concerning the teaching of the object-oriented approach are raised.

1 Computing at the OU

The Open University (OU) offers its programmes of study in the U.K., the Republic of Ireland, and throughout continental Europe—with approximately 200,000 students studying with the university each year. The degree programme is modular and there are no named degrees. Very few constraints are imposed on how students may order or combine courses. This means that an introductory course may be taken by students wishing to specialise in computing, or by those who plan a much broader degree. Of the 2,500 students per year who complete the current introductory computing course nearly half will have chosen it for vocational reasons; typically they have studied a foundation course in technology or mathematics prior to the computing course. Predominantly they are male, and the number of female students taking the course is dropping. The new course must satisfy both students wishing to develop a computing profile and those who take it as a minor component in a general degree. It must also attempt to attract more female students and those students who might consider the course without either the mathematics or technology foundation courses. Anecdotal evidence indicates that female students find the course too male-biased and feel unusually reticent in tutorial groups because of the professional programming experience of male students (the average age of students is 34 and most are in employment).

Computing is studied at the OU in much the same way as other courses: students use a combination of specially written distance learning texts in combination with broadcast radio and television programmes designed for the course. In addition, students spend considerable time programming using personal computers. The new introductory course is worth 60 CAT points and will be need some thirty text units, four audio cassette programmes and sixteen television programmes; software development requires approximately ten person-years of effort. Therefore, a course team of nearly twenty academics and numerous programmers, designers, editors, BBC producers and project management staff need three years to produce all the material to the high standards of quality demanded by the institution and its students.

The current introductory computing course is extremely successful with 2,500 students per year finishing the course; but is now showing its age with an out-of-date programming environment, nothing in the way of modern HCI, and old-fashioned television programmes. It is a classical computer science course; it concentrates on data structures, programming and top-down

design, but includes units on software engineering, database systems, structured analysis and design, and the social impact of computing. What distinguishes the current course from those at other institutions in the style of presentation and tutorial support that are particular to the distance mode. The current course's message about computing is, implicitly, that it is about programming and about algorithms. In common with many similar courses it leads students to believe that a single design approach always leads to good solutions and that design is straightforward; i.e. the statement of some 'problem' can be refined stepwise, and without review or backtracking, towards an executable 'solution.'

OU students are relatively isolated from their peers and tutors, and this has a major impact on the syllabus and pedagogy for a particular course. Students may not have easy access to their colleagues and so often lack the informal support network which conventional students enjoy. Consequently, there is a high drop-out rate early in courses when students often become disillusioned by the difficulty of new ideas or by the general ethos of the course which may not be what they expected. It is therefore imperative to ensure that students 'survive' the first part of a course in which a new concepts are being taught. The first assignment in continuous assessment (which counts for 50% of the final course score) is a watershed in this context. Moreover, internal studies have provided evidence that students are directed by the content of their continuous assessment questions, to the extent that they spend much longer doing them than course teams advise. Students also use assignment to determine what to study, to pace themselves and to assess their own progress.

The current combination of students using goal-directed learning and academics teaching a goal-oriented design and programming method appears to work against the development of strong analytical and abstraction skills. Even students who have previous professional programming experience do not properly deepen their knowledge sufficiently for more advanced courses on formal methods, database systems, etc.

Given the factors above, the new course needs to have at least the following characteristics:

1. It must give high priority to analysis and abstraction.

2. The design method must exploit notions of encapsulation and abstraction and must encourage students to reflect on choices they have to make.

3. Early weeks of work must simultaneously be challenging, be encouraging and hold the interest of students.

4. Novelty of content and delivery should be used to ensure course longevity and to establish a 'level playing field' for all students.

5. The programming language must at very least support abstract data types and must facilitate programming modern graphical user interfaces.

6. Whatever software environments students might need must exhibit the concepts taught and must provide support not only for the programming language chose but for visualisation and comprehension tools.

