Communications of the Association for Information Systems

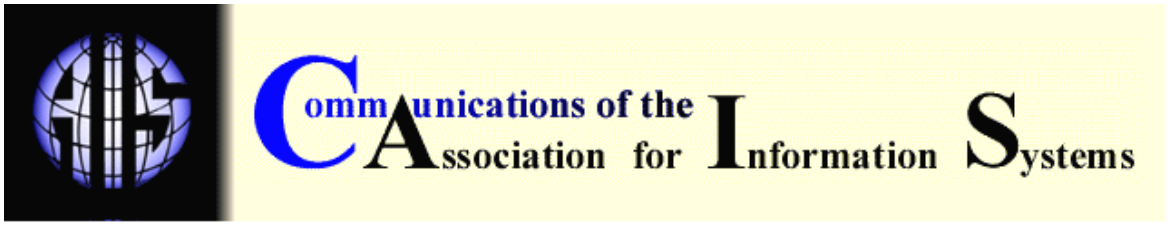
Volume 8 Article 4

January 2002

# Object-Oriented Systems Development: A Review of Empirical Research

Richard A. Johnson

*Southwest Missouri State University*, richardjohnson@smsu.edu




# OBJECT-ORIENTED SYSTEMS DEVELOPMENT: A REVIEW OF EMPIRICAL RESEARCH

**Richard A. Johnson**
*College of Business Administration*
*Southwest Missouri State University*
richardjohnson@smsu.edu

**ABSTRACT**

Object-oriented systems development (OOSD) is viewed by many as the best available solution to the ongoing "software crisis." However, some caution that OOSD is so complex that it may never become a mainstream methodology. To settle the controversy requires high-quality empirical evidence. This paper surveys the most rigorous research on OOSD available over the past decade. A review of these empirical studies indicates that the weight of the evidence tends to slightly favor OOSD, although most studies fail to build on a theoretical foundation, many suffer from inadequate experimental designs, and some draw highly questionable conclusions from the evidence. This set of conditions points to the need for additional, higher quality research to build a better case either for or against OOSD.

**KEYWORDS:** object-oriented, object-orientation, systems development, empirical research, methodology

## I. INTRODUCTION

Object technology (OT) is a recently emerging branch of information technology of which object-oriented systems development (OOSD) is an extremely vital element. Many practitioners (Booch, 1994; Coad and Yourdon, 1991; Coleman, Arnold, Bodoff, Dollin, Gilchrist, Hayes, and Jeremaes, 1994; Jacobson, Christerson, Jonsson, and Overgaard, 1995; Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen, 1991) believe OOSD to be far superior to conventional systems development (CSD). OOSD is viewed so highly in some circles that it has been elevated to the "unified software development process" (Jacobson, Booch, and Rumbaugh, 1999).

The advocates of OOSD claim many advantages including easier modeling, increased code reuse, higher system quality, and easier maintenance. However, some express serious concern about certain disadvantages of OOSD, such as its difficulty to learn, slower development time, and poorer run-time performance (Pancake, 1995; Fichman and Kemerer, 1993). While the OO paradigm appears full of promise, many developers seem reluctant to accept it wholeheartedly, especially within the business community (Pancake, 1995). In the case of OOSD, some say that "technology adoption is mostly the result of marketing forces, not scientific

evidence" (Briand, Arisholm, Counsell, Houdek, and Thevenod-Fosse, 1999, p. 388). As Smith and McKeen (1996) have observed, OT is "still long on hype and short on results . . ." (p. 28).

With the large amount of hype surrounding OOSD, it seems reasonable to search for hard core, empirical evidence before committing scarce corporate resources to its large-scale adoption. While such evidence is generally lacking (Briand et al., 1999), some research on OOSD has been reported over the past decade. The purpose of this paper is to present and examine the methodologies and conclusions of some of the most meaningful empirical studies on the pros and cons of OOSD (Section III). After these studies have been reviewed, implications of the findings are discussed and possible future research is proposed, including a framework to guide researchers. The results of this study should prove beneficial to practitioners who are considering the adoption of OOSD and to researchers who are considering further exploration of this important new paradigm of systems development.

## II. BACKGROUND

### WHAT IS OOSD?

OOSD is generally considered a revolutionary, rather than evolutionary, advancement in systems development (Hardgrave, 1997) primarily because of its radical departure from conventional systems development (CSD) (Fichman and Kemerer, 1992). While CSD is based on decomposing a system into procedures (process oriented) or data (data oriented), OOSD is predicated on decomposing a problem into interacting objects that encapsulate both data and behavior (Booch, 1994). An object's *attributes* are captured in its data structure. An object's *behavior* (also known as functions, procedures, or operations) is actuated when it receives *messages* (function calls) from other objects. The object sending the message need only know *what* is to be done, not *how* it is done. The object receiving the message contains the implementation details. The encapsulation of data and behavior into a single entity (i.e., the object) is germane to providing many benefits over CSD.

Just as CSD consists of the phases of analysis, design, and implementation (primarily programming), OOSD involves OO analysis (OOA), OO design (OOD), and OO programming (OOP). Throughout the 1990's, an abundance of OOA and OOD methods emerged (e.g., Booch, 1994; Coad and Yourdon, 1991; Coleman et al., 1994; Jacobson et al., 1992; Rumbaugh et al., 1991; Shlaer and Mellor, 1992; Wirfs-Brock and Johnson, 1990), eventually culminating in a standard known as the Unified Modeling Language (UML) (Rumbaugh et al., 1998). Adding to the multiplicity of methods, there are almost as many OOP languages (e.g., C++, Smalltalk, Java, and the next version of Visual Basic). All phases of OOSD are founded on what is called the OO paradigm (Korson and McGregor, 1990) consisting of a set of five basic precepts: (1) object, (2) class, (3) inheritance, (4) polymorphism and (5) dynamic binding. In-depth discussions of OO concepts can be found in Booch (1994), Coad and Yourdon (1991), and Rumbaugh et al. (1991), among others.

