Cognitive Complexities Confronting Software Developers Utilizing Object Technology

Allen Stix
Pauline Mosley
Computer Science
Pace University
Pleasantville, NY 10570
astix@pace.edu
pmosley@pace.edu

Abstract

Many managers are adopting object technology initiatives to develop high-quality products more efficiently. Consequently, software practices are changing and there are repercussions across corporate and academic training. As new practices promise to yield increasing benefits in the software development cycle, the complexity is becoming proportionately greater.

This paper examines the notion that object technologists, like the workers in artificial intelligence, have underestimated the complexities associated with the analysis, design, and coding of software from "virtual things" (i.e. objects). One manifestation is that the information systems community making up the software infrastructure in many organizations is resisting, misapplying, or only very slowly understanding objects. Another manifestation is that the academic community is perplexed by the way that objects are changing the view of software and how, correspondingly, the computer science curriculum must be adapted.

Going beyond merely documenting the cognitive complexities of object technology, this research identifies those atomic-level constituents responsible for the shift from simple to complex with respect to analysis and design. This paper concludes by proposing a pedagogical model that addresses these constituents and could thereby reduce the learning curve for retraining practitioners and restore clarity to the computer science curriculum.

Keywords: Object-oriented programming, software engineering, software development, pedagogy

1. STATEMENT OF THE PROBLEM

To survive, software organizations must have the ability to adapt to a dynamically changing technological culture (Holmberg and Mathiassen 2001). Literature suggests that survival of the fittest is contingent upon how quickly an organization can respond to technological options and market needs, while at the same time delivering high-quality products and services.

One way organizations' have chosen to deal with these changes is to implement object technology. This methodology facilitates the use of reusable software component libraries thereby producing better systems structures that are adaptable and extensible. Although, the benefits of object-oriented systems are recognized, the cognitive complexities associated with these pure object solutions present a documentably steep learning curve, which can lead to longer development cycles and more costly investments. Experts say that this long learning curve prevents companies from using object-oriented programming properly or taking advantage of it, and that its benefits can't really be taught, at least not understood from a book (Sleeman 1986;Tilley 1996; Reid 1993). In addition, studies have shown that object technologists, similar to the workers in artificial intelligence, have underestimated the complexities associated with "virtual things" (i.e. objects).

There is little empirical proof that the complexities associated with object technology are due to the inherent nature of its paradigm or if it is due to improper learning acquisition, which then leads to misapplication of the technology. Mr. Fred Brooks (Brooks 1987), author of *No Silver Bullet*, states that the essence of a software system is the conceptual, almost inspirational, core of logic and design that forms the underpinnings of the system. He goes on to say that because the role of accidental factors is limited and since technology can help developers only with the accidental factors, no development in technology can affect an order of magnitude increase in the productivity of software developers. Therefore, for organizations to capitalize on the benefits of this technology and impact the quality of software, there is a need to improve the usability of objects and object technology to make the concepts amenable to information technologists , but especially programmers. If object-oriented programming is indeed a new paradigm, then all of the old techniques appropriate for other paradigms need to be re-examined and possibly replaced by new techniques. This replacement includes pedagogical techniques no less than it includes programming techniques (Bergin 2000).

The goal of this paper is to identify atomic-level properties that cause the knowledge acquisition in object technology to shift from simple to complex with respect to logic and design. In addition, this study seeks to establish a foundation for a pedagogical model that will reduce the learning curve for educating high quality designers in industry and academia (the retraining of commercial developers and the indoctrination of those new to software development), thereby promoting this technology to gain wider acceptance and proper implementation usage.

2. OVERVIEW OF THE RESEARCH PROCESS

The focus of this research is to investigate the cognitive factors that are likely to impact object-technology adoption or implementation. Qualitative methodologies will be used to obtain an understanding of the complexities associated with object technology. The objective is to further improve the human aspects of computer utilization: ease of learning, ease of use, software developer satisfaction with the system, and the impact of software construction on quality systems.

Research Methodology – Industrial Content Analysis

A field study using qualitative research methods was used to develop an understanding of how software practitioners acquire their knowledge of object technology. (Kaplan and Maxwell 1994) argued that the goal of understanding a phenomenon from the insider perspective is all but lost when textual data is quantified. The qualitative method is aimed at explanation and understanding, rather than prediction and control. While the findings from such a study are particularistic, (Eisenhardft 1989; Glaser and Strauss 1967; Yin 1989; and Orlikowski 1993) suggested that "analytic generalization" from the results of such a study to theoretical concepts and patterns, rather than "statistical generalization" from samples to populations, can be produced by combining insights generated inductively through the field study with those obtained from existing formal theory.