After much discussion the course team has concluded that object-oriented technology is most likely to be the best basis for meeting most of the above requirements. However, this conclusion has not been whole-heartedly endorsed by all of the team for a variety of reasons, one of which is the degree to which any object-oriented approach will force us to adopt the whole object-oriented 'paradigm'—the whole culture and rhetoric of object-oriented development. (Cook (1994) analyses a variety of cases and discusses the appropriateness of the object-oriented paradigm.) Even those team members who promote the importance of object-oriented concepts remain dubious about which ideas are sufficiently mature to be relied upon and to be adapted for introductory programming.

In the sections which follow we explore how our decision to adopt an object-oriented approach has resulted in the related concepts dominating our syllabus considerations and has led us to a different emphasis than is usual for the object-oriented paradigm. We first describe the basic principles of the new course—how, in particular, we have diverged from the classical approach. Second we record how an object-oriented syllabus evolved and report our views that it must be present in an introductory course. Third, we discuss three possible pedagogical strategies for the object-oriented paradigm. Finally we outline requirements for computer-based support for object-oriented programming. We demonstrate by this reflection on our

processes that a decision to adopt an object-oriented paradigm has pervaded the design of the new syllabus.

2 A New Emphasis for Introductory Computing

For the new course we are taking a different stance from the data 'structures + algorithms' philosophy of the current introductory course, but have not yet finally decided on a suitable title that encapsulates the new philosophy (the course is known by its code, M206). The main message about computing in the new course is that software has structure and is constructed from components. The course aims to deliver the following to students:

1. knowledge of computing concepts and a vocabulary for discussing them,

2. analysis and programming skills,

3. a framework for the theory and discipline of computing.

The primary outcome of study for a student will be her ability to recognize what a computing artefact is—in terms of its use and construction. Having completed the course a student should be able to analyse a piece of software, to understand its use and usage, and to understand from that analysis something about the design choices made. This understanding will necessarily come from experience designing and implementing programs. Before gaining this experience, we want students to have an adequate and accurate language for describing software.

This theme of language—the *language of software*—comprises a rich vocabulary for describing and analysing software and provides a 'rhythm' for the course. The theme is introduced in early and can be re-visited at intervals through the course as more sophisticated understanding is developed, e.g. starting from a user's view (what does application X do?), moving to a constructor's view (how is application X structured), then becoming more technically sophisticated and incorporating more and more implications for software design (does X do what is does effectively or can a better design be developed?).

The course is not primarily intended to train programmers or even to teach programming *per se*, but to teach that programming is the building of

structured artefacts in order to accomplish a goal or serve some purpose, like solving a problem. This perspective will underpin the learning of software development skills for those who will eventually become programmers. The course intends to convey a view of computing and software development at all scales—from what may be termed programming in the small to programming in the large—even though students cannot be expected to originate much code of truly large programs. However, we do fully expect them to be able to construct large programs from library components supplied with a programming system or by the course team.

Thus the course will teach people about computing in part through programming. However the course will also teach people about computing by having them read, analyse, de-bug, modify, and test programs. Most importantly, students will be asked to reflect on their work and the processes they have used. Hence the course will foster a broad understanding of practical computing and will choose particular concepts to study in depth—for example, notions of structure, abstraction, and component-based construction.

The theme of construction from components provides an approach to programming at all scales, which can be used to introduce different styles if reasoning and construction, for example presenting procedural and object-oriented techniques as different component construction 'paradigms.' (We outline how these are to be combined later.)

In large measure, therefore, the new course is defined by the combination, of its themes and the skills it sets out to develop:

- abstraction;
- the language of software;
- software has structure;
- software development as a rational engineering process;
- software has dynamic, predictable behaviour but this is defined by static text or graphics;
- software uses a multiplicity of views;
- components are significant software structures;

- analytical skills;
- debugging;
- reading programs;
- reasoning about programs and program behaviour.

We next describe how various object-oriented approaches were considered and some of the constraints that required a compromise.