### ADVANTAGES AND DISADVANTAGES OF USING OOSD

The application of OOSD is usually expected to result in many distinct advantages over CSD. However, disadvantages associated with OOSD are also reported. A fairly comprehensive list of these advantages and disadvantages, as reported by both experienced and novice OO developers, was compiled by the author from previous research (Johnson, 2000) and is presented in Table 1 in priority rank.

Table 1. Reported OOSD Advantages and Disadvantages

| Priority Rank | Advantages | Disadvantages |
|---|---|---|
| 1 | An easier modeling process | Decreased system run-time performance |
| 2 | Improved modularity of systems | Unavailability of adequate OO DBMS's |
| 3 | Improved maintainability of systems | Increased initial development time |
| 4 | Improved quality of systems | Unavailability of OO CASE tools |
| 5 | More understandable A&D models | Confusion of too many OOA/D methods |
| 6 | Greater stability of designs over time | Inability to try OOSD before committing |
| 7 | More flexible/adaptable development | Complexity of OOA/D methods |
| 8 | An easier transition between phases | Complexity of OOP languages |
| 9 | More effective code reuse | Incompatibility of OOSD with processes |
| 10 | Improved communication with developers | Inability to demonstrate OOSD benefits |
| 11 | Improved productivity of your work | Difficulty learning OOA/D methods |
| 12 | More effective A&D model reuse | A more difficult programming process |
| 13 | Greater user satisfaction with systems | Difficulty learning OO programming |
| 14 | Improved communication with users | Unsuitability of OOSD for projects |

A review of some of the most recent and important empirical research into these advantages and disadvantages follows. The discussion is organized around the particular development phase (analysis, design, or programming) under investigation.

## III. SELECTED EMPIRICAL STUDIES ON OOSD

### RESEARCH ON OO ANALYSIS AND DESIGN

As evidenced by the reported OOSD advantages and disadvantages in Table 1, it is apparent that most of the emphasis in the OO literature is placed on the analysis and design of OO systems. This emphasis is to be expected since proper analysis and design are generally considered cornerstones of system success (Brooks, 1987). Much of the empirical research on OOSD focuses on these two phases of development.

*Boehm-Davis and Ross* (1992) compared the quality of designs and solutions for various projects using three different types of systems development methodologies:

- procedural,
- data-oriented (Jackson System Development, or JSD), and
- object-oriented.

The eighteen subjects in this study were professional programmers divided into three groups of six. Each group received training and/or had previous experience in one of the three different methodologies. The subjects were asked to provide designs and write pseudo-code for three different systems. Data were collected on solution completeness, time to design and code, and solution complexity. The findings show that the JSD and OO groups

- generated significantly more complete solutions,

- required significantly less development time, and
- produced less complex solutions

than the procedural group. The accomplishments of the OO group look even more impressive given that the JSD group had three to four times more overall development experience and more than twice the experience with the JSD methodology compared to either the procedural or OO group. Thus, one can conclude that for professional developers, OO designs and solutions are of higher quality and take less time than procedural designs and solutions. While the results look somewhat encouraging for OO, the experimental design of the research has some serious limitations, primarily because it did not control for the prior level of experience across methodologies.

*Vessey and Conger* (1994) also compared the same three analysis methods: process-oriented (structured), data-oriented (Jackson System Development), and object-oriented (Booch). Six software engineering students, inexperienced in any analysis method, received the same training in all three methods during a university course. They were then assigned to one of three groups (two students per group) and given equally complex analysis problems to solve using one of the three methods. The researchers performed a protocol analysis and determined that novice analysts found OOA more difficult to learn and apply than data-oriented analysis and data-oriented analysis more difficult to learn and apply than process-oriented analysis. This study seemingly contradicts the previous study's findings that OO is easier to apply (Boehm-Davis and Ross, 1992). While the methods used by developers in the Vessey and Conger (1994) study were almost identical to those in the Boehm-Davis and Ross (1992) study, Vessey and Conger used students instead of experienced developers and used a much smaller sample size (n=6 vs. n=18). Also, the students were not randomly assigned to groups.

*Pennington, Lee and Rehder* (1995) performed a protocol analysis on a total of ten experienced, professional developers. Three were expert procedural developers, four were expert OO developers, and three were novice OO developers (who were, however, expert procedural developers). All three groups were given a relatively simple swim meet scoring problem and asked to create a complete design using their respective methods. Completed designs were judged in terms of quality while developers were evaluated on productivity. The results showed that the designs of the OO experts were more complete but took more time compared to the procedural experts. Even though they took more time, the OO experts were graded more efficient than the procedural experts when overall design quality was considered. The study concludes that OO designs are of higher quality than procedural designs and take less time to complete.

*Hardgrave and Dalal* (1995) performed a laboratory study of 56 advanced undergraduate MIS majors, all enrolled in a senior-level DBMS course, to compare two competing data modeling techniques: the extended entity-relationship (EER) model (McFadden and Hoffer, 1991) and the Object Modeling Technique (OMT) of Rumbaugh et al. (1991). Although not explicitly identified in the paper, the study appears to be based on the cognitive problem-solving theory of Newell and Simon (1972) positing that characteristics of the task environment and characteristics of the developer jointly influence problem-solving performance. The independent (task) variables in this study were modeling technique (OMT or EER) and complexity of the resulting model (developer characteristics were not measured). The students were randomly assigned to one of four groups, with each group given a previously prepared, completed model to review (simple OMT, complex OMT, simple EER, and complex EER). The students in each group, who had already received training in the techniques, were provided with two additional one-hour lectures specifically on their respective models. They were then asked to take a test on their understanding of the models and complete a follow-up questionnaire. The dependent variables were

- level of understanding (measured by the score on the test),
- time to understand (measured by the time to complete the test), and
- perceived ease-of-use (measured by item scores on a questionnaire).