Qualitative research can be either positivist or interpretive. Positivist studies attempt to test theory and to increase the predictive understanding of phenomena (Myers 1997). Orlikowski and Baroudi classified information system research as positivist if there was evidence of formal propositions, quantifiable measures of variables, hypothesis testing, and the drawing of inferences about a phenomenon from the sample to a stated population. Interpretive studies attempt to understand phenomena through the meanings that people assign to them (Myers 1997), and are "aimed at producing an understanding of the context of the information system, and the process whereby the information system influences and is influenced by the context." Interpretive research focuses on the full complexity of human sense-making as the situation

emerges not on predefined dependent and independent variables.

This study seeks to generate an understanding of how object technology learning takes place and the complexities associated with the learning process, rather than to test a set of hypotheses. Thus, this study can be classified as interpretive rather than positivist in nature.

This perspective is important because insights generated from qualitative studies provide a useful complement to quantitative computer science research by enabling results to be understood and explained within the business context. The qualitative methodology is perfect for this type of study because it is appropriate as a discovery methodology. This approach is conducive to understanding the exact sources of the resistance and the complexity of implementing an object technology. (Recall, these are the barriers identified by (Kenneth Kendall 1999)) associated with technological advancement in general). Object technology is an emerging technology, and unless we understand why there are issues of resistance and complexity, this technology will not advance to the technological sublime phase.

This field study design enables the learning phenomena and its application within an organization to be understood in terms of the interactions of the conditions and actions that exist within the organizational context. Therefore, by incorporating multiple sources of largely qualitative methods (interviews, surveys, direct observation, etc.), the research strategy selected for this study provides for triangulation of evidence and the preservation of contextual factors, while minimizing the likelihood of overlooking operative variables and dynamic processes.

Participants

Participants for this study were recruited on the basis of the researcher's prior profession as a corporate technology trainer for major Fortune 500 firms. The goal was to obtain a representative cross-section of firms and programmers with varying levels of experience and expertise from which meaningful data for the study could be derived. Programmers representing various training backgrounds, genders, ethnicities, and various corporate cultures were sought to provide a variety of different perspectives regarding the use of object technology.

Participants spanned all functional areas and were diverse in organizational levels. The participants included thirteen firms (two legal institutions, one banking firm, four software development firms, one utility company, one city organization, one life insurance company, and three financial firms). Thus, the data collected from these participants can be considered to be a good representation of the domain of possible responses.

Primary Data Collection Method - Interviews

Data for this study was collected using a variety of methods, including personal interviews, surveys, and on-site observation. Personal interviews were the primary source of data. Interview questions were pre-tested first with academic colleagues. Interviews for this study were semi-structured, allowing for open-ended responses and were guided by a set of questions regarding participants' programming experience and knowledge of object technology. Information was sought on object technology concepts and on perceptions of the organizational environment, size, structure, composition, location, training, and support.

Interviews were conducted at the participants' place of work in the participant's office or work area. Additional interviews were conducted via telephone for those participants' where a face-to-face interview was not convenient. Interviews were scheduled with an intended duration of two hours. Most interviews lasted 75 minutes and probably could have gone longer, but the participants usually needed to return to work.

The purpose of the interviews was to gain an emic view of learning object-technology with respect to logic and design through the respondent's perceptions, attitudes, and opinions. Thus, while the planned set of questions served as a guide for the interviews, each interview proceeded in a slightly different fashion from the others. That is, questions were not read verbatim, nor in exactly the same order, to each participant. This approach enabled a more natural flow of information and an ability to probe more deeply into the striking responses given by the participants. This approach also encouraged the participant's to say whatever came to mind, and in any order, so as to capture their first impressions and impulse reactions for consideration.

Secondary Data Collection Method - Questionnaires

Apart from the interviews, additional data was collected via questionnaire. A cover letter explained the purpose of the study, sought cooperation for participation, and requested that the questionnaire be completed by a programmer. Questionnaires were distributed and returned by email. Firms were selected based on the researcher's prior corporate contacts.

Open format questions are those that ask for unprompted opinions. The objective of the questionnaire is to learn how software practitioners acquire their knowledge of object technology as well as to gain insight into their learning process of object technology. Thus, this type of format is good for soliciting subjective data as well as increasing the likelihood of receiving unexpected and insightful suggestions. Questions were tested for

ambiguity, non-colloquial expressions, and succinctness. In addition, all questions went through an "item-rationale" process to instill a cohesive logical flow and fulfill the objectives of the questionnaire.