3 The Development of an Object-oriented Strategy

The development of any strategy has as much to do with the interaction of the people involved (and the effectiveness of an individual argument on a given day) as it has to do with any objective view of its value. In the case of an OU course team, some twenty academics may bring forward proposals with strong scholarly reasons for adopting one pedagogical approach or another. While strong arguments in favour of teaching object-oriented programming in an introductory course have been published there is no evidence that the 'face-to-face' techniques used at, for example, Carleton University (Lalonde and Pugh, 1990) can be easily translated to the distance mode. A number of factors need to be addressed—both those peculiar to the Open University and general to computing academia. We sketch two of these below and briefly describe how we sought a compromise.

3.1 The Need for Longevity

The significant investment needed when producing a course demanding the resources mentioned earlier means that OU courses must have a life of between four and six years. M206 will not be reviewed for replacement until 2001; this means that it must be for current for at least six or seven years after its conception. Hence we have had to examine what changes are in train in conventional universities, what students want, what changes are perceptible in he computing industry and how various experts are expecting software development to evolve in the next decade. Much crystal ball gazing is therefore needed.

Put crudely, in terms of programming languages, what we see in conventional universities is a relatively slow movement towards object-oriented programming. For the most part this means students beginning to program abstract data types, with the likes of C++ (e.g. Lee and Stroud, 1994), and moving on to more advanced object concepts later. We have become convinced that the trend toward teaching object-oriented concepts is a reflection of the increase if their importance to the industry, where the increase in the use of object technology is evident.

However, the medium-to-long term importance of emerging technologies is of greater importance to our planning. The continuing industry preoccupation with maintenance and reverse engineering and recent reports of development with reuse yielding significant increases in productivity and quality has led us to consider a component-based perspective to be extremely important. While components are not synonymous with objects, the latter are effective realisations of the component concept.

For the above reasons, allowing object-oriented concepts to pervade the course does seem to assist our particular problems of longevity.

3.2 Attitudes to Object Technology

Attitudes on the importance of object technology can be conveniently approximated by three simplified positions:

(a) It's another fad and will pass,

(b) It's just another modularity technique that can be adopted piecemeal where appropriate

(c) It's a true paradigm shift, and will eventually pervade most of computing (Cook, 1994).

It is not the aim of this paper to argue for or against any of these positions; for some of the arguments see Udell, 1994; Kay, 1993; Love, 1993; Pope, 1994, and Cook 1994. However, it is important to accept the existence of the different positions. (Note that positions 2 and 3 may be entirely compatible. Holders of a new belief often have to start off being critical of their opposition until their belief becomes established, at which point they can afford to be generous.)

Many Computer Science departments need not confront such problems yet, since they can keep an eye on developments and adapt their teaching a little year by year, (assuming that patterns of mutually dependent courses do not render any necessary change impractical). Gradual change is not an option for us, for reasons discussed earlier, since the course must be written very soon based on decisions taken now, to run more or less unchanged from 1997 to 2001–3. Hence we have been forced to explore the pedagogic implications of possible positions on the object paradigm now.

The course team did not collectively agree on any of the three positions noted above, but fortunately, were easily able to agree that object-orientedness should be a central element of the teaching strategy, and that object-oriented concepts should be taught well and clearly. However, having made this decision, the course team was split between apparently incompatible teaching strategies on the best way to achieve this aim.

3.3 Seeking a compromise

An incremental and evolutionary approach was taken to developing our ideas. When it became clear that there was not a consenus on a whole-hearted concentration on object-oriented programming (for example teaching Smalltalk from the outset) we investigated three strategies for meeting the requirements stated above. These strategies concentrated on the beginning of the course where motivation and retention is so crucial. One strategy produced ideas on creating software artefacts without conventional text programming—using recording and scripting systems for graphical user interfaces. Another argued that concentrating early on near-expert use of a programming environment, and understanding conditional and repetitive structures was essential for the type of course we wanted. The third tried to introduce object-oriented ideas from the beginning using a suitable object-oriented language (not C++) but spending considerable time exploring and analysing carefully constructed object-oriented systems before becoming involved.