The results indicated that, for both simple and complex systems, OMT models were more quickly understood than EER models. However, no significant difference was found for the depth of understanding and the perceived ease of use of the two methods, regardless of task complexity. Thus, OO modeling techniques may be understood more quickly but not more completely compared to data-oriented techniques. One possible shortcoming of this study is that it compares

object-oriented to data-oriented modeling techniques. These two methods are much more closely related than object-oriented and process-oriented techniques, so differences in understanding or perceived ease of use may be difficult to detect, and even if detected, less relevant to the concerns of many practitioners and researchers.

*Wang* (1996a) performed an experiment using thirty-two undergraduate students with no previous systems analysis training or experience. The subjects were randomly divided into two groups. One group was trained for five hours on the data flow diagram (DFD) method, while the other group was trained for five hours on an object-oriented analysis method. The subjects were then presented with a mini-case in management information systems analysis. The OO group spent significantly less time on their analyses of the problem and created solutions that were significantly more accurate. After completing the analysis, the subjects responded to a questionnaire concerning their perceptions of the analysis method used. The OO group reported that the OOA method was easier to learn and understand. The OOA method was also rated superior overall. This study confirms the results of several previously cited studies: OOA produces higher quality models more quickly than procedural analysis.

In a separate study, Wang (1996b) again compared a structured method of analysis (DFD) with object-oriented analysis (OOA) using two groups of inexperienced undergraduate MIS majors. Students were randomly assigned to two groups, 24 in the DFD group and 20 in the OOA group. Each participant learned his respective analysis method and created analysis diagrams based on information in a mini-case study. The total time allowed for training and problem solving was 7.5 hours spanning several class sessions. The two dependent variables were the syntactic and semantic accuracy (in conveying system requirements) of the resulting analysis diagrams. Using ANOVA techniques, the results indicated that the syntactic accuracy for the DFD group was significantly greater in the early sessions, but that syntactic accuracy for the OOA group was significantly greater in the last session. However, there was no significant difference in semantic accuracy for the DFD and the OOA groups. Apparently contradicting the results of his own previous study (Wang 1996a), this experiment concludes that OOA appears more difficult to learn than DFD, and that OOA does not produce solutions of higher quality.

*Herbsleb, Klein, Olson, Brunner, Olson, and Harding (1995).* Another important benefit claimed for OOA and OOD is improved communication among development team members, as well as between users and developers (Garceau, Jancura, and Kneiss, 1993). The assumption is that OO is easier to understand, but it is not clear whether increased understanding should lead to increased or decreased communication. No empirical research was found on improved communication between users and developers, but Herbsleb, Klein, Olson, Brunner, Olson, and Harding (1995) focused on developer interaction during the design phase of OO projects. In this research, several field studies were conducted using developers' time sheets, videotapes of meetings on design activities, and semi-structured interviews with developers. Results indicated that when OOD methods are used, fewer spontaneous episodes of clarification occur. Also, planned summaries and walkthroughs occur much more often when using OOD. More attention was given to the reasons for specific design choices for the OO projects. OOD seems to encourage a deeper inquiry into the reasons underlying design decisions but less inquiry into the requirements. The authors of this paper believe these findings indicate improved communication in software development teams, which leads to greater understanding of requirements.

However, there may be alternative explanations. For example, fewer spontaneous episodes of clarification could occur if developers wish to disguise a lack of understanding. The increased number of planned summaries and walk-through's could result if developers perceive a lack of understanding among peers. Thus, the study may indicate that OOD decreases one form of communication and increases another simply because OOD is new and/or more difficult to understand, not because it is easier or more natural.

*Davies, Gilmore, and Green* (1995). Supporters of OOSD claim that thinking in terms of interacting objects, rather than in terms of functions or procedures, should be more natural to humans (Pancake, 1995). Davies, Gilmore, and Green (1995) set out to test the claim that OO decomposition of the problem domain is more natural to the ways of human cognition than functional decomposition. Twelve expert and twelve novice programmers were presented with cards containing fragments of code from a large C++ library for graphics applications. Their task was to sort the cards according to any criteria they felt appropriate. The purpose of the sort was

to determine whether the subjects would perform functional or object-oriented decompositions of the problem domain. Subjects performed the sorting of code fragments and reported the reasons for their sorting (categorized as either function-based or object-based). The results showed that expert subjects seemed to focus more on the functional properties of the code while the novice subjects tended to classify the code fragments according to important features of the OO paradigm (class membership, object similarity, or inheritance relations). According to the authors, the "results appear to suggest fairly clearly that functional information is of much greater importance to experts than is information about objects and their relations" (p. 242). The implication is that OO decomposition is not more natural for expert developers, as was expected by the researchers. Of course, an alternative explanation is that experts are simply more experienced with functional decomposition and tended to see the code fragments in that way.

*Agarwal, Sinha, and Tanniru* (1996) performed a thorough experiment comparing the ability of novice analysts to perform a requirements analysis using either a process-oriented (PO) or an object-oriented (OO) analysis methodology. This study explicitly identifies the Newell and Simon (1972) problem-solving theory as its basis, extending it with the concept of "cognitive fit" (Vessey, 1991), which implies that a closer fit between the nature of the task and the way it is represented in the problem space should improve problem-solving performance. A total of 43 undergraduate students (with no prior training or experience in any type of systems analysis) were randomly divided into two groups: a PO group (n=24) and an OO group (n=19). Each group was trained six hours in its respective analysis methodology—the DeMarco (1978) method for the PO group and the Coad and Yourdon (1991) method for the OO group. Individuals in each group were then presented with two problems to analyze—one problem was clearly more function-strong (PO) while the other was more structure-strong (OO). According to the theory of cognitive fit, the PO group should perform better on the PO problem, while the OO group should perform better on the OO problem. The researchers found that the PO group had significantly better overall performance than the OO group on the PO task, but that there was no difference in overall performance between the two groups on the OO task. The researchers concluded that PO methodologies should be easier for novices to learn than OO methodologies, possibly because people may have a greater tendency to reason procedurally.

