South African
Computer
Journal
Number 11
May 1994

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 11
Mei 1994

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarverenging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof John Shochot
University of the Witwatersrand
Private Bag 3
WITS 2050
Email: 035ebrs@witsvma.wits.ac.za

Production Editor
Dr Riël Smit
Mosaic Software (Pty) Ltd
P.O.Box 16650
Vlaeberg 8018
Email: gds@cs.uct.ac.za

Subscriptions

	Annual	Single copy
Southern Africa:	R45,00	R15,00
Elsewhere:	$45,00	$15,00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

The Object-Oriented Paradigm: Uncertainties and Insecurities

Stephen R Schach
Computer Science Department, Vanderbilt University, Nashville, TN 37235, U.S.A.

Abstract

The object-oriented paradigm is widely promoted as the optimal way to develop software. However, there is as yet no experimental evidence to back that assertion, and it is extremely unlikely that such evidence could be obtained. For the present, the best we can do is to rely on theoretical reasons for the superiority of the object-oriented paradigm over the structured paradigm. In addition to this lack of evidence regarding the cost effectiveness of the paradigm as a whole, we also do not have adequate data regarding good management techniques for each of the life-cycle phases of the paradigm. Until such data become available, the best that can be done is to extrapolate from the structured techniques.
Keywords: *Object-oriented paradigm, software engineering, management of software.*
Computing Review Categories: *D.1.5, D.2.2, D.2.9, D.2.m, K.6.3*

Received: June 1993

1 Introduction

The software buzzword for the 1980s was "structured." Software engineers extolled the use of structured systems analysis [2], structured design [21], structured programming [7], and structured testing [12]. The use of these techniques resulted in substantial improvements in software quality and productivity. Nevertheless, toward the end of the 1980s, it became apparent that the so-called structured paradigm was not the complete answer to all our problems. Software projects still ran late and over budget, and delivered software still had residual faults ("bugs"). The claim is nowadays repeatedly made that the way to ensure quality software is to abandon the structured paradigm and instead adopt the object-oriented paradigm. And the term "object-oriented" has become the software buzzword for the 1990s.

From a purely technical viewpoint, it is straightforward to implement the object-oriented paradigm within an organization. After all, there is a plethora of books, courses, videotapes, CASE tools, and consultancy organizations that will provide technical information regarding object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP). However, expertise with regard to the purely technical aspects of software development is by no means enough–in addition, management has to be knowledgeable regarding the managerial implications of the object-oriented paradigm.

This paper has two themes. The first is that it is hard to prove that the object-oriented paradigm is an improvement over the structured paradigm, although there are strong theoretical reasons for believing this to be so. The second theme is that considerably more research is needed before managers will be able to manage the object-oriented techniques with the same facility as the structured techniques.

The remainder of the paper is organized as follows. Difficulties in proving that the object-oriented paradigm is cost effective are discussed in Section 2, while theoretical reasons why this should be so are presented in Section 3. The second theme, namely difficulties in managing the object-oriented paradigm, is presented in Section 4. Conclusions are given in Section 5.

2 Cost Effectiveness of the Object-Oriented Paradigm

Before jumping on the object-oriented bandwagon, the managers of a software organization have to know whether the object-oriented paradigm is indeed as effective as its proponents claim, or whether it is just another passing fad. The key question that has to be answered is: Is the object-oriented paradigm cost effective? That is to say, management has to know whether it is cheaper over the long haul to invest in object-oriented technology or to continue using the structured paradigm.

In theory, it is easy to compare the two paradigms. All that has to be done is to construct two versions of a non-trivial product, one team using the structured paradigm, the other team using the object-oriented paradigm. Then, the resulting products are compared from a variety of viewpoints such as cost, duration, quality, and so on. There are just two things wrong with this experiment–it would cost too much to perform, and even if it were performed as described, the results would be meaningless.

The first problem is that of cost. Suppose that the internal cost of the product as a whole is $200,000. Perhaps the client can be charged as much as $300,000. But now the development has to be done in duplicate. The cost to the software developers is then $400,000. No software development organization that is run for profit can afford to perform such an experiment. But the situation is worse than that. The proposed experiment uses only two teams. No sensible conclusions could be drawn from an experiment based on such a small sample. To get around this problem, the experiment must incorporate many more teams. At

least 10 teams should use each paradigm, and the resulting 20 products should then be compared. This would incur a total cost of $4,000,000. Clearly the prohibitive cost makes the experiment impractical.