While none of these strategies, which are described in the next section, fully satisfied our course team, they did expose strong support for a software component orientation in the course and confirmed the commitment of all discussants to developing students' analysis skills and to improving their ability to abstract. Furthermore, the idea of not 'programming' first (i.e.

avoiding a textual language) was thought to be worth taking further and now forms an essential part of the course team's pedagogy.

The incremental approach has proved beneficial, not just because it has allowed sufficient time for reflection and informal debate. It has allowed us to extrapolate from certain 'obvious' approaches which we found to be flawed. For example, it was estimated that it would take over half the course study time to proceed from the traditional starting point of language learning (expressions and statements) to using object-oriented facilities inheritance and polymorphism; this appears to be consistent with the decision of colleagues in other institutions to postpone these ideas to later courses; see Lee and Stroud, 1994.

Next we review the alternatives considered so as to show how object-orientedness pervades a syllabus whatever strategy is considered desirable.

4 Three Pedagogical Strategies for Objects

We will now review three strategies which are appropriate to the requirements given earlier; we examine some of the advantages and limitations of each, their apparent incompatibilities, and the reasons why they finally all had to be rejected in favour of a newly synthesised approach.

Note that several points are related to the types of interfaces and environments that students will be used to prior to embarking on the course and during the course; we believe that students must be able to write programs whose interfaces are similar to those that they are used to.

A significant difficulty is the development of an explicit strategy that students can employ to derive designs from natural language or graphical specifications. This problem is related to the programming language chosen; a pure object-oriented language would allow a straightforward conceptual model of computation to be used; a hybrid language would require that the imperative–store model coexist with an object-oriented model. A further pedagogic consideration is the use of algorithm design. Traditional approaches use algorithm design early in order to develop non-trivial applications. A programming method, language and environment that facilitates early development of non-trivial applications is likely to change, or even diminish, the role of algorithm design.

4.1 Pedagogic Strategy 1: A 'Pure objects' Approach

This is in some senses the simplest pedagogical strategy of the three. It may be seen as growing partly out of the classic 'Boxes' approach pioneered by Adele Goldberg and Alan Kay (1977). The key points of this approach are as follows:

1 Treat objects and messages as fundamental.

2 Use a pure object-oriented language (such as Smalltalk).

3 Hide the full power of a programming environment initially and work in a simplified but progressively disclosed teaching environment (e.g. only allow access to a few classes).

4 Introduce the object model and object terminology as the basic way of describing computation.

5 Teach decomposition of a computation into objects as fundamental.

We will note the initially perceived advantages and disadvantages for each of the pedagogic strategies. Kay (1993) argues that these advantages are no accident. His argument is that when decomposing a computation (or anything else), it is better from if the decomposition is recursive—from the points of view of mathematics, design, aesthetics, systems theory and engineering principles. Kay maintains that decomposing a computation into inert data and separate processing algorithm immediately violates this principle since neither component can compute anything alone. This violation leads to complexity and problems in conventional methodologies when decomposing these components further independently, which after all clearly affect each other. On the other hand, decomposing a computation into lots of 'little' active computational units—objects—meets the requirement for good recursive decomposition.

Perceived advantages of an 'objects first' approach

- *Conceptual simplicity:* a very small number of concepts are required compared with a traditional approach (but see Jones, 1994 for a contrasting view).

- *Intellectual power:* the traditional imperative–store approach may be viewed as a conceptually simple subset of a pure object oriented approach—the reverse is not true.

- *Concrete support for abstraction:* this approach is excellent for teaching Abstract Data types and good principles of modularity—since ADTs are concretely supported.

- *Engineering merits* : excellent support is provided for good software engineering and modularity.

- *Student motivation*: this approach has good re-use properties allow students to build substantial programs very early.

- *Good environments:* excellent and easy-to-alter graphical environments are available (at least for Smalltalk).

- *Avoiding bad habits:* avoidance of 'C++ syndrome' where students write 'object-oriented programs' with just one class.