*Morris, Speier, and Hoffer* (1999) again put the issue of prior experience to the test. As in the Agarwal et al. (1996) study, the Newell and Simon (1972) problem-solving theory served as the theoretical backdrop. Seventy-one student subjects, 34 of whom were procedurally experienced (juniors and seniors in a SA&D course) and 37 of whom were novices (freshmen and sophomores in an introduction to computing course), received brief training in DFD and OOA (Coad and Yourdon, 1991) methods. They were then asked to develop models for two simple tasks. The dependent variables under examination included (1) subjective mental workload, a self-reported assessment of the cognitive effort utilized that is based on a previously validated instrument, (2) solution quality, (3) time to solution, and (4) attitudinal measures of confidence, preference, and ease of use. Both groups of subjects reported that the subjective mental workload for OOA was significantly higher than for DFD, indicating that OOA would be more difficult to learn and/or use. There was no significant difference in the time to solution for either group, or for the solution quality for the novice group. The procedurally experienced group created a DFD solution that was of higher quality than the OOA solution. This outcome is not surprising since many believe that procedural methods must be unlearned before OO methods can be learned (Pancake, 1995). A somewhat surprising result was that the novice group judged OOA easier to use and preferable to DFD, even though the subjective mental workload measure was higher for OOA. This finding could imply that while OOA may be more challenging, it is also more intellectually stimulating to novices.

## RESEARCH ON OO PROGRAMMING

Many advocates of OOSD claim that it enhances both productivity and quality, primarily because of increases in model and code reuse. While no empirical studies on model reuse were found, several studies did explore the issue of code reuse.

*Lewis, Henry, Kafura, and Schulman* (1992) tested hypotheses related to code reuse in a controlled experiment with 21 senior software engineering students. Business applications were created using either a procedural language (Pascal) or an object-oriented language (C++). They

determined that using the OO paradigm resulted in higher levels of reuse and improved programmer productivity.

*Chen and Chen* (1994) performed a laboratory experiment to compare three approaches to programming: (1) the traditional method of coding a system from scratch, (2) an OO method employing components of reusable C++ code, and (3) reusable design frameworks. A framework is a completed OO design of a subsystem, consisting of concrete, collaborating classes that are subsequently catalogued into a library (Gamma, 1995). Three teams of graduate students (two groups of seven and one of six) were given the same development task, a window manager system. Each group was assigned one of the three methods to code the problem solution. It was determined that programmers were more productive and produced higher quality systems when provided reusable components and frameworks than when building a system from scratch. However, the results may be questioned as it appears the teams employing reuse were provided with detailed information on the availability of the components that were previously tailor-made for the target system. Also, students were not randomly assigned to different experimental groups.

*Basili, Briand, and Melo* (1996), in a fairly rigorous study, tested the hypotheses that higher rates of code reuse result in (1) higher software quality, (2) lower software maintenance, and (3) higher developer productivity. Eight development teams, each containing three randomly assigned computer-science graduate students, were given the task of developing a complete management information system for a hypothetical video rental business. OMT (Rumbaugh et al., 1991) was used for analysis and design and C++ was used as the implementation language. The eight teams were provided with equal training in the use of various types of C++ libraries for creating and applying GUI's, functions, and databases. Thorough measurements were taken on the amount of code reuse and the overall progress of all eight teams. Statistically significant results were obtained supporting all three hypotheses.

*Harrison, Samaraweera, Dobie, and Lewis* (1996) compared a functional language (SML) to an object-oriented language (C++) in a controlled experiment using several different metrics. One developer, who was equally experienced in both languages, was asked to create a total of twelve different algorithms using both SML and C++. The statistically significant conclusions included (1) the procedural language required twice the testing time, (2) the procedural language resulted in 2.5 times the error rate, and (3) the procedural language resulted in 1.5 times as much reuse as the OO language. There were no significant differences in modification requests, modification times, and total development time. These results confirm some of the expected advantages of OOP but contradict others. Obviously, a serious limitation of this study is the use of only one developer (a sample size of one).

*Wood, Daly, Miller, and Roper* (1999). While reuse is generally expected to improve programmer productivity and system quality, other aspects of the OO paradigm might negatively impact such measures. For example, increasing levels of inheritance may exacerbate, rather than mitigate, maintenance efforts. Wood, Daly, Miller, and Roper (1999) performed a series of well-designed experiments on student subjects to determine if the number of inheritance levels incorporated in two different OO database applications (one relatively simple and one somewhat more complex) affected the amount of time required to add program code for new classes. The results indicated that when three levels of inheritance are involved, maintenance of the system required significantly less time than for a “flat” system (i.e., no inheritance), but when five levels of inheritance are involved, maintenance of the system required significantly more time than for a flat system. Thus, the use of inheritance may improve maintenance efficiency up to a point, but its overuse could degrade maintenance performance. One shortcoming of this study was the failure to provide the subjects with system design documentation to aid in their maintenance efforts.

*Briand, Wust, Daly, and Porter* (2000). In a study dealing with both function calls and inheritance, Briand, Wust, Daly, and Porter (2000) discovered that the frequency of method invocations and the depth of inheritance hierarchies are the major determinants of fault-proneness of resulting software classes. Eight three-person teams of upper division undergraduate students, with no previous OO experience, were taught OOAD. Each team developed a medium-sized MIS for a hypothetical video rental business. The OMT analysis and design method (Rumbaugh et al., 1991) was used with C++ as the implementation language. Independent testers, consisting of experienced software professionals, evaluated the coded classes for faults. Existing measures of coupling (classes using methods or attributes in other

classes), cohesion (methods within a class using common attributes of the class) and inheritance (classes deriving methods from ancestor classes) defined at the class level were used as independent variables to predict the probability of fault-proneness in class code. The classes investigated were either developed from scratch or were extensive modifications of library classes. Univariate analysis showed that increased levels of coupling and inheritance significantly impact fault-proneness of classes while cohesion does not. Multivariate analysis showed that models involving coupling and inheritance measures could be developed to automatically detect faulty classes with an accuracy rate approaching 90%.

