Software Engineering Education
The Shift to Object Oriented Programming

Paulo Bianchi Franca [*]

International Technological University and
Universidade Federal do Rio de Janeiro

Abstract:
This paper discusses the introduction of the Object Oriented Programming paradigm in Software Engineering Education. Although the subject has being growing in importance over the past few years, its inclusion among the software engineer assets is still very limited.

It is suggested that teaching of OOP be included as early as possible in undergraduate and graduate curricula as opposed to the current situation of leaving the subject to advanced courses; Issues related to re-educating experienced programmers, including faculty, are also presented.

Keywords: Object Oriented Programming, C + +,
Education

Introduction

In the author's view, the teaching of Object Oriented Programming is a case of massive effort; beginner students must face this approach from their early programming days and live comfortably with it through their carrers; Experienced professionals should, on the other hand, be given adequate opportunities to understand and use OOP.

This paper results from the author's experience while participating in several activities related to the teaching of OOP while acting as faculty at Santa Clara University in Silicon Valley. These activities include: graduate and undergraduate classes, industry meetings, textbook and courseware preparation and technical seminars.

* Current Address: Computer and Software Engineering Dept.
4701 Patrick Henry Dr. Building 17, Santa Clara, California, 95054
E-mail address: Pfranca@vms1.nce.ufrj.br

Motivation:

Some two decades ago, programmers were shaken with the advent of structured programming. This new technique imposed several restrictions on the way programs should be written in order to produce cleaner, more reliable and readable code, also leading, hopefully, to higher productivity, and lower maintenace costs.

In a few years, structured programming became the only way of programming. Even the syntax of the most popular languages was changed to enforce the structured technique. In fact, the younger generation of readers may even have some difficulty understanding this paragraph since they never knew what was on before structured programming.

Structured programming had a very strong allied at that time: All good programmers were already using some of its principles even without knowing it. As these principles were formulated, they all could recognize their value and became active preachers to bring the light to their fellow "spaghetti code" programmers.

The Object Oriented Programming (OOP) approach is playing a very similar role; its promises are: much cleaner and readable code, higher productivity and lower maintenance costs. Essentially, these are the same promises made by Structured Programming; however, in some respects OOP reaches for broader results mostly because of its extensive support for reuse.

Unfortunately, unlike what happened to Structured Programming in its early days, the novel ideas behind OOP have not been in use by good programmers and the reason can be easily understood: You could write a structured program in a non-structured language by just adhering to some principles, later modifications in the compilers merely reinforced the approach. Nevertheless, implementing objects that can be encapsulated, inherited, used polymorphically and, finally, reused, would be a very difficult thing for a programmer to even imagine without having a special compiler.

It is also true that reuse started with the invention of the subroutine library.

If we all agree that OOP is important in fullfilling its promises to a higher extent than Structured Programming did, we must understand that this technique must be mastered by all programmers and not just a few. This makes the subject a case of mass education. Benefits of software reuse and OOP are supported by several authors [1,2,3]; however educational requirements to achieve the desirable changes have not been sufficiently exploited. Of course there are many others who do not attach that importance to OOP and we shall not use this space to argue.

Requirements:

There are a few points that, currently, need enhancement in sofware engineering education. The list below was informally extracted from software industry meetings[4] dealing with the subject. The points outlined below are generally present in our teaching environment and inconsistent with the real professional environment; any changes that, at least, preserve the quality of teaching while bringing students closer to the real enviroment can be seen as an improvement.