One interesting claimed advantage is that programmers who learn a traditional imperative–store approach first appear to have more have difficulty absorbing an object-oriented outlook than complete beginners. The converse does not appear to be the case.

Perceived disadvantages of an 'objects first' approach

- general distrust of extreme rapid change;

- perceived lack of teaching resources;

- belief that objects as a concept are in some way too abstract;

- scarcity of experience in universities of teaching objects first (though see Lalonde and Pugh, 1990);

- fears of effects on other courses;

- fears about the need to make radical modifications to the programming environment;

- dislike of semantic aspects of object-oriented languages (such as the semantics of expressions in Smalltalk).

4.2 Pedagogic Strategy 2: An 'Objects late' approach

The second pedagogical strategy is more complex, but much easier to describe because of its strong traditional (structured programming) component. The essence of this approach is as follows.

1 Use a hybrid language, such as Ada 9X, C++, Oberon-2, etc.

2 Teach traditionally, working up from expressions and statement to the concept of an abstract data type (ADT).

3 Finally introduce the concept of object as a development of the ADT.

Perceived advantages of the 'objects late' approach

- builds on existing staff strengths;
- safe, evolutionary change.

Perceived disadvantages of the 'objects late' approach

Conceptually this approach is potentially complex for the beginning student: beginners must be exposed in one course to a series of diverse, subtly differing, and partially redundant concepts. For example:

- both reference and value semantics for assignment;
- both basic data types and objects (and their type incompatibilities);
- both polymorphism and overloading;
- in many candidate languages the use of pointers for references is too apparent (and explicit dereferencing is needed);
- in many candidate languages, no explicit garbage collection is provided.

In general, having a virtual machine that must function both with, and entirely without objects leads to apparent inconsistencies, redundancies and overloading of notation. Furthermore, the concept of abstract data types becomes difficult and abstruse, rather than immediate and concrete. In short, even though a hybrid language be a practical choice, students are likely to be confused by the time they reach the object concepts, violating the decision to teach objects well.

4.3 Pedagogic Strategy 3: A 'Programming last' approach

This was the third and most radical strategy which was explored. Consequently we give some extra detail.

Traditionally, introductory computing courses have begun with programming: they introduce the student to the syntax of some simple statements in the chosen programming language (such as a "`print`" statement) and then they show the student how to execute the program with some simple data (such as "`print 'hello, world'`"). The students then add more syntax to their vocabulary and progress to using iteration and selection in other simple programs. Students with little background in computing will fail to see the practical value of tiny programs that print their name and the number of days until their next birthday while students who have programmed before will be bored with such tasks.

This strategy is to teach computing by beginning without teaching traditional low-level programming. Instead, start by formalizing the language of software:

1 We begin by working with everyday software which the student will already be familiar with: a word processor, a spreadsheet, and a drawing application.

2 We then formalize the students' understanding of modern software interfaces by identifying the name and functionality of various visual components common to each of the applications.

3 Examine the units which each application works with (e.g. characters, words, lines, paragraphs, cells, lines, objects, etc.). The student is then given a number of problems and asked which application is most appropriate for a given problem. All these exercises build the vocabulary needed.

From here we progress to communicating between applications, first using the simple manual 'cut and paste' mechanism, then using a script recorder to automatically record a script by demonstration to automate the process. Subsequently we bring in an interface building tool kit and have the student link this new script to a button on their desktop. The students may not realize it at this point, but they will have just written their first self contained program.

Now we have the student begin to work with the script-writing environment to examine and modify the script that they have just written. They will then create practical scripts to manipulate data between the applications and learn some of the basics of debugging, iteration, and selection along the way as well as make use of the interface building tool kit to create stand-alone scripts. By the end of this first section of the course the students will have the same type of programming skills as in a traditional course, but they will have achieved this by using everyday software and practical problems as opposed to an abstract language and toy problems. Some of the students may not even realize at this point that they have been programming.