## RESEARCH ON ALL PHASES OF OOSD

*Fedorowicz and Villeneuve* (1999). Most empirical research on OOSD focuses on a single phase of the systems development life cycle and on a limited set of concerns (e.g., difficulty of learning or productivity). In a more comprehensive approach, Fedorowicz and Villeneuve (1999) conducted a survey of 228 computer-industry practitioners with interest in commercial OO tools, 70% of whom were at least somewhat experienced with such tools. Overall, respondents

(1) preferred OOAD to traditional analysis and design methods,
(2) found OOSD to be more useful and efficient,
(3) preferred OO for communication with users and team members, and
(4) found that objects are both shareable and reusable.

These attitudes became more substantial with increases in

(1) the number of SDLC steps involved,
(2) project size, and
(3) OO experience,

as well as when OO tools and formal methodologies were used. However, respondents found that it is more difficult to acquire OO skills, but this view becomes less substantial with increasing OO experience, and with the increased use of OO tools and methodologies. They also noted that OOA takes longer than traditional analysis, but that a net savings should result over the entire SDLC. One major limitation of this study is that only sketchy information was reported on the specific type and duration of OO experience of the respondents.

*Johnson* (2000) conducted a somewhat similar survey of 150 randomly selected, experienced developers from across the U.S. in order to encompass all OOSD phases and a very wide range of concerns. A total of 96 of these subjects were seasoned OO developers, and the remaining 54 (henceforth called non-OO developers) were experienced developers trained in OOSD, but with no significant industrial experience on OO projects. The survey included questions concerning the fourteen specific advantages and fourteen specific disadvantages of all phases of OOSD (analysis, design, and programming) listed in Table 1. On a scale from 1 (low) to 7 (high), the OO developers assigned a median rating of 6 for such advantages of OOSD as easier modeling, higher quality systems, improved maintainability, easier transition between development phases, and more effective code reuse. Regarding OOSD disadvantages, the OO developers did not judge any in the list of fourteen (such as increased development time, complexity, difficulty to learn, or unsuitability for projects) to be of concern. On the contrary, OO developers generally believed that OOA, OOD and OOP were *not* difficult to learn. The OO developers were significantly more convinced of the vast majority of OO advantages than the non-OO developers, and significantly less convinced that OOSD is difficult to learn, compared to the non-OO developers. Apparently, actual experience with OOSD may enhance beliefs in the advantages of OOSD and reduce concerns about the disadvantages of OOSD. An alternative explanation is that OOSD may be accompanied by a “halo effect” where those committed to it believe it is indeed a “silver bullet.”

Table 2 summarizes results of the studies cited.

## IV. GENERAL CONCLUSIONS ABOUT OOSD

Eighteen empirical studies, representing some of the best available in the field of OOSD, were reviewed in Section III. These studies span multiple phases of OOSD: analysis (OOA), design (OOD), and programming (OOP). A majority of the studies concentrate on the areas of OOA and OOD where most of the benefits of object technology are expected to accrue (Booch, 1994). In general, the conclusions reached in these studies are somewhat mixed.

- Eight studies (53%) were favorable toward OOSD,
- four (27%) were unfavorable, and
- the remainder (20%) included both positive and negative results.

Interestingly, the same researcher (Wang, 1996a, 1996b), using very similar subjects and tasks, obtained apparently conflicting results.

In nearly every instance where studies were favorable to OOSD, higher system quality and developer productivity were cited as primary benefits. On the other hand, nearly every negative result focused on the difficulty of learning OOSD. These results are consistent with the anecdotal OO literature. Only Johnson (2000) obtained results that consistently claimed OOSD is not more difficult than traditional methods. In any event, the results suggest that while OOSD may be somewhat more difficult to learn than conventional methods of systems development, the effort spent in education and training may ultimately pay off in increased quality and productivity.

Some studies discussed above present mixed results on other important OOSD issues. For example, the OO paradigm was found to be more natural for developers (Davies et al., 1995), although the logical derivation of this conclusion from the data is highly suspect. The conclusion that OOSD enhances communication (Herbsleb et al., 1995) may actually highlight a potential disadvantage of OOSD, i.e., that OOSD may be more confusing, thus necessitating an increased level of communication. Several studies (Lewis et al., 1992; Chen and Chen, 1994; Basili et al., 1996) found that OO code reuse improves productivity and quality, while Harrison et al. (1996) claim that OO programming was accompanied by less reuse. However, the latter study was based on a sample size of only one. Thus, most of the evidence supports the claim that OOSD enhances code reuse.

Upon careful examination of Table 2, nearly all cases where only negative results were obtained stemmed from the use of inexperienced students as subjects. This suggests that proper learning can play a tremendous role in the ultimate effectiveness of OOSD. Students given only a few hours or weeks of training in OOSD should not be expected to perform OO tasks particularly well, especially given that OOSD may be somewhat difficult to learn. The conventional wisdom is that proficiency in OOSD may require six to eighteen months of full-time experience (Fayad, Tsai, and Fulghum, 1996). Thus, many of the negative results could be attributed to the types of subjects chosen and the amount of training provided.

## V. THEORY ISSUES

IS researchers would overwhelmingly agree that theory should form the basis for any serious empirical study. In the case of research into the advantages and disadvantages of OOSD, theory has definitely taken a back seat. Of the eighteen studies cited in this paper, only three explicitly attempt to rely on previously accepted theory (Hardgrave and Dalal (1995), Agarwal et al. (1996), and Morris et al. (1999)). In these cases, the theoretical background was the information-processing model of Newell and Simon (1972). As Figure 1 illustrates, this theory posits that problem-solving performance is directly influenced by the individual's internal representation of the problem, which in turn is influenced jointly by characteristics of the task environment and characteristics of the information processing system (i.e., the problem-solver). For example, the ability to successfully solve a systems development problem may depend on the approach (OO vs. procedural) and the information processing system (experienced vs. inexperienced systems developers) employed. In defense of the other fifteen studies, most appear to be at least implicitly (although sometimes loosely) based on this same problem-solving theory.