A total of five firms were requested to submit their responses via questionnaires. Table 1 shows the industries represented in the combined sample of interviews and questionnaires.

Industry	Name	Percentage	Sample Size
Banking	Citibank	7.7%	n = 1
Diversified Finance	J.P. Morgan Jackson Lewis Merrill Lynch	23.1%	n = 3
Government	Riverbay Corp.	7.7%	n = 1
Health Services	Metropolitan Life	7.7%	n = 1
Legal	Simpson, Thatcher & Barlett, Amster, Rothstein & Ebenstein	15.4%	n = 2
Software	Oracle Voyetra Turtle Beach MarketData Corp. Kraft, Kennedy, Lesser	30.8%	n = 4
Utilities	NYNEX	7.7%	n = 1
Total		100%	

Table 1: Industry profile for respondents of questionnaires and interviews.

Table 2 highlights the respondent's position profile. A majority of respondents were programmers 66%; 17% were analysts, 9% were managers, and 5% held top information technology management positions. This cross-section of position status will be significant in the results.

Position	N=51
Top Management (VPs, CIO, directors)	3
Middle management (mangers)	5
Analysts	8
Programmers	34

Table 2: Industry profile for respondents of questionnaires and interviews.

The approach used to analyze the data collected from the interviews and the surveys is known among methodologists in the social sciences as "iterative content analysis" and "open coding." These are described in (Glaser and Strauss 1967; Yin 1989; and Miles and Huberman 1994).

Iterative content analysis refers to examining the data as it arrives (i.e. as it is collected) and modifying our instruments in accordance with early finding. Specifically, it enables new insights to be derived from each round of interviewing before data collection from the next round begins.

Open coding involves identifying themes within what the respondents are saying. Its strength is that variables and processes (i.e. operative factors and how they interrelate) are discovered as opposed to being preconceived. As a hypothesis-generating technique, it is the most appropriate form of investigation when the formulation of causal hypotheses would be premature.

Putting this as the social scientist does, iterative content analysis and open coding techniques are able "to take advantage of the uniqueness of a specific case and the emergence of new themes to improve resultant theory." (Van Hillergersberg, Kumar, and Welke 1995). Data from multiple individuals is continuously contextualized to bridge the settings of the software practitioners with the theoretical framework of the study. This type of data analysis entails going back and forth between data and concepts, and begins early in the data collection process rather that at the end. For the sake of convenience, we shall refer to this approach concisely as content analysis.

3. RESULTS - THE THEMES THAT EMERGED

The content analysis revealed a number of common themes in the interview and questionnaire data. The most prevalent emergent theme was the practitioner's response to the question: How did you learn object technology? A full 89% responded that they were self-taught. This pattern of responses strongly suggests that corporate infrastructures do not perceive training as a high priority. Or, the formal training that practitioners do receive is not adequate; thus they resort to teaching themselves. Additionally, only 22% perceived an increase in the quality of object-oriented software systems. This could be attributed to the fact that the practitioners are self-taught and are misapplying or not completely applying the technology; thus, they are unable to achieve the paradigm's rich benefits. Table 3 summarizes the relative frequency of the responses by the participants.

Percentage of Times the Themes were Identified in the Responses	Percentage of Responses
Knowledge Acquisition	
Obtained by attending a learning institution	33%
On-the-job-training and hands-on practice	22%
Obtained by reading object-technology textbooks (self-taught)	89%
Object Technology Challenges	
Understanding objects	86%
Understanding how to design	71%
Software Engineering Practices	
Increased efficiency	44%
Increased effectiveness/quality	22%
Effective Programming	
One year needed to become an effective programmer	
Two years needed to become an effective programmer	11%
Three years needed to become an effective programmer	44%
More than three years needed to become an effective programmer	22%
Learning institutions should teach design to better prepare students	67%

Table 3: Frequency of the predominant themes from respondents who report sharing this experience, inclination, or belief.

Knowledge Acquisition Theme

Most participants stated that they learn object-oriented programming language predominantly by reading object

technology textbooks and learning on their own. Reasons given by the participants for this included:

(1) Not educating and enlisting management support before switching to object technology,

(2) Upper management unwilling to allocate the necessary time and resources for design up front and for testing following development

(3) Fear of letting management know that object technology requires support and training

As stated by the participants:

In this industry, everything evolves around deadlines. If upper management's expectations aren't met, it may mean a loss of a job or demotion. So when I was told to learn object technology, I did it by reading handouts, textbooks, and asking questions as I went along. I received no formal job training.