Perceived advantages of the 'programming last' approach

- creates a 'level playing field' for students;
- establishes the language of software fairly naturally;
- encourages weaker students without boring the stronger ones;
- develops analysis skills early to facilitate relating the software they write to the software they use everyday.

Perceived disadvantages of the 'programming last' approach

- lack of experience teaching using this style;
- fears that course will be seen as a computer literacy course.

This strategy met with almost universal approval as a way of introducing the course, although it was not clear how later parts of a course based on such an approach might proceed. In the next section we explain how these strategies

were used to synthesize an object-oriented approach where programming comes late, if not exactly 'last.'

5 Synthesis of a New Teaching Strategy

Given the lack of agreement on the first two approaches, and given that the third approach applied chiefly to the beginning of the course, it was decided to attempt to synthesise a new approach that met the constraint of teaching object concepts clearly and well, as well as all of the other constraints described earlier, combining the best features of each approach were possible.

The teaching sequence we have devised can be described as falling into four (unequal) parts:

(a) The Everyday Software Approach.

(b) Elementary Programming and The Software Development System.

(c) The Core.

(d) Wider aspects.

The 'everyday software approach' allows the third strategy for objects. It is intended to establish a vocabulary and develop analytical skills so that students are equipped to deal with later concepts. Students will be set specific goals on the use of software and on its manipulation, rather than on its construction from a low-level of abstraction. In practical terms it involves running, describing, and analysing everyday software applications—such as word processors, database systems and graphics editors—and using combinations of them and getting them to communicate (even remotely).

During this first part, which is at the heart of the synthesis, students will encounter object-oriented ideas. These will range from describing the launching of applications and the opening of documents in terms of messages being sent to objects. Early in this part will introduce the student to the concept of communication between computing components (applications); later we build on this understanding as we introduce communication between objects in an object-oriented programming language. More subtly, ideas of polymorphism and inheritance can be introduced—e.g. sending the message "open" to different objects.

The second part of the course requires exposure to low-level programming concepts before proceeding to core ideas about constructing programs. Moreover, it does require an exploratory approach and demands that students continue to use the analytical skills developed earlier; we intend that they learn about their programming system by guided exploration and analysis. We also intend to teach certain skills explicitly: those of learning elementary language constructs, analysing and de-bugging programs.

The 'core' of the course, in outline, looks like a conventional programming course insofar as a student encounters expressions, statements, etc. Objects will effect this part in several ways. First the programming language, will support a component-oriented and object-oriented view of software. Smalltalk and Eiffel are actively being considered, but pragmatic factors may lead to a choice of Ada-9X, object-oriented Modula-2, Oberon-2 or Object-oriented Turing. In many ways the language choice is less important than the availability of an environment that will support a student in the distance mode; we foresee an increased need for program analysis and visualization tools to help students reason about object-oriented programs.

The intention of the final part of the course is to teach social issues, ethics, etc. as part of the practical work on constructing software; for example, programming a simulator for a nuclear control system or a patient monitoring system will be more effective in conveying the relevant issues. (The reason for separately identifying this part is that it allows for the possibility of frequent (in OU terms) replacement of the wider aspects material.)

Much of the above will require significant technological support, the cost of much of which will be borne by students. We briefly explore our new approaches in this area.

6 Innovations in Course Delivery

The traditional method of delivering an OU course is via a custom designed textbooks with accompanying audio–visual material. In M206 we will also have the opportunity to innovate on our delivery method by delivering course material in a hypertext multimedia application. This means that static diagrams of computing abstractions in the text can be animated when necessary using software visualizations to show dynamic execution over time. Another advantage of electronic delivery of the teaching material is the

availability of automatic translation into other forms, such as spoken word translation of text for visually impaired students.

In the first section of the course we will examine communication between the application and the user by formalizing our description of the interface. We also examine communication between applications; the next obvious step is communication between computers, which leads into a look at the Internet. Students will use e-mail and electronic conference (in addition to face-to-face tutorials) for communicating with their tutor. They will expand their network use from here by using modern network navigation tools to search hypertext multimedia databases around the world, thus developing a valuable computer literacy skill for the next decade. Students will also look at the social implications of very large interlinked global networks in terms of privacy, nuisance, and personal safety.