Table 2. Summary Of Selected Empirical Research on OOSD

| | Date | Phases Studied | Types of Subjects | System Charac-teristics | n | Dependent Variables | Slant of Results | Major Conclusions |
|---|---|---|---|---|---|---|---|---|
| Boehm-Davis and Ross | 1992 | OOA/ OOD | Experienced developers | Simple /partial | 18 | Quality, productivity | Pro | OOA/D results in higher quality and productivity than procedural analysis/design. |
| Vessey and Conger | 1994 | OOA | Inexperienced students | Simple /partial | 6 | Ease of learning/use | Con | OOA is more difficult to learn and use than procedural analysis. |
| Pennington et al. | 1995 | OOD | Experienced developers | Simple /partial | 10 | Quality, productivity | Pro | OOD results in higher quality and productivity than procedural design. |
| Hardgrave and Dalal | 1995 | OOA | Inexperienced students | Simple /partial | 56 | Ease of learning/use | Pro & con | OOA is more readily understood, but not more completely understood than data-oriented analysis. |
| Wang | 1996a | OOA | Inexperienced students | Simple /partial | 32 | Ease of learning | Pro | OOA is easier to learn and understand than procedural analysis. |
| Wang | 1996b | OOA | Inexperienced students | Simple /partial | 44 | Ease of learning, quality | Con | OOA is more difficult to learn and does not result in higher quality than procedural analysis. |
| Morris et al. | 1999 | OOA | Experienced and inexperienced students | Simple/ partial | 71 | SMW, quality, productivity, ease of learning/use | Pro & con | OOA is more difficult to use; DFD-experienced students produced higher quality OOA models than inexperienced students, inexperienced students preferred OOA over DFD. |
| Lewis et al. | 1992 | OOP | Experienced students | Simple /partial | 21 | Level of reuse, productivity | Pro | OOP results in greater reuse and productivity than procedural programming. |
| Chen and Chen | 1994 | OOP | Experienced students | Simple /complete | 20 | Quality, productivity | Pro | OO reuse results in higher quality and productivity than programming from scratch. |
| Basili et al. (1996) | 1996 | OOP | Experienced students | Simple /complete | 24 | Quality, productivity, maintainability | Pro | Increasing OO reuse results in higher quality and productivity, less maintenance. |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Harrison et al. (1996) | 1996 | OOP | Experienced developer | Simple /partial | 1 | Quality, productivity, reuse | Pro & con | OOP results in less testing time, higher quality, but less reuse. |
| Wood et al. (1999) | 1999 | OOP | Experienced students | Simple /partial | 31 | Maintainability | Pro & con | Low inheritance depth results in less maintenance time, higher depth in more maintenance time. |
| Briand et al. (2000) | 2000 | OOP | Experienced students | Simple/ complete | 18 | Maintainability | Con | Increased levels of coupling and inheritance contribute to fault-proneness of classes. |
| Agarwal et al. (1996) | 1996 | OOA | Inexperienced students | Simple /partial | 43 | Cognitive fit | Con | OOA is more difficult to learn, results in lower quality com-pared to procedural analysis. |
| Davies et al. (1995) | 1995 | OOA | Experienced/ inexperienced developers | Simple /partial | 24 | Cognitive fit | Con | OOA is not more natural to experienced developers. |
| Herbsleb et al. (1995) | 1995 | OOA/ OOD | Experienced developers | Complex/ partial | 12 | Level of communication | Pro | OOA/D results in increased communication among developers. |
| Fedorowicz & Villeneuve (1999) | 1999 | OOA/ OOD/ OOP | Experienced developers | Complex/ complete | 228 | Preference, ease of learning, productivity, communication , reuse | Pro & con | Preference for OOSD increases with project size and OO experience; OO novices find OOSD difficult to learn: OO results in higher productivity, better communication; objects are sharable and reusable. |
| Johnson (2000) | 2000 | OOA/ OOD/ OOP | Experienced developers | Complex/ complete | 150 | Many advantages/ disadvantages | Pro | OOSD is easier to learn and use, very suitable for projects, and results in higher quality and productivity compared to CSD. |

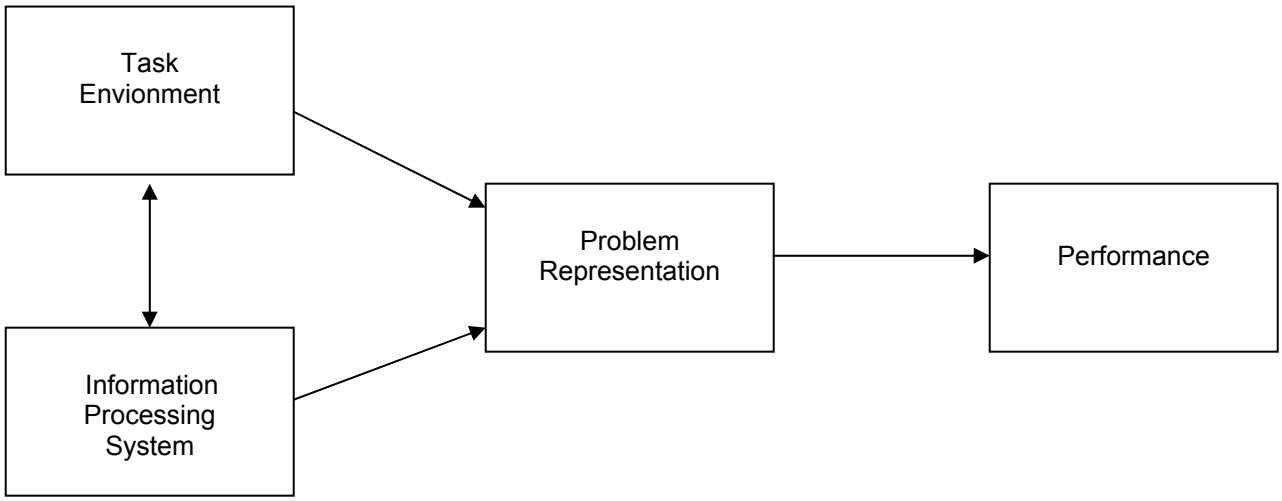



Figure 1. General Theoretical Model (Newell And Simon, 1972)

## VI. METHODOLOGY ISSUES

Table 2 also highlights several possible methodological weaknesses in empirical studies of OOSD. First, many studies are very narrow in scope, focusing on only one phase of the life cycle, and even then comparing only two specific techniques, such as an OO class diagram vs. a data flow diagram. While the resolution of such narrow issues may be necessary and helpful, it does not address the entire system development process and could unfairly skew perceptions either for or against OOSD. For example, OOA may indeed take more time to complete than procedural analysis, supporting a conclusion that analysts are less productive when using OO (Vessey and Conger, 1994). However, more time spent in OOA may result in much less implementation and maintenance time for the completed system. The overall result could be improved productivity using OOSD.