Initially, I learned it on my own. Following that, I took advanced courses. Much of my growth in object-oriented technology has occurred from studying the practices of the experts combined with theory and a lot, a lot, a lot of hands-on practice. Practice is the key. But, good instructions and a solid foundation are invaluable.

Others stated that they learned object technology on the job. These respondents perceived their firms to be very supportive in their learning process. Many of them expressed that because they had been with their respective firms for over 15 years, their experience along with their corporate culture provided an environment in which they could learn object technology from other professional programmers regardless of deadlines.

Object Technology Challenges Theme

Many of those interviewed perceived the biggest challenge of learning object technology is understanding objects. Participants cited a number of difficulties in trying to learn, implement, and maintain object-oriented systems. The following comments illustrate this point:

The most challenging part in learning object technology is thinking in terms of objects rather than functions.

The most difficult part for me is design. Schools do not teach design, therefore acquiring this skill is very, very hard.

For the individual coming from a procedural background, learning object technology can be difficult. This is because we have the tendency to think and design procedurally, while at the same time implementing via the use of an object-oriented language.

The Java classes was difficult as well as switching from procedural to object-oriented thinking is key.

This pattern of responses could be because an overwhelming majority of the respondents learned a procedural language as their first language and are shifting to object technology. As one participant stated:

If someone has learned procedural first, they must "unlearn" the old stuff first.

If this unlearning is needed, one wonders how a practitioner can learn object technology while continuing to be responsible for building or maintaining procedural software. Also, this suggests that perhaps further research is warranted for those programmers whose first language is not an object-oriented one. More of these individuals are coming down the pike.

Software Engineering Practices Theme

Participants perceived that use of object technology makes systems more effective and efficient. Programmers stated that once they survive the learning curve, they possess a higher degree of control on systems, and this software approach enables them to increase the control over the execution of software-related duties. As noted by one participant:

The power of object-oriented programming well outweighs any grief experienced while learning object-oriented programming concepts. Eventually one realizes that object-oriented programming, with all its power, is actually easier than procedural coding, because of how it relates to everyday life… objects with properties and methods, driven by events.

With the use of object technology, my firm is able to implement new systems and reengineer legacy systems better than before.

Most study respondents perceived that they were able to make a stronger and more cohesive system through the use of this technology, even if they fully didn't understand the technology. Thus they did not dispute object technology benefits as they can see that it applies practically to software engineering.

Effective Programming Theme

Almost half of the participants, 44%, responded that it takes three years to become an effective object-oriented programmer. Maintaining complex object-oriented systems requires more than syntactical program constructs; it entails being knowledgeable in a much larger skill set that is essential for reusability and extensibility. Participants perceived that the knowledge

acquisition of these skills requires three years before the programmer becomes what they consider fully effective.

However, participants who were procedural programmers for over 10 years perceive the time to be shorter. They view the acquiring of the skills as merely a language transfer as opposed to learning logic and design. In other words, they are viewing the migration to an object-oriented language as the same kind of switch they experienced earlier in going from one procedural language to another. They are insensitive to the migration as a full paradigmatic shift. This may provide a rationale for why 11% stated only one year is needed.

Participants suggested that one way learning institutions could better prepare them for their careers would be to offer courses in design. Design was perceived as the major challenge for programmers to be effective. As stated:

The number of years it takes a programmer to be effective is contingent upon his or her job title. A programmer needs one to two years; a designer needs three to four years. A designer needs more time to become effective. One can only become an expert by practice and experience. That is why I say three to four years.

It took at least three years to get traction and about five years to be effective.

Object orientation is not harder than procedural, it is just a different mind set. However, I would say that it takes three to five years to become effective. If someone has learned procedural first, they must unlearn the old stuff first.

That's hard to say. I guess someone who is fairly intelligent, with decent computer savvy, could become relatively proficient in an object-oriented programming language within three years, one to two for Visual Basic and two to three for C++.

4. TYPOLOGY: ORTHOGONAL SOURCES OF COMPLEXITY

The data collected en masse strongly suggests that there are specific orthogonal sources of complexity. Specifically, six different areas make learning to program in the object paradigm difficult.

Understanding the Notion of an Object as a Virtual Thing

An object may model something real or imagined, something tangible or intangible, or something small or large. When an object models something real, sometimes contrived or fictitious properties may supplement. One of our respondents states "… that it may take a seasoned systems analyst six months to two years to key-in to what an object is, but once the concept hits, they see objects everywhere."