The theme of understanding that programs are dynamic with predictable behaviour will contribute directly to an understanding of testing, in which the student must predict the expected behaviour of program—must in effect simulate its behaviour in order to produce test data. Dynamic visualization is pertinent to this theme.

However, these innovations require significant processing capacity. At present computing students need only purchase a simple PC and printer. Current courses only need floppy drives and 512k of memory. We are now advising students that by 1997 we will be assuming that they own, or have access to, a computer which has the power of a fast Intel 486 machine with at least 8 Mb of memory, 250 Mb hard disk and a high-speed modem. It is also likely that large volumes of on-line information will be assumed so a CD-ROM drive may also be a requirement. The programming language and its development environment will be supplied to students; depending on the cost of CD-ROM drives and potential savings to the university in terms of print costs, these devices may also be supplied to students.

It is expected that considerable use of remote facilities will be needed by the syllabus. However, we are expecting that ideas about an electronic campus (using conferencing, electronic mail, on-line information services, etc.) will be coming to fruition during the lifetime of M206 and so considerable effort will be put into integrating these technologies.

Given the power of the type of computer system students will have, we are also planning computer-assisted learning systems for the course and developments in this area are in progress.

7 Conclusion and Discussion

We have outlined a teaching approach for beginners which allows object concepts to be tackled early, but which does not neglect more traditional views of computing. The difficulties in designing the curriculum for this course lie in the diversity of the students' background and the breadth of the goals of the course: we must accept all students, including those who may have no background in mathematics or computing, and when they leave the course they must have skills which they can take to other courses in the sciences and humanities as well as sufficient preparation to take more advanced computing courses if they should choose to do so. We will have succeeded in our goals if some students who take the course as a one-off advanced literacy course later decide to study more advanced computing courses *and* if we retain those students who originally took the course as a preparation for further study in computing.

As we analysed the various strategies, it became clear that the approach of initially exposing students to a set of applications, while building up a language for describing and specifying software applications more abstractly, provided a good means of laying the grounds for a very concrete, and not at all abstract notion of an object. (Indeed, this partly recapitulates the development of the object concept as traced in Kay, 1993). An application could be discussed with beginners as having state and responding to various commands via a well defined interface. This could lead very naturally, if informally, to simple notions of objects, messages, interface, state and encapsulation. Clearly, such an approach is not too conceptually abstract for beginners.

Objects 'late' may be too late

A second realisation came when it became apparent that with the '*objects late*' approach, even if the traditional imperative–store part of the course was cut to the bone (just one type of loop, etc.), it would be impossible to reach the object part of the course until near the end, when students were tired and unlikely to take in the new ideas as well as required.

Terminological exploration

Object advocates on the team had initially found it hard to persuade object sceptics that traditional concepts were relatively easily described in object terminology, but not vice versa. The following example seemed to establish this point, although it is important that it should be treated as suggestive rather than rigorous. Consider the problem of using a pure object language, such as Smalltalk, to teach a traditional imperative–store approach. Bearing in mind that Smalltalk has explicit assignment, unlike, say, Self (Chang and Ungar, 1993), it turns out that to a good order of approximation, this requires no new constructs, only optional restrictions, as follows:

- Restrict attention to a single class.
- Restrict attention to a single object of that class.
- Use bare minimum rest of class hierarchy e.g. integers, strings, arrays etc. as a sort of function library.
- Designate one method as the main program.
- Use no instance variables (unless global variables are desired).
- Deal only with objects with state ('basic data types').

Similarly, if we wish to describe various phenomena in a traditional imperative–store approach using object terminology, it turns out that this is not too hard: one message understood by a typical Pascal program is "execute."