The software development as seen in most classes:

- start from scratch
  real life programs seldom start from scratch; in fact, professional programmers spend most of their professional lives changing, adapting and fixing existing code.
- lack specifications
  real life programs must obey very strict specifications; they must use or fit into some preexisting software.
- lack group work
  Students are, often, requested to work by themselves; even when workgroup is allowed, still many prefer to work alone. This avoids them the problem of scheduling meetings, compromising on design views and establishing well defined interfaces; those are interesting issues for their professional life.
- do not need maintenance
  Most of the time, once the homework is done, the students may just as well dispose of that code for they never intend to see it again; How can one see the need for maintenance then? Maintenance costs are know to be very high and real life software must be developed with maintenance costs in mind. Our students are not motivated enough even to document and comment their programs.
- have no reuse
  In real life, reuse is very badly desirable and is strongly affected by the programmer's habits and discipline. Strong management action is often needed to enforce reuse because programmers tend to redevelop and, eventually recycle (rather than reuse). Although we might want to do more research on this, I think we can agree that the more our programmers have practiced reuse, the less management action will be required.
- lack of customers approval
  In real life there is a customer that either likes or dislikes the software. Factors contributing to that go far beyond code efficiency, and beauty of design; possibly the issues of interfaces, ease of use, documentation etc. are far more important.

Among the topics listed above, we can point out reuse as a mandatory requirement if we intend to teach object oriented programming; there would not be much point on OOP otherwise. In addition, we can also agree that reuse, maintenance and modifying existing code (non-scratch) are strongly related and the more the students can understand these issues, the more, hopefully, they will be adepts of reuse.

Programmers like to write code, and one of the main adversaries of reuse is the fact that programmers rather rewrite code than to read specifications on somebodyelse's code. They may argue that it may take them less time to rewrite a new code than to learn how to use an existing one. It may happen that, even including the ever underestimated debugging time, this turns out to be true. Nevertheless, this approach completely neglects maintenance problems that will only be show in the future. At that time, several slightly different modules may have to be fixed at an unknown cost.

Undergraduate programs:

Inclusion of OOP in undergraduate curricula has a few alternatives:

- Advanced courses on Senior years
- intermediate courses
- introductory courses
- integrated courses

Advanced courses may have been used to introduce OOP in many leading schools. The main disadvantages are:

- Students are already used to program in the plain structured way; they have to change their mentality.
- Students may leave school thinking that OOP is just "one more tool" they can use, we fail to make them understand that this is "the way" they should develop software.

On the other hand there are some advantages:

- Students have more maturity to understand the issues of productivity, maintenance costs, reusability, etc.

- Students will not be disappointed or confused to see other instructors lack of usage of OOP (since they are about done with school when they can notice that).

By intermediate courses we mean any course offered during the first couple or years where students have a prerequisite of knowing programming already. Some particular interesting cases are:

- Data Structures is a particularly interesting course for students to appreciate OOP. If care is taken to have them reuse (and consequently write accordingly) their code, they will easily become addicts.
- A course in a second language, such as C + + can also succeed; however, since students have to spend most of their attention to the new syntactical aspects of the language, some of the appreciation for OOP may be left unnoticed.

Main disadvantages:

- Students just learned how to program (without objects) and have already to update some of their concepts.
- If the approach of teaching a second language is taken, and this language is not extensively used in the remaining of the course, students may never try to use OOP again!

Advantage:

- Data structures is an ideal course for OOP appreciation. This would be even better if the students knew OOP and use it during the data structures class.

Introductory courses exposing OOP as soon as students face their first programming class are very promising. We may expect the students to build their programs using objects more naturally if they start really early. In fact, many students may have already some exposure to programming from their time in high school; but this rather enforces our reasoning: we don't want to teach them the same stuff they learn in high school, so why not use objects then?

Although teaching at this point has several advantages, we must, however, be aware of some practical disadvantages:

- Lack of educational material
  currently, there is a very restricted supply of textbooks for beginner programmers using OOP (I am trying to contribute to change this).
- Language complexity
  If students had trouble enough to understand simple languages like basic, what should happen with a language that supports OOP?
- Later disappointment
  If students learn OOP right at the beginning and most other instructors don't even know what OOP is for, students may be seriously disappointed.

Integrated course sequence is, of course, the long range alternative. This would probably mean that the introductory course would teach most of the basic concepts, the data structure class would reinforce them and most other courses involving programming would expect the students to use OOP. If, at the very least, the Object Oriented Programming language tought in the introductory class remains as the dominant language throughout he course, this task will be easier. The main problem one may be facing is the education of faculty which will be discussed later on.