Introductory programming textbooks all seem to open by giving a nod to objects, but these discussions are too superficial to be helpful and the theme is left to dangle.

Understanding the Concepts that Arise in Connection with Objects

Object-oriented concepts include such things as data members to hold state and function members (methods) that offer services, public and private members, constructors, static members, and inheritance.

These concepts unfold to include everything associated with identifying the objects comprising a system, their responsibilities, and their collaborators. In addition, there are issues of factoring and planning for extensible derivation hierarchies so that a polymorphic executive can interact with objects regardless of their type. The principles of object design become so encumbered that patterns are needed for guidance (e.g. the Facade pattern, the Adapter pattern, the Decorator pattern, and so on [37, 80].

Learning the Mechanics for Managing Objects and Associated Constructs

Learning the mechanics includes learning to use keywords, structural mechanisms, and conventions for declaring classes and extending classes; allocating and using objects; and everything else that follows. For example, this learning includes accessing the invoking object (this) within a method, overriding an inherited method, accessing an overridden method, and chaining constructors.

As these constructs are studied, complications arise as structural inevitabilities. For instance, from inheritance comes the question of how to treat the private members of a superclass within a subclass—this need gives rise to the protected access status. From the object idea of factoring, comes the construct of an abstract class and a slew of rules. Other issues that are inextricably bound-up with the nature of objects are the notions of object identity versus object equality (e.g. == versus equals ()).

The complications above tend to transcend specific languages, but different languages may exclude certain conceptual possibilities (e.g. multiple inheritances), introduce others (e.g. interfaces), and handle elements in distinctive fashions (e.g. the shadowing of fields). In Java all objects are dynamic and all methods are virtual. In C++ primitives do not need wrapper classes, but the language needs templates because C++ lacks a subsuming object hierarchy.

Obtaining Hand-On Experience and Practice

Hands-on experience and practice is less generous because object-oriented programs require more scaffolding: Many lines of code are needed to set things up and see them work. Experimenting with abstract classes and constructor chaining requires a much more elaborate test bed than experimenting with for loops.

Learning the Diagrammatic Tools for Conveying Object Design

Today the superceding and universally used diagrammatic tool is the Unified Modeling Language. This language is extremely expressive but very complicated. While a flow chart is intuitively obvious, as are hierarchical input and output diagrams, the Unified Modeling Language is anything but. Whole professional books are devoted to the language. One cannot be a well-equipped object analyst and designer without knowing how to read and write elements including use case, activity, and interaction diagrams, as well as sequence, class, state, and deployment diagrams.

Applying Advanced Programming Constructs

Apart from the objects themselves, applications require more features, and languages offer more constructs than ever before. One cannot go very far in Java without implementing a graphical user interface, catching exceptions, using threads, writing client-server programs, and the like. There is more to learn than ever before.

5. PROPOSED COGNITIVE MODEL

The objective of this model illustrated in Figure 1 is to present a larger overview of the proposed approach to combating cognitive overhead. The top two boxes show the development of one skill set: that relating directly to programming (or developing the problem solving and coding ability to implement algorithms). The bottom two boxes show the development of the independent skill set relating to "object" think analysis and design. The box on the far right represents the fusion of these skill sets into a whole that is greater than the sum of its parts.

Relative to the implications of the research, it is clear that when new content comes into a course that had been already full, something has to come out. So far, we (collectively) have not decided what has to be removed and where it will be re-housed.

This model suggests that two independent skill sets be learned concurrently. The object aficionados insist on learning objects first. The bottom-up contingent asserts learning programming problem solving is more fundamental than learning about the packaging of logic into objects. This typology supports both these views in that skill set one, programming logic, will implement a procedural approach, and skill set two, advanced programming, will implement a top-down approach. In addition, the model identifies exactly what objects are adding to the curriculum and, by so doing, provides the basis for concrete, reasoned, productive discussion.

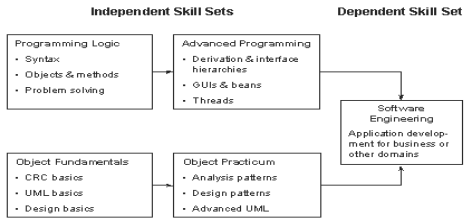
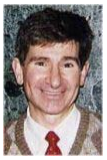


Figure 1: Object-Technology Cognitive Model

6. CONCLUSION

As the information systems community responds to the market's demand for object technologists, many cognitive issues need to be addressed. Faculty members will need to consider issues, such as the current curriculum, book and software adoption, student's skill sets, and faculty development. The results of our qualitative study, from our interviews and questionnaires – indicate that two major challenges software practitioners are confronted with are: understanding objects and understanding how to design. Furthermore, the evidence gathered suggests that programming constructs and design are two independent skill sets that must be learned concurrently to effectively implement and achieve the benefits of object technology. Also, presented were six orthogonal sources of complexity that makes learning to program in the object paradigm difficult. Lastly, a proposed model was presented to combat cognitive overhead. Object technology is an area that requires continual training. Faculty members need the opportunity to explore new pedagogical models to combat the cognitive complexities associated with object technology to meet the demands of teaching object technology courses.

