IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Suite 200, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com
ITP4486

Old Dogs and New Tricks: Retraining Legacy Programmers in Object Oriented Technology

H. James Nelson
David Eccles School of Business, The University of Utah, Salt Lake City, UT 84112
(801) 587-9165, actjn@business.utah.edu

Deb Armstrong
School of Business, The University of Kansas, Lawrence, KS 66045, darmstrong@ukans.edu

ABSTRACT

Faced with a chronic shortage of skilled object-oriented programmers, and burdened with an oversupply of procedural programmers leaving Y2K projects, organizations must find ways of retraining existing programmers in the new tricks of object technology. However, traditional methods of training do not address the difficulties of making the mind shift from one paradigm to another. This paper describes an effort to develop a course that shifts experienced procedural programmers into "object thinking," thus making subsequent traditional OO training more efficient and effective. We describe a long-term exploratory field study that follows students over two years and compare their OO thinking to that of expert OO programmers and to programmers who underwent traditional OO training. This research is supported by a grant from The Boeing Company.

INTRODUCTION

Over the past few years we have seen considerable effort being put forth to correct a design problem in legacy computer systems: the Year 2000 Problem. As this work declines (whether it is complete or it is too late) projects that have been on hold are reawakening. Organizational information systems (IS) departments are turning from legacy system "fire fighting" to new systems development. Once again, organizations are searching for the Silver Bullet of software technology that will allow the rapid development of highly flexible, easily maintainable, user friendly software.

In their search, organizations have discovered a suddenly much more mature Object Oriented (OO) programming environment. What was once a bewildering array of competing modeling techniques, development approaches, and diagramming methods, is now a largely unified set of standards for OO software analysis, design, and implementation. This result is primarily due to the work of the Object Management Group's Object Analysis and Design Taskforce and the development of the Unified Modeling Language (UML) (Martin & Odell, 1998).

However, organizations have also discovered that there is a severe shortage of skilled information systems professionals. Especially scarce are those IS professionals who are well trained in OO techniques. So on the one hand, organizations have a pool of programmers who are experts in the organization's domain (whether it is banking, insurance, or aircraft manufacturing) but unfortunately are experts in legacy software development and procedural programming languages. On the other hand organizations cannot get the skilled IS professionals they need (at any price) for OO development. Even when found, these new hires may take years to become integrated into an organization's interpersonal network and become very "useful" to the organization (Lee & Allen, 1982). The solution, it seems, is to retrain the pool of expert procedural programmers in the Object Oriented techniques.

Unfortunately, retraining experienced procedural programmers has proved to be difficult in practice (Wasserman, 1991). There are some significant differences between procedure-oriented and OO techniques. The OO model requires a fundamental shift in the way programmers think about and approach problems. Making this shift is far more difficult than learning another procedural programming language, despite what some OO language proponents would have you believe. The procedure-oriented programmer must begin thinking in terms of objects that *are* things and have certain behaviors rather than blocks of code that *do* things. This shift in thinking is not easy to achieve.

Research based on learning theories indicates that an expert programmer's prior knowledge of procedural programming techniques actively hinders his or her learning object technology, making their learning more difficult compared to those who have had no prior programming experience at all (Nelson, Irwin, & Monarchi, 1997). Similar learning difficulties and have been well documented in other domains such as learning a new natural language or a new motor skill (Crowder, 1976). Typically, the experienced procedural programmer is thrown in to the OO world with little more than a class or two of OO theory and the differences between C and C++, and is expected to be immediately productive. The shortened software development cycle so prevalent today does not allow the programmer to make the shift to OO thinking. Stuck "between worlds" and pressured to produce *something*, the programmer falls back on well-known procedural techniques. He or she produces code that is representative of neither procedural or OO thinking. This hybrid code is notoriously difficult to maintain; much worse than strictly procedural or strictly OO code. This problem is nicely summarized in the following two quotes from different corners of IBM:

> "IBM's new object-oriented COBOL products allow the world's three million COBOL programmers to create objects in a language they already know. With little or no retraining, developers can migrate to an object-oriented environment and immediately receive the associated benefits."
> Tim Negris, vice president, marketing, IBM Software Solutions.
> "Basically, if your manager hears about object-oriented programming and decides that this is the way the world

This paper appears in *Challenges of Information Technology Management in the 21st Century,* the proceedings of the Information Resources Management Association International Conference.