Suggested actions:

An interesting problem that has to be solved is the choice of the OOP language to teach. Since we are restricting this discussion to education of Software Engineers, I feel tempted to adopt C + + without further discussions due to its popularity in the software industry and its good support for objects. In fact, if we want to teach a language that happens to be widely used and known in the production environment, there is no reason to chose anything else except, of course, for the difficulty to teach it...

Once the will to incorporate OOP is present, clearly the integrated sequence is the desirable choice. However, this may not be achieved very quickly due to interaction with most every other course in the curriculum; also, faculty, as a whole, may not be able to respond to this change fast enough. If, at the very least, students are exposed to objects in "introduction to programming"and "introduction to datastructures", we may expect them to contribute to this change. The importance of the introductory course cannot be underestimated. A discussion of some desirable features and suggestions on how to implement this course are included below.

The introductory course has a key role in this change. It is at this point that students will be presented with the concepts of programming with objects and will learn the language that will be dominant in their trainning afterwards. Needless to say, this course bears a great responsibility; it is then that students really decide whether they want to become software engineers or not. The purpose is to teach them how to program, how to use objects, and catch their interest. It is a well known fact that C + + was never meant to be understood, let alone be taught; therefore we could expect a great challenge to run this course successfully. The solution to this problem may be found in the OOP approach itself: use objects to teach about objects.

Most of the course materials on introductory programming still use the same approach used when computers were expensive and unavailable. This has evolved slowly to assume that the student has a computer to test programs as he goes through class material. However, still the instructor has to build the new concepts on top of those already known. For example, we must teach some boring I/O very early so that students can see the results of their programs. Also, we often require some more math ability than really required so that we have a subject to practice programming (factorials, fibonacci numbers, prime numbers, etc.) In many cases the algorithms involved are not familiar to the students and the result is that they have to learn the math concept, the algorithm and the language all at once. If you add this with the intricacies of C++ you may have the perfect formula for failure...

The proposed course is built on the following principles:

- Stress only on issues we want students to learn and as little as possible at a time.
- Have students implement algorithms they are familiar or can easily develop.
- Provide invisible solutions to concepts yet to be explained.
- Make results of programs evident to students.
- Stress on long-range software engineering concepts such as: reuse, maintainability, readability, etc.

I think it can be easily understood that this is not a trivial task. OOP has all the principles we need to support this kind of teaching but... the tools are not yet available to support this; as a result, an attempt to provide a friendlly learning environment requires a substantial work in developing course material. Nevertheless, this work has to be started sometime, and some suggestions are shown in the next section.

Proposed Textbook

A textbook under development is an attempt to address most of the above principles; this text is actually composed of three distinct parts:

- A textbook;
- A storybook;
- Software support;

The textbook is the standard text one would expect to find. The contents depart from the traditional due to the availability of the other unique parts.

The storybook is, actually, a collection of fictional tales that illustrate, in an informal way, most of the concepts that will be dealt with in the text. Those stories are scattered in two or three page segments throughout the text in such a way that they match the subject in the textbook chapters.

The software support takes an approach very different than what you can usually see in textbooks that provide a diskette. The standard procedure is to provide examples to the student. Instead, this proposal is to provide working material; a set of classes and programs that the student will use to develop the exercises. This software will allow, for example, animation, sound, and other features to be used to show results of the students' exercises.

The contribution of this software to the course cannot be underestimated:

- Stress on one subject at a time:
  Ordinarily, students are required to learn I/O statements so that they can see the result of their programs; in this case, we do not require that; students manipulate objects that show on the screen and whatever they do with those objects, they can, immediately, see and evaluate whether they have done the right thing or not.
  The software support hides all details which are, at a given point, either unknown or irrelevant to the students, and allows them to concentrate on the relevant issues like the algorithm development and implementation.
- Algorithms that are familiar to the student
  Instead of prime numbers, roots of equations etc. the first lessons use algorithms such as performing fitness exercices, walking around a rectangular room, drawing familiar shapes, etc. Those are things that the students know how to do, they only have to worry about how to explain that in a programming language; this avoids the problem of having to learn the algorithm and the programming at the same time.
- Invisible solutions to unknown problems
  Anything that the student does not know and does not have to learn right away, is implemented in some kind of object whose functioning is invisible to the students. This allow students to see results in graphic, animated and sound forms, even before being exposed to input/output.
- Make results evident
  Beginner students often have substantial trouble to understand the results of their programs. No wonder; all they usually have is a bunch of numbers which they don't readily understand whether are right or wrong. We can do much better than that. We can make students actually see a character moving or talking according to the instructions in his program.
- Software engineering concepts
  How can students understand the burden of maintenance if they never have to fix or modify a piece

of code? how can they value reuse? what about documentation?
It is important that students be required to recycle software that they have developed in order to solve different exercises. As they go through code they have developed a few weeks ago, they will start to value comments. Exercises should also include fixing or altering supplied programs. Reuse of software by means of deriving classes, should be extensively explored. It is important that students also know how to create classes and not simply to use a class library.

Preliminary results

Building of the textbook/software package has been a very interesting experience.
Environment
I chose the Turbo C++ for windows environment; I think the windows environment is particularly important for the student and Turbo C++ is a reasonably complete package widely available for a convenient price. Unfortunately, the choice of the windows environment forced me to dive into a different new (and complicated) world despite the help of object windows.

The most unusual aspect of this software is that you are not developing an application; you are developing an open-ended software that is to be completed with the students' work!

Moreover, the students are supposed to have the impression that they are doing the programs all by themselves, therefore it is mandatory that they are subject to the minimum possible rules. All they should care are the objects that are offered for their use and the programming language itself.
Objects
A few header files are supplied that include all the classes that students are supposed to use; from these classes, they instantiate the objects which are used to implement and test the programs. Some classes available are:

- Athletes:
  pictures of people that respond to messages that show the person in different positions, output text in a box, or emit sounds. Many algorithms make use of athletes: they are used to explain sequence, functions, repetitions, decisions, etc.
- Robots:
  A robot is shown inside a room that may either be empty or contain walls so as to constrain the robot's movements. Robots can be used to draw pictures or to explore rooms. Robots are useful to explain decisions arrays, functions etc.

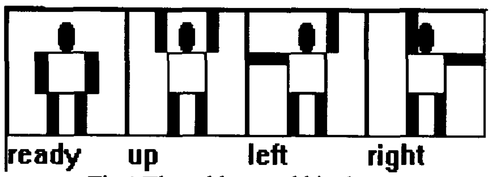

Fig.1 The athlete and his shapes

As an example of the exercises Fig.1 shows a member of the athete class in the four possible positions: ready, up, left, right.

By sending appropriate messages the student can make the athlete perform several fitness exercises; for example: alternating between "ready" and "up" the athlete performs a "JumpJack".

The code below would cause the athlete, represented by the object "Sal" to perform a Jump Jack:

```
    #include "athlete.h"
    void main()
    {  athlete Sal;
      Sal.ready();
      Sal.up();
      Sal.ready();}
```

Observe that the students are programming in the "hostile" C++ environment but the supplied class "athlete" removes some of the "hostility" requiring only adherence to C++ syntax rules. Also, the students understand that what they want is to have the athete alternate between these two positions (which a litlle later will include functions and repetitions). This is what we call an algorithm that does not require further learning, the students know it very clearly.

Athletes may also display a few words or emit sounds; the animation may be synchronized very easily with the real-time clock in the computer. This whole environment adds to the purpose of letting the students learn to program with greater ease and incentives.