7. AUTHOR BIOGRAPHIES



Allen Stix is an associate professor in the Computer Science Department at Pace University. Among the courses he has taught since joining the computer science faculty in 1982 are software engineering, algorithms and data structures, artificial intelligence, compiler design, and both fundamental and advanced programming. He has written articles on C++, Java, viruses, and graph theory; and, with Susan M. Merritt, has written the textbook Migrating from Pascal to C++ (Springer-Verlag). He holds a Ph.D. from the University of Pittsburgh where he studied mathematical

modeling.

Pauline Mosley is a lecturer at Pace University., she received her D.P.S. in Computing from Pace University. She is the recipient of the Award for Teaching Excellence from Who's Who Among America's Teachers. Her research interest includes cognitive factors that are likely to impact object technology adoption or implementation and evaluating the effects of object technology on software engineering decision outcomes and processes.

8. REFERENCES

Ackerman, P.L., Sternberg, R.J., and Glaser, R. (1989) *Learning and Individual Differences*, (Freeman: New York, New York).

Arnow, D. and Weiss, G. (2002) *Introduction to Programming Using Java: An Object-Oriented Approach,* (Addison-Wesley: Reading: Massachusetts).

Baldwin, C.Y., and Clark, K.B. (1997) "Managing in an Age of Modularity", *Harvard Business Review*, vol. 75, no.2, 84-93.

Beaubouef, T., Lucas, R., and Howatt, J. "The UNLOCK System: Enhancing Problem Solving Skills in CS-1 Students", *ITICSE 2000 Working Group Reports,* vol. 33, no.2, June, 43 –46.

Beck, K. (2000) *Extreme Programming Explained Embrace Change*, (Addison-Wesley: Reading, Massachusetts), 177-179.

Bellin, D., and Simone, S.S. (1997) *The CRC Card Book* (Addison-Wesley: Reading Massachusetts), 1-10.

Berard, E. (1993) *Essays on Object-Oriented Software Engineering* (Addison-Wesley).

Bergin, J. (2000) Student Design Spring. Online Internet. http://csis.pace.edu/~bergin/PedPat1.3.html

Bergin, J. Online. Internet. "Fourteen Pedagogical Patterns", Pedagogical Patterns Project: http;//www-lifia.info.unlp.edu.ar/ppp/

Bergin, J., McNally M., Goldweber M., Hartley S., Kelemn C., Naps T., and Power C. (2000) "Non-Programming Resources for an Introduction to CS: A Collection of resources for the first courses in Computer Science", "*SIGCSE,* Vol. 33, #2, June 89-100.

Bergin, J., Stehlik, M., Roberts, J., and Pattis, R. (1997) *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming,* (John Wiley and Sons, Inc.: New York).

Booch, G. (1994) *Object-Oriented Systems Analysis and Design* (Addison-Wesley: Reading Maccachusetts).

Brooks, F.P.(1987) "No Silver Bullet: Essence and Accidents of Software Engineering," in IEEE Computer 20, 4, 10-19.

Brooks, R. (1983) "Towards a theory of the comprehension of Computer Programs", *International Journal of Man-Machine Studies*, no. 18, 543-554.

Cackowski, D., Najdawi, M., and Chung, Q.B. (2000) "Object Analysis in Organizational Design: A Solution for Matrix Organizations", *Project Management Journal*, vol. 31, no. 3, 44-51.

Caputo, K. (1998) *CMM Implementation Guide: Choreographing Software Process Improvement,* (Addison-Wesley: Reading, Massachusetts).

Carbone, A., Hurst, J., Mitchell, I., and Gunstone, D. "Characteristics of Programming Exercises that Lead to Poor Learning Tendencies: Part II", *The Sixth Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, vol 33, no.3, 93-96.

Coad, P., and Yourdan E. (1991) *Object-Oriented Analysis (*Prentice-Hall: New York, New York), second edition.