> is going, what he'll do is he'll give you Stroustrup and a compiler and say, "Here, be wonderful." And you, the hapless programmer, realize you can't be wonderful, but you have to be productive. You have to show something, so what you do is slip back into the way you were programming before."
> John Vlissides, IBM T.J. Watson Research Center

The key to this problem is to circumvent the active interference of an experienced programmer's procedural knowledge. If one could do this, if one could shift a programmer from being an expert learning a new paradigm to being a novice learning that paradigm for the first time, training could be more effective, the learning curve could be shortened, and the programmer could produce good software sooner. In addition, the experienced programmer's domain knowledge is ready to be used, and does not have to be built up as would be the case with new hires.

This paper describes an exploratory field experiment in enhancing OO learning. We developed a short course (two half-days) that is designed to shift expert procedural programmers into "OO thinking." Three classes of students were followed for up to two years to determine if this class had any effect on their subsequent performance in other OO classes and on OO projects. In the next section, we describe the theory behind our "Object Oriented Technology Training Improvement" (OOTTI) project. We then describe the course structure itself, and then the results we have obtained to date. We conclude with a discussion of how courses such as OOTTI can improve learning in domains other than object oriented programming.

COGNITIVE INTERFERENCE AND LEARNING

In the traditional learning model, the student is taught the concrete aspects of a language and gradually gains the more abstract "expert thinking" of the programming paradigm. The student is typically an expert practitioner of procedural programming and is deep in doing, generally automatically, procedural programming. The programming is performed naturally and without much thought about the rules of the language. He or she is in the "concrete" of procedural programming. When training in OO begins, the student is taught definitions, concepts, and language constructs and is then presented with a series of problems where these elements can be used. The training moves the student from concrete procedural programming to concrete object oriented programming. Once outside the classroom, the student puts these concrete concepts into practice and gradually becomes skilled in their application. Over time, the programmer gains the abstract, "expert thinking" necessary to use OO techniques more or less automatically (see Figure 1).

Figure 1. OO Training, state of the art.

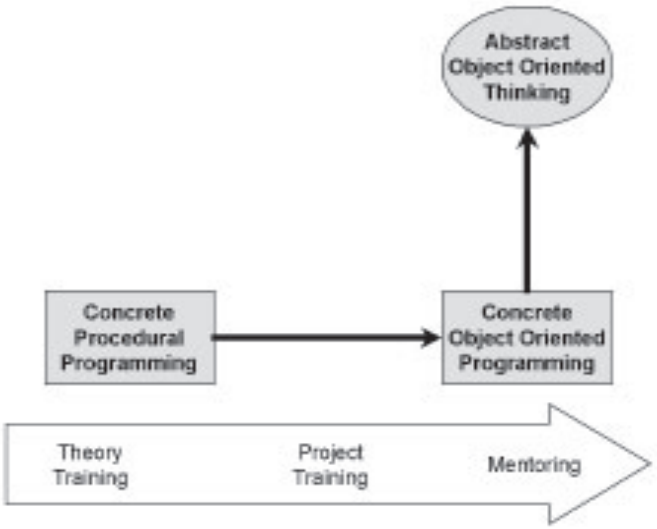


Figure 2. The "expected" learning curve.

The traditional training model works well if the student has no prior programming experience or is simply learning a new language in a familiar paradigm. The product that is produced may not be as efficient or as "clean" as that produced by an expert in the paradigm, but the product that is produced will generally follow the rules of the particular programming paradigm. Figure 2 shows a "normal" learning curve.

However, the traditional training model breaks down when the student is expert in one or more languages in one paradigm (such as procedural programming), and attempts to learn a language from another paradigm (such as object oriented programming). The new paradigm requires the student to learn an entirely different way of thinking about a problem before using the new language. For example, in the procedural paradigm the programmer approaches the problem by identifying functions, the steps that must be followed to produce the result: the "how." In contrast, in the object oriented paradigm, the programmer approaches the problem by first identifying the things that take part in the process: the "what."