Faculty Education:

A very relevant obstacle for incorporating OOP in an integrated fashion in any curriculum has to do with the education of faculty. Of course there are faculty members that are not committed to OOP and, again, I don't want to argue with that. The point is that even though many faculty members are charmed by the OOP idea, they have a lot of effort to get into it because:

- We cannot attend classes: our work makes it difficult to arrange for a specific schedule to accompany a class lecture regularly. Moreover, most of us like to be standing, not seating in a classroom...
- We think we can do it by ourselves: As a matter of

fact we can, at the expense of a very strict discipline. We get a compiler, a book and start to work. How long does this discipline last? It may vary from one to another but, unless you really have an objective (like having to teach it next quarter) you may end up doing like most or our students and leaving this for whenever you have free time (after retirement, presumably...).

In my understanding, what is needed in this case is an environment that:

- Is flexible enough to be accomodated with a moving schedule;
- Provides the ability to self-pace;
- Allows consultations and exchange of ideas with peers;
- Requires a minimum progress rate;
- Requires as little time as possible.

An interesting experiment was conducted using a hypertext based system running on PC's and the Internet access to provide an evironment with the above features. An experimental course was started with 22 participants, all faculty, from US, Puerto Rico, Argentina, Brazil, Bolivia, Colombia, Chile, Cuba, Israel and Spain.

The key component in this experiment is was the hypertext- based course material that runs on a PC. This was originally developed to replace transparencies in the classroom and provide several programs to demonstrate use of features and techniques of OOP and C++ in graduate classes. Later, some more material was added to this package to reduce dependency on a textbook. This package, clearly, runs independently of internet and allows the reader to view each lecture subject in any order, skip some, ask for more details in others, refer to previous topics, look at sample programs, etc. Moreover, since this package runs under MS-Windows, it is very easy to switch to the compiler and execute or modify the programs supplied in source form. Homeworks are requested for each lecture and a test program is provided. Most of the time the student has to produce code to implement a class (hopefully reusing previous exercises) and run the test program. This not only keeps the time at a minimum but also forces the new code to comply with specifications.

This package alone would provide a little more than a textbook but, if this is put together with internet access, the benefits raise exponentially.

First thing to notice is that participants now may have somebody to talk with. During the specified course duration they may correspond with the instructor via Internet to ask questions, make suggestions and submit homework; and that is not all! Since the internet address of that group of participants is broadcasted they may exchange ideas and views between themselves also. This is specially useful if one considers that in some regions there may be nobody to exchange ideas on the subject!

Second thing is that, although the pace is loose, there is an effort to synchronize the minimum speed. Each lecture requires a homework which should be tested and forwarded through the internet. If the homework is not received by the due date, the participant will not receive the material of the remaining lectures and will, therefore, be out of the course.

Although many participants did not complete the whole course, results may be considered satisfactory. Some participants kept a high degree of interest and only occasionally, lost pace with the original schedule. Some others have given up after the second homework. As it could be expected, some countries have substantially more difficulty than others with internet access, and this clearly influenced the particpants in very interesting ways. In most cases, participants with restricted access to internet were the most active.

In general, the combination Hypertext/Internet is highly effective for teaching this kind of material and far less expensive than satellite television or teleconferencing. The moderate costs are even motivating the interest in providing a similar course as a self-paced version for graduate students. Another possibility under investigation is to change the hypertext platform to Mosaic.

Acknowledgements:

This work as been supported by the CNPq, Brazilian Council for Scientific and Technological development; I also wish to thank the Federal Universtity of Rio de Janeiro and Santa Clara University, specially to Prof. Daniel Lewis, for the opportunity to develop and test the course materials including the experimental course via the internet; to the International Technological University for the continued support to this project and to the University of California, Berkeley for the interest in experimenting a self paced version.

References:

[1]Gabel, David - Software Engineering. IEEE Spectrum January, 1994 - Technology forecast. pg 38.
[2]Lieberherr, Karl J. and Xiao, Cun - Object Oriented Software Evolution. IEEE transactions on Software Engineering. V.19 N4 April, 1993.
[3]Banker, Rajiv; Kauffman, Robert J. and Zweig, Dani - Repository Evaluation of Software Reuse. IEEE Transactions on Software Engineering. V.19 N4 April, 1993.
[4]Softinco- Software Industry Coalition meetings and unpublished work documents.