Coad, P., and Yourdan E. (1991) *Object-Oriented Design (*Prentice-Hall: New York, New York), first edition.

Cockburn, A. (2000) "Selecting a Project's Methodology", *IEEE Software",* July/August, 64-70.

Cornelius, B. (2001) *Understanding Java,* (Addison-Wesley: England)

Daly J., Brooks A., Miller J., Roper M., and Wood M. (1996) "Evaluating Inheritance Depth on the Maintability of Object-Oriented Software", *Empirical Software End., An International Journal*, vol. 1, no. 2, 109-132.

Decker, R. (1993) "Top-Down Teaching: Object-Oriented Programming in CS 1", *Association of Computing Machinery, SIGSCE,* vol.1.

Deitel and Deitel. (1994) *C++ How To Program* (Prentice Hall: New York, New York), 1-4.

D'Souza, D. (1997) "Objects: Education vs Training", *ICON Computing*, http://www.iconcomp.com/papers

Eisenhardt, K.M. (1989) "Building Theories From Case Study Research", *Academy of Management Review*, vol. 14, no. 4, 432-450.

El-Najdawi, M.K., and Liberatore, M.J. (1997) "Matrix Management Effectiveness: An Update for Research and Engineering Organizations", *Project Management Journal*, vol. 28, no. 1, 25-31.

Entwistle, N.J., and Ramsden, P. (1983) *Understanding Student Learning,* (Croom Helm: London).

Eyring, J.D., Johnson, D.S., and Francis, D.J. (1993) "A Cross-Level Units-Of-Analysis Approach to Individual Differences in Skill Acquisition", *Journal of Applied Psychology*, vol. 78, 805-814.

Fichman, R. and Kemerer C.K. (1997) "Object Technology and Reuse: Lessons from the Early Adopters", *Computer,* vol. 30, no.10, 47-59.

Fisher, S.L., and Ford, J.K. (1998) "Differential Effects of Learner Effort and Goal Orientation on Two learning Outcomes", *Personnel Psychology*, vol. 51, 397-420.

Gamma, E. (1995) *Design Patterns* (Addison-Wesley).

Glaser, B.G. and Strauss, A.L. (1967) *The Discovery of Grounded Theory: Strategies for Qualitative Research*, (Aldine: New York).

Gibbon, C.A., and Higgins, C.A. (1996) "Towards a Learner-Centered Approach to Teaching Object-Oriented Design", *The Proceedings of the third Asia-Pacific Software Engineering Conference*, p.110- 117.

Gora, M. (1996) "Object-Oriented Analysis and Design: The Good, the Bard, and the Ugly of OOAD Methodologies, and Various Approached to Using Them", *DBMS*, May, 4-19.

Holmberg, L. and Mathiassen L. (2001) "Survival Patterns in Fast-Moving Software Organizations", *Focus*,

Holt, R. (1994) "Introducing Undergraduates to Object Orientation Using the Turing Language", *SIGCSE",* vol. 26 no. 1, 324-328.

Isoda, S. (1995) "Experiences of a Software Reuse Project", *J. System and Software*, vol. 30, no. 3, 171-186.

Kaplan, B., and Maxwell, J.A. (1994) "Qualitative Research Methods for Evaluating Computer Information Systems", *Thousand Oaks, CA Sage*.

Kamin, S., Mickunas, M., and Reingold, E., (2002) *An Introduction to Computer Science Using Java,* (McGraw-Hill: New York, New York).

Kendall, K. (1999) *Emerging Information Technologies* (Sage Publications: Thousand Oaks, California), 2-3.

Koffman, E. and Wolz, U. (1999) *Problem Solving with Java*, (Addison-Wesley: Reading, Massachusetts).

Kolling M., and Tosenberg, J. (2001) "Guidelines for Teaching Object Orientation with Java". *Proceedings of the Sixth Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education,* 33-36.

Lang, J., and Bogovich, B. (2001) "Object-Oriented Programming and Design Patterns", *ITICSE 2001 Working Group Reports,* vol. 33, no.4, 68-70.

Lewis, J. and Loftus, W. (1998) *Java Software Solutions: Foundations of Program Design,* (Addison-Wesley: Reading, Massachusetts).

Liang, Y. Daniel (2001) *Introduction to Java Programming- Third Edition,* (Prentice Hall: Englewood Cliffs, New Jersey).

Martin, J. and Odell, J. (1995) *Object-Oriented Methods: A Foundation,* (Prentice Hall: Englewood Cliffs, New Jersey).