Another major problem with many empirical studies in OOSD is the small sample size and otherwise poor experimental design. Sample sizes of one or two per treatment are very susceptible to attack in terms of both internal and external validity. Such studies may be unable to detect significant differences between procedural and OO methods due to low statistical power. Poor experimental designs that fail to randomly assign subjects to treatments or otherwise fail to control for developer experience would suffer in the ability to generalize to the population of developers.

As discussed earlier, studies often use inexperienced students as subjects. Such practices may be acceptable when the purpose of the research is strictly to explore the difficulty of learning OOSD, but not when research questions focus on the quality and productivity of models or completed systems. Also, the question of learning OOSD may be even more critical to experienced procedural developers who may be forced by management to make the transition to OO in the middle of their careers, but no studies were found that specifically address this group.

Another concern is the use of extremely simple problems in laboratory studies. Results of such studies are hardly generalizable to what are often referred to as “industrial strength” development projects (Booch, 1994). It is possible that process-oriented

methodologies may be superior to OO for simple problems, while the power of the OO approach may be more evident for complex problems.

Finally, potential problems exist with studies that attempt to train novice students quickly in OOSD. Instructors at universities where such studies are conducted are likely to be significantly less experienced in the new OO methodologies than the more established procedural methodologies. This condition could result in less than optimum conditions for effectively and efficiently transferring complex OO knowledge, making it even more difficult for students to learn OO adequately.

## VII. FUTURE RESEARCH

Clearly, more research of higher quality is needed to determine with greater certainty how OOSD compares to CSD. Theory should serve as the cornerstone of every study. Laboratory experiments could be designed to determine how well subjects, especially students, are able to learn and apply the multiple facets of OOSD. Field studies and survey research on experienced developers could explore the transition to OOSD in industry and the effectiveness of OOSD on complex projects. Researchers could also investigate whether learning OOSD is more difficult for novice or experienced developers. The appropriateness of OOSD or CSD approaches for certain types of real-world problems (e.g., simple vs. complex, function-intensive vs. data-intensive) should be addressed. The dynamics of how teams of developers approach OOSD vs. CSD should be explored. Longitudinal studies should be conducted to determine if those who found OOSD difficult to learn eventually mastered the techniques and whether those who found OOSD less difficult to learn were any more successful at applying the methodology.

One problem with conducting future research on OOSD involves clearly defining a strategy to address specific research questions. The following list presents several dimensions that empirical researchers should consider in the design of future experiments or field studies on the pros and cons of OOSD:

- Underlying theory used to develop hypotheses
- Types of methodologies to be compared: process-oriented, data-oriented, object-oriented
- Types of applications to be developed: function-intensive, data-intensive, hybrid
- Complexity of applications to be developed: simple classroom vs. complex industrial-strength
- Level of previous OO development experience: novice vs. experienced
- Type of previous experience: process-oriented, data-oriented, object-oriented
- Type of development approach: individual vs. team
- Type of experiment: laboratory vs. field (including survey research)
- Sample size: small vs. large
- Scope of experiment: single OOSD phase/partial system vs. multiple phases/complete systems
- Time frame of research: cross-sectional vs. longitudinal

As is apparent from the list above, the choices for empirical investigation of OOSD are numerous. An ideal situation would be to collect detailed data on experienced individual developers or development teams who create identical complete real-world systems (perhaps of varying complexity) using both conventional and OO methods. For all phases of such projects, direct comparisons could be made in a head-to-head competition.

Regardless of the particular research question to be addressed, better experimental designs with tighter controls and larger samples could enhance validity. The obvious dilemma in this type of research is obtaining the cooperation of sufficiently large numbers of qualified subjects for laboratory or field studies. However, without adequate experimental designs, a quick resolution to the OO controversy will remain elusive.

## VIII. CONCLUSION

To resolve the question of whether object-oriented systems development (OOSD) is indeed superior to conventional systems development (CSD) requires strong empirical evidence. This paper reviewed eighteen empirical studies on a wide variety of OOSD phases and techniques. These studies are marked by two primary but somewhat conflicting characteristics. First, the majority of studies found OOSD superior to CSD. The major strengths of OOSD are improvements in quality and productivity while the primary weakness of OOSD is its apparent difficulty to learn. Second, the theoretical foundations and research methodologies represented by many of the studies are seriously lacking, suggesting a call for future research to address the OO controversy more concretely.

*Editor's Note*: This article was received on August 21, 2001. It was with the author for three and a half months for one revision. The article was published on January 28, 2002

## REFERENCES

Agarwal, R, Sinha, A.P., and Tanniru, M. (1996). Cognitive fit in requirements modeling: a study of object and process methodologies. *Journal of Management Information Systems*, 13:2 (Fall), 137-162.

Basili, V., Briand, L. and Melo, W. (1996). How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39:10 (October), 104-116.

Boehm-Davis, D. and Ross, L. (1992). Program design methodologies and the software development process. *International Journal of Man-machine Studies*, 36, 1-19.

Booch, G. (1994). *Object-oriented analysis and design with applications*, 2nd ed. Benjamin/Cummings (Redwood City, CA).

Briand, L., Arisholm, E., Counsell, S., Houdek, F., and Thevenod-Fosse, P. (1999). Empirical studies of object-oriented artifacts, methods, and processes: state of the art and future directions. *Empirical Software Engineering: An International Journal*, 4:4 (December), 387-404.