An expert structured programmer will learn the theory behind the language and most of the new language constructs fairly easily. However, on the job the student will discover new problems unlike those "in the book." Under deadline pressures, the student will fall back upon the more familiar procedural programming knowledge rather than try to figure out the unfamiliar OO programming knowledge. Consciously or unconsciously, cognitive interference takes hold and the result is a software product that is neither procedural nor OO, and much harder to maintain than either. Figure 3 shows the changes to the learning curve brought about by cognitive interference.

If the expert procedural programmer were to "think like an expert" in the OO paradigm, then the procedural programming knowledge would not surface in response to a problem. The student would remain in the OO paradigm, cognitive interference

Figure 3. Learning curve for expert procedural programmers.

would be avoided and the expert would learn at worst as a novice would. The problem, then, is to get the student thinking as an expert *before* learning any of the theory or constructs of the language.

QUANTUM SHIFT LEARNING

In order to create this expert OO thinking in an expert procedural programmer, we used techniques described by Senge (Senge, 1990; Senge, Kleiner, Roberts, Ross, & Smith, 1994). Briefly, this involves surfacing and challenging the student's procedural programming mental models while building a team learning environment, building a shared vision of the new OO programming paradigm, and then engaging in systems thinking to solidify the abstract OO concepts with concrete examples. We designed a short, two half-day, introduction to object oriented thinking course that would serve as a first course to any other object oriented training. The steps, and how they were integrated into the course, are described more fully below. Figure 4 is an outline of the course.

Surfacing and Challenging Existing Mental Models

The students in the course are all experts in procedural programming. They perform their programming tasks naturally and easily, with about as much thought as one would use when riding a bicycle. The problem is to move them to performing object oriented programming just as easily and without falling back on their natural procedural knowledge.

The first half-day session is dedicated to making the students aware of what they are doing as they perform procedural programming. The students must be made aware of the mental models they use. This first session is tedious, frustrating, repetitive, and very necessary. The process involves asking the students questions, and through these questions leading them to understand what they had been performing naturally. The questioning process looks something like the following. The answers to a question drives the next set of questions.

- Tell me about your experience in procedural (structural) programming.
- What do you call the piece you work on? What is the smallest piece that a person would typically be working on?
- Is there a process you follow?
- Once you get to that level, what do you do?
- What do you do when you have more complex system to build?
- What's the first thing you think about?
- What do you do when dealing with very small pieces of code?
- How do you code and come up with a black box?
- Following any specific order? Top to bottom?
- How do you know how to do it from requirements?
- What is it that you do to get what the user wants?
- Suppose you're done with the requirements document, what do you do to turn it into code?
- Is there a method to decide functionality?
- What's the next level of decomposition?

The student is driven to explain, in tedious detail, how they perform what is for them an automatic process. They travel from the concrete programming to understanding procedural programming in the abstract.

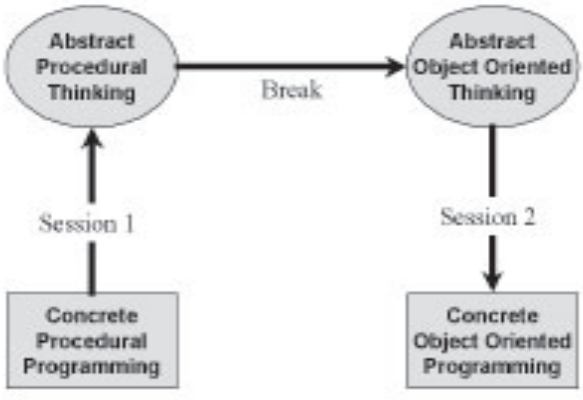
At the end of the first session, the students also begin to understand some of the problems of the procedural programming paradigm. Procedures, the business processes, are the parts of the organization that change most rapidly. The students discuss the amount of software maintenance they have to do and how virtually all of it is caused by changes in requirements, driven by changes to the business processes. Cognitive dissonance sets in as they wonder why one would base a programming paradigm on the part of the organization that are the most unstable. The first session ends with a suggestion that the students figure out a method to make software more stable.

Building a Shared Vision

The students return the next day full of ideas. With very little guidance, they realize for themselves that basing a programming paradigm on things that do not change (objects) makes software much more stable and flexible than basing a paradigm on things that tend to change rapidly (the processes). With this realization, they take ownership of the solution; they came up with it themselves, and this is much more powerful than if they had read it in a book or heard it in a lecture.

- What changes in a system?
- What sort of business rules?
- How? What's different?
- What initiated the change?