McHardy, and Gordon, R. (1999) "Bottom-up or Top-down: A Comparison of Two Methods for Teaching a Method Skill", *Dissertation Abstracts Online*, MAI, 38, no.2, 322.

Meyer, B. "The Reusability Challenge", *Interactive Software Engineering*, vol. 29, no. 2, February, 76-78.

Miles, M.B., and Huberman, A.M. (1994) *Qualitative Data Analysis: An Expanded Sourcebook,* (Sage: Thousand Oaks, CA).

Morisio M., Ezran M., and Tully C. "Success and Failure Factors in Software Reuse", *IEEE Transactions on Software Engineering*, vol. 28, no. 4, 340-356.

Myers, M.D. (1997) "Qualitative Research in Information Systems", *Management Information Systems Quarterly,* vol. 21, no. 2, 241-242.

Orlikowski, W.J. (1993) "CASE Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development", *Management Information Systems Quarterly*, vol. 17, no.3 , 309-340.

Orlikowski, W.J. and Baroudi, J.J. (1991) "Studying Information Technology in Organizations: Research Approaches and Assumptions", *Information Systems Research,* vol. 2, no. 1, 1-28.

Pennington, N. (1987) "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs", *Cognitive Psychology*, no. 19, 295-341.

Pintrich, P.R., and Garcia, T. (1993) "Intra-individual Differences in Students' Motivation and Self-regulated Learning, *Zeitschrift fur Padagogische Psychologie,* vol.7, 99-107.

Poulin, J. (1999) "Reuse: Been There, Done That", *Communications of the ACM*, vol. 42, no.5, 98-100.

Pressman, R. (1997) *Software Engineering A Practitioner's Approach*, (McGraw-Hill: New York, New York).

Reid, R. (1993) "The Object-Oriented Paradigm in CS1", *SIGCSE*, Vol. 25, #1, 265-269.

Rentsch, T. (1982) "Object-Oriented Programming", *SIGPLAN Notices,* vol. 17, 12.

Shalloway, A. and Trott J. (2002) *Design Patterns Explained A New Perspective on Object-Oriented Design* (Addison-Wesley: New York), 3-20.

Sheetz, S.D., Irwin, G., Tegarden, D.P., Nelson, H.J., and Monarchi, D.E. (1997) "Exploring the Difficulties of Learning Object-Oriented Techniques", *Journal of Management Information Systems*, no. 14, 2.

Sleeman, Derek. (1986) "The Challenges of teaching computer programming", *Communications of the ACM*, vol. 29, issue 9, 840-841.

Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. (1982) "What do novices know about programming?", *Directions in Human Computer Interaction*, 27-54.

Staugaard, A. (1999) *Java For Computer Information Systems,* (Prentice Hall: Upper Saddle River, New Jersey).

Tilley, S.R., Paul S., and Smith, D.R. (1996) "Program Comprehension", *Proceedings of the Fourth Workshop on Program Comprehension,* IEEE Computer Society, Los Alamitos, CA, 19-28.

Tracz. W. (1995) "Confession of a Used-Program Salesman: Lessons Learnt", *Proceedings Symposium Software Reliability,* 11-13.

Van Hillegersberg, J., Kumar K., and Welke R. (1995) "An Empirical Analysis of Performance and Strategies of Programmers New to Object-Oriented Techniques", *Proceedings of the Seventh Workshop Psychology of Programming Interest Group,* January.

Warr, P.B., and Allan, C. (1998) "Learning Strategies and Occupational Training", *International Review of industrial and Organizational Psychology,* vol. 13, 83-121.

Warr, P.B., and Bunce, D.J. (1995) "Trainee characteristics and the outcomes of open learning", *Personnel Psychology,* vol.48, 347-375.

Warr, P.B., and Downing, J. (2000) "Learning Strategies, Learning Anxiety and Knowledge Acquistion, *British Journal of Psychology*, vol. 91, 311-333.

Webster, B. (1995) *Pitfalls of Object-Oriented Development* (M & T Books: New York, New York), 52-55.

Weiss, D.M. and Lai, C.T.R. (1999) *Software Product-Line Engineering: A Family-Based Software Development Approach,* (Addison-Wesley).

Westfall, R. (2001) "Hello, World Considered Harmful", *Communications of the ACM*, vol.44, no.10, 129-130.

Wieringa, R. and White. (1998) "A Survey and Object-Oriented Software Specification Methods and Techniques", *Communications of the ACM Computing Surveys,* vol. 30, no.4.

Yin, R.K. (1989) *Case Study Research Design and Methods,* (Sage: Thousand Oaks, CA).