Briand, L., Wust, J., Daly, J., and Porter, D. (2000). Exploring relationships between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51, 245-273.

Brooks, F.P. (1987). No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20:4 (April), 10-19.

Chen, D.J. and Chen, D.T.K. (1994). An experimental study of using reusable software design frameworks to achieve software reuse. *Journal of Object Oriented Programming*, 7:2 (May), 56-67

Coad, P. and Yourdon, E. (1991). *Object-oriented analysis*, 2nd ed. Yourdon Press (Englewood Cliffs, NJ).

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. and Jeremaes, P. (1994). *Object-oriented development: the fusion method*. Prentice-Hall (Englewood Cliffs, NJ).

Davies, S.P., Gilmore, D.J. and Green, T.R.G. (1995). Are objects that important? Effects of expertise and familiarity on classification of object-oriented code. *Human-Computer Interaction*, 10, 227-248.

DeMarco, T. (1978). *Structured analysis and system specification*. Prentice-Hall (Englewood Cliffs, NJ).

Fayad, M., Tsai, W. and Fulghum, M. (1996). Transition to object-oriented software development. *Communications of the ACM*, 39:2 (February), 108-121.

Fedorowicz, J. and Villeneuve, A. (1999). Surveying object technology usage and benefits: A test of conventional wisdom. *Information & Management*, 35, 331-344.

Fichman, R.G. and Kemerer, C.F. (1992). Object-oriented and conventional analysis and design methodologies: comparison and critique. *IEEE Computer*, 25:10 (October), 22-39.

Fichman, R.G. and Kemerer, C.F. (1993). Adoption of software engineering process innovations: the case of object orientation. *Sloan Management Review*, 34:2 (Winter), 7-22.

Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley (Reading, MA).

Garceau, L., Jancura, E., and Kneiss, J. (1993). Object-oriented analysis and design: a new approach to systems development. *Journal of systems management*, 44:1 (January), 25-33.

Hardgrave, B. (1997). Adopting object-oriented technology: evolution or revolution*? The Journal of Systems and Software*, 37, 19-25.

Hardgrave, B. and Dalal, N. (1995). Comparing object-oriented and extended-entity-relationship data models. *Journal of Database Management*, 6:3 (Summer), 15-21.

Harrison, R., Samaraweera, L., Dobie, M., and Lewis, P. (1996). Comparing programming paradigms: an evaluation of functional and object-oriented programs. *Software Engineering Journal*, 11:4 (July), 247-254.

Herbsleb, J., Klein, H., Olson, G., Brunner, H., Olson, J., and Harding, J. (1995). Object-oriented analysis and design in software project teams. *Human-Computer Interaction*, 10, 249-292.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The unified software development process,* Reading, MA: Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. (1995). *Object-oriented software engineering: a use case driven approach*, 2nd ed. Addison-Wesley (Wokingham, England).

Johnson, R.A. (2000). The up's and down's of OOSD: does experience make a difference? *Communications of the ACM*, (forthcoming).

Korson, T. and McGregor, J. (1990). Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, 33:9 (September), 40-60.

Lewis, J., Henry, S., Kafura, D., and Schulman, R. (1992). On the relationship between the object-oriented paradigm and software reuse: an empirical investigation. *Journal of Object-Oriented Programming*, 5:4, 35-42.

McFadden, F. and Hoffer, J. (1991). *Database Management*, 3rd ed. Benjamin/Cummings (Redwood City, CA).

Morris, M., Speier, C., and Hoffer, J. (1999). An examination of procedural and object-oriented systems analysis methods: Does prior experience help or hinder performance? *Decision Sciences Journal*, 30:1 (Winter), 107-136.

Newell, A., and Simon, H.A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.

Pancake, C.M. (1995). The promise and the cost of object technology: a five-year forecast. *Communications of the ACM*, 38:10 (October), 33-49.

Pennington, N., Lee, A.Y., and Rehder, B. (1995). Cognitive *activities and levels of abstraction in procedural and object-oriented design. Human*-Computer Interaction, 10, 171-226.

Rumbaugh, J., Booch, G., and Jacobson, I. (1998). *The unified modeling language reference manual,* Reading, MA: Addison-Wesley.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991*). Object-oriented modeling and design*. Prentice Hall (Englewood Cliffs, NJ).

Shlaer, S. and Mellor, S. (1989). *Object lifecycles: modeling the world in states*. Prentice-Hall (Englewood Cliffs, NJ).

Smith, H.A. and McKeen, J.D. (1996). Object-oriented technology: getting beyond the hype. *The DATA BASE for Advances in Information Systems*, 27:2 (Spring), 20-29.

Vessey, I. (1991). Cognitive fit: a theory-based analysis of the graphs versus tables literature. *Decision Sciences*, 22:2, 219-240.

Vessey, I. and Conger, S. (1994). Requirements specification: Learning object, process, and data methodologies. *Communications of the ACM*, 37:5 (May), 102-113.

Wang, S. (1996a). Toward formalized object-oriented management information system analysis. *Journal of Management Information Systems*, 12:4 (Spring), 117-141.

Wang, S. (1996b). Two MIS analysis methods: an experimental comparison. *Journal of Education for Business*, 71:3 (Jan/Feb), 136-142.

Wirfs-Brock, R. and Johnson, R. (1990). Surveying current research in object-oriented design. *Communications of the ACM*, 33:9 (Sep.), 105-124.

Wood, M., Daly, J., Miller, J., and Roper, M. (1999). Multi-method research: an empirical investigation of object-oriented technology. *The Journal of Systems and Software*, 48:1 (Aug.), 13-26.

## ABOUT THE AUTHOR

**Richard A. Johnson** is Assistant Professor of Computer Information Systems in the College of Business Administration at Southwest Missouri State University. He received his Ph.D. in Business Administration from the University of Arkansas. His Research interests include object-oriented develoment, E-business systems development, and ERP systems. Dr. Johnson's publications appear in *Communniucations of the ACM, DATA BASE*, and the *Journal of Systems and Software, among others*.


ISSN: 1529-3181