Figure 4. Structure of the OO Thinking course.



- What causes change?
- Is it easy to generalize code?
- How can you write software so that things don't change as much?
- How do we create a system so it responds well to change?
- What is data?.
- What are the characteristics of data?
- What should be based on data?

Then, the students are encouraged to work on a programming problem using these new insights. Guided by the instructor, they invent for themselves how to approach a problem, how to decompose it, and how to construct a solution that is stable, is flexible, and is easy to maintain. They develop abstract, expert OO thinking.

Systems Thinking

Finally, and only after the students have created OO thinking for themselves, are they introduced to definitions, methods, and language constructs. They learn the "real" names for the concepts that they invented. They learn about classes and instances, methods and message passing, inheritance and polymorphism. Time permitting, the students finish with a walkthrough of a simple OO program showing how these concepts are put into practice. They travel from abstract OO thinking into concrete OO practice.

AN EXPLORATORY FIELD EXPERIMENT

We tested our theory of learning with a field experiment at a major manufacturing organization. Twenty-eight expert procedural programmers took part in the study, split across three separate classes.. All were expert FORTRAN programmers with an average of twelve years of experience. All had limited (if any) exposure to OO technology. The course evolved over time: the first two classes were delivered over three three-hour morning sessions. The final class was delivered over two four-hour morning sessions. The classes were structured as we described in the previous section. These twenty-eight procedural programmers formed our treatment group.

We wished to determine if the students had indeed gained "expert OO thinking." To do this, we needed a baseline for expert thinking. We conducted semi-structured interviews of five expert OO developers from the same organization. These experts were not self-selected, but rather were identified as the mentors or "OO gurus" that everyone would go to for OO help. These experts were asked a set of questions to help develop a model of expert OO thinking. The interviews were taped, transcribed, and identified by a code number to preserve the confidentiality of the interview.

We also interviewed a "control group" of programmers from the same organization. These programmers were experts in procedural programming, and who had taken the traditional training route to OO programming. They had from one to two years of OO programming experience on live projects and had taken an average of three OO classes, generally classes in development methods and in C++ and/or Java. These programmers were asked the same questions as the experts.

RESULTS

The students were interviewed at intervals after the class to determine if the class had any affect on their subsequent OO learning or their performance on OO programming projects. The students were interviewed one month, one year, and two years after the class.

After one month, the students generally forgot the definitions (as expected). For example, they had difficulty defining "class," "instance," "polymorphism," and so on. A key question in this immediate follow-up interview was "How do you map OO concepts to procedural programming concepts?" All but one student stated that there is no mapping from OO to procedural programming. This is an encouraging result as it is the same reply as that given by the expert OO programmers:

> "I don't think you can come up with a comparison that says that this concept maps with another concept. It's a really different way of doing things."

However, the "control group" had other opinions:

> "Methods map to procedures. Data flows map to messages. I do think of methods very much like a subroutine."

It appeared from these initial interviews that the students had developed at least preliminary expert thinking, although they had not been able to put it into practice.

> "I think that I would know when I'm drifting astray. I'd have an idea that, no this isn't quite right, and I'd go back to the text or some reference and satisfy my unease and redirect myself. I'd have an idea that I was drifting into "Structured Land." Enough to stop myself before I went the full route."

One-year Followup

The first long-term interview was conducted one year after the class in the last quarter of 1998. There was considerable difficulty in locating the students. Several had left the company, and many more were not available for a variety of reasons for the one-year followup. A total of eight students were finally interviewed. Six of the students were participants in Class 1 and two of the students were from Class 2. The length of the interviews ranged from 35 to 75 minutes.

The main finding from these interviews was their observations of the problems that experienced procedural programmers have when learning OO technology: the change in thinking. Seven of the eight individuals mentioned this as the number one problem in making the transition.

> "The concept of getting away from the procedural and more into the object, you know, encapsulated object type programming rather than thinking in the serial type step by step through this whole thing."
>
> "It seems like with OO you look at it more as they're more autonomous for objects, whereas with procedural I don't think of it that way at all. I try to think of them as sort of being independent. That they are self-contained. They know what to do what they do, and they've got their own little world and they take care of it. Whereas I don't think of it like that when I'm doing procedural."
>
> "Changing their mindset. You have to think differently to work with object oriented."