By contrast, it turns out that simulating a full object oriented language using only a traditional language with no object facilities requires construction rather than subtraction: namely, writing an object interpreter. Similarly, describing the full range of object phenomena using imperative–store terminology alone tends to become convoluted.

For this level of course, detailed method may not matter

When analysing typical problems set to students in traditional courses, we realised that they are typically very leading. For example, what purported to be a real-world question about solving a problem to do with devising

questionnaires for establishing passengers preferences about buses was really a leading question inviting the student simply to nest a conditional within a loop. At this level of problem, whether using a traditional or object oriented approach, precise software development methods matter less than any reasonable approach that helps the student to learn and solve problems or create models.

Realisations such as these allowed us to synthesise a new approach with key points as follows:

Objects and imperative–store concepts in parallel

- Use a 'programming last' or component-based approach to introduce basic concepts of objects and interaction between objects.
- Introduce objects early, much as outlined in the 'pure objects' approach (not necessarily using a pure object language).
- Treat objects and messages as fundamental concepts, but take time over the 'message fragments' stage.
- Deal clearly with sequence, iteration and conditionals as ways of structuring these fragments.
- Introduce concepts relevant to both an 'objects early' and a traditional approach side by side.

We believe that this approach combines the following benefits.

- *Conceptual simplicity:* a smaller number of concepts are required (compared with an 'objects late' approach), to introduce concepts from both object and imperative–store paradigms.
- *Concrete support for abstraction:* ADTs will be concretely supported early, unlike in an 'objects late' approach.
- *Student motivation*: Students will be enabled to build substantial programs early, by exploiting object-based re-use techniques.
- *Avoiding bad habits:* avoidance of 'C++ syndrome' described earlier, where students write 'object-oriented programs' with just one class.

At the same time, this approach meets pragmatic organisational needs by building on existing staff strengths, and represents a path of relatively safe evolutionary change.

Remaining problems

While the course team continues to work on the development of concrete teaching sequences, and on the choice of a final language and environment, in one sense the contradictions requiring the original synthesis are still present. For example, use of a particular language can import a lot of conceptual baggage. Choice of a pure object oriented language such Smalltalk would tend to lead to a result closely related to the original 'objects first approach'; while choice of C++, ADA 9X, etc. might force a relatively cluttered approach in terms of the number of concepts forced on the student's attention. A comparatively pure object-oriented language such as Eiffel that looks similar to Pascal on the surface might be a good compromise candidate if it were not for the current lack of good student environments available for Eiffel.

References

Udell, John (1994) Componentware. In Byte, May 1994, page 46-56.

Pope, Stephen Trav[illegible] (1994) [illegible] Open University Video for postgraduate course M868: [illegible]ject-[illegible]nted [illegible]oftware Technology, Band 4, Walton Hall Milton Keyn[illegible]

Kay, Alan (1993) The Early History of Smalltalk, in ACM SIGPLAN Notices, Volume 28, No 3 March 1993

Lalonde, Wilf and Pugh, John (1990) 'Smalltalk as the first programming language' in the Journal of Object Oriented programming, December, 1990.

Love, T. (1993) Object Lessons. SIGS Books , New York.

Cook, S. (1994) Analysis, Design, Programming: What's the difference?. In Teaching and Training in the Technology of Objects (proceeding of TATOO, 1994).

Goldberg, Adele. and Kay, Alan C. (1977a) Methods for teaching the programming Language Smalltalk, Xerox Palo Alto Research Centre , June 1977.

Goldberg, Adele. and Kay, Alan C. (1977b) Smalltalk in the classroom, Xerox Palo Alto Research Centre , June 1977.

Chang, B and Ungar, D (1993) Experiencing Self Objects: an object -based artificial reality. Technical Report, Sun MicroSystems.

Jones, A, (1994) Smalltalk – an educator's dream? University of Wales, College of Cardiff.

Lee, P.A. and Stroud, R.J., (1994). C++ As An Initial Programming Language, in M. Woodman, Programming Languages: Experience and Practice, Chapman and Hall, London (in press).