One informant thought the object oriented concepts were not that different from the structured concepts stating,

> "I thought the whole idea of trying to teach somebody who knew structured programming to go to object oriented programming was not that big a deal. To me, I guess it's somewhat fundamental. So why should it be a big deal?"

However, this student also had a great deal of difficulty with the structure of the class itself. She strongly disliked the "guided questions" method of teaching, preferring more answers rather than more questions.

Two-year Followup

Interviews with the students were requested again in the last quarter of 1999. The same problems occurred in trying to locate the students for the followup interviews. Many more students had left the company, and more students were unavailable. A total of four students were interviewed. One student was a participant in Class 1, two were from Class 2, and one was from Class 3. The length of the interviews ranged from 20 to 45 minutes. The results paralleled the one-year interviews.

> "Well, some say you're still trying to accomplish the same thing, but you basically just take a different approach to get there. I guess I don't know that the object concepts map that well back to the structured concepts. The way you have to think about the problem and break them up is very different."

> "It's just the way you think about a problem has to totally change. One of the main differences with the OO concepts is that the data stays along with the objects and also the code. Where you don't have that in structured. So you have to do more thinking ahead of time than when you're doing structured programming. As far as what you want your objects to be and stuff like that."

Comparison with the Experts

Five expert object oriented programmers with the same organization were interviewed in the last quarter of 1998 and the first quarter of 1999. The length of the interviews ranged from 45 to 90 minutes. The experts had an average of 12 years of OO experience.

The results of these interviews were very similar to the student interviews. The experts also thought that the main problem that experienced structured programmers have when learning OO is the change in thinking.

> "Well, it's a significant shift in mindset and the problem is you have to unlearn things that are structured top down composition, functional programming taught you."

> "The biggest problem is changing from the functional decomposition to the object decomposition. Changing from their mode of thinking of writing down a sequence of things that happen to other things, to data that's somewhere else. The behavior/data separation that you have in a procedural language to the notion of a 'thing' that contains both data and program."

DISCUSSION AND CONCLUSION

Generalizations from such a small sample are, of course, risky. Any long-term studies of experienced programmers in a volatile job market such as exists today is difficult. However, the results we obtained are encouraging enough so that we will continue following as many students as possible through their OO careers, and we will continue to refine and enhance the Object Oriented Technology Training Improvement class.

The results of our interviews with the students we could locate indicates that immediately after the class, and over an extended period of time, they had the same view of object oriented programming as OO experts. They viewed OO as a completely different way of thinking about problems, and when stuck in a difficult task, they did not fall back on their old procedural programming experience. In contrast, the programmers who did not take our class continued to have a strong mapping from object oriented concepts to structured concepts. In confidential interviews, they revealed that their code was not "pure" OO. They did employ procedural techniques, even when they knew it was "wrong," in order to meet deadline pressures. These habits persisted even after two years of experience on OO programming projects.

The training technique we used to shift expert procedural programmers to expert OO thinking should work in any environment where similar paradigm shifts are necessary. For example, in transitioning thinking from the "chaos" to "process thinking" of the SEI/CMM capability maturity levels, shifting from monolithic to distributed computing, and so on.

This exploratory study is just the beginning in a series of experiments that explore the nature of paradigm shifts in organizations.

REFERENCES

Crowder, R. G. (1976). *Principles of Learning and Memory*. Hillsdale: Erlbaum.

Lee, D. M., & Allen, T. J. (1982). Integrating New Staff: Implications for Acquiring New Technology. *Management Science, 28*(12), 1405-1420.

Martin, J., & Odell, J. J. (1998). *Object Oriented Methods: A Foundation*. Upper Saddle River: Prentice-Hall.

Nelson, H. J., Irwin, G., & Monarchi, D. E. (1997). Journeys Up the Mountain: Different Paths to Learning Object Oriented Technology. *Accounting, Management, and Information Technologies, 7*(1).

Senge, P. M. (1990). *The Fifth Discipline: The Art and Practice of the Learning Organization*. New York: Doubleday.

Senge, P. M., Kleiner, A., Roberts, C., Ross, R. B., & Smith, B. J. (1994). *The Fifth Discipline Fieldbook*. New York: Doubleday.

Wasserman, A. I. (1991). Object Oriented Thinking. *Object Magazine, 1*(3), 10-13.