In fact, if the experiment is to be meaningful, the cost would have to be far greater than $4,000,000. The cost of a product is not just the cost of development but also the cost of maintenance which, on average, is twice the cost of development [17]. Thus, when comparing two versions of the product, they should be separately maintained until the maintenance implications of the two paradigms can be compared. The cost implications of building two versions of the same product have been pointed out. How much more impractical is the idea of maintaining, for the next 10 or 15 years, two parallel versions, let alone the twenty (or preferably more) parallel versions needed for statistical proof. The issue of cost means that continuing an impractical experiment through the maintenance phase makes the experiment even more impractical. Nevertheless, a critical aspect when comparing the two paradigms is how the resulting products behave during the maintenance phase.

Further complications are caused by the observed huge differences between the abilities of software professionals. Sackman performed a series of experiments and measured differences of up to 28 to 1 between pairs of programmers with respect to items such as coding and debugging time, and product size [16]. Superficially, this is easy to explain: An experienced programmer will almost always outperform an entry-level programmer. But that is *not* what Sackman and his colleagues measured. They worked with matched pairs of computer professionals. What is most alarming about their results is that the biggest observed differences, such as the figure of 28 to 1 quoted previously, were between pairs of *experienced* programmers. These huge observed differences mean that considerably more than 10 pairs of teams using each paradigm would be needed to ensure that observed results cannot be ascribed to differences between individual software professionals. Thus the cost of performing a credible experiment would be astronomical.

One way of keeping experimentation costs down is to use students as subjects. However, the validity of such experiments is dubious. There are a number of differences between college students doing a class project and software professionals developing a piece of software. First, their motivations are different. Students are usually motivated by the grade they hope to achieve for the course, whereas the motivation for software professionals is job satisfaction and/or money. A second difference is that of scale. The largest practical classroom-based team project can be at most 2 person-months of effort, hardly a product that can be considered to be programming-in-the-many [17]. Even with a graduate class, 3 person-months is probably the upper limit. Third, an experiment performed as part of a class can last only as long as that class, as opposed to the year or more needed to develop a nontrivial product, let alone the 10 or 15 years over which a product is maintained. For all these reasons it simply is not valid to use students as subjects in experiments to compare the two paradigms. Thus, the results of experiments performed on students to measure the effectiveness of the object-oriented paradigm, such as [5] and [9], should be treated as indications, not experimental proof.

In summary, there is no way that experiments can be performed to prove that the object-oriented paradigm is more cost effective than the structured paradigm. The best we can do is to come up with theoretical reasons. This is discussed in the next section.

3 Theoretical Reasons for Switching to the Object-Oriented Paradigm

An incomplete definition of the object-oriented paradigm is that the product is designed in terms of abstract data types, and the variables ("objects") are instantiations of abstract data types. But defining an object as an instantiation of an abstract data type is too simplistic. Something more is needed, namely inheritance. The basic idea behind inheritance is that new data types can be defined as extensions of previously defined types, rather than having to be defined from scratch [13].

The advantages of using objects (or rather, classes) are precisely those of using abstract data types, including data abstraction and procedural abstraction [17]. In addition, however, the inheritance aspect of classes provides a further level of abstraction, leading to easier and less fault-prone product development. With regard to maintenance, classes, like abstract data types, appear to be features of products that will not change from version to version. Thus it is hoped that products designed in terms of classes (and hence objects) will be easier to maintain than many present-day products that were developed before the object-oriented paradigm became widely accepted by software engineers. Reinforcing this is the fact that object-oriented analysis (OOA) is a modeling technique [15]. The output from OOA consists of objects and classes which are abstractions of key aspects of the portion of the real world being modeled. Thus, even though the functionality of the product may be modified during maintenance, the underlying objects and classes are unlikely to change.

Another way of viewing the maintenance advantages of the object-oriented paradigm is to compare it to the structured paradigm. The methods of the structured paradigm fall into two classes, namely process-oriented and data-oriented. The primary focus of a process-oriented method (such as structured design [21]) is the processes of the product; the data on which those processes are performed are of secondary importance. Conversely, a data-oriented method (such as JSD [6]) concentrates on the data, and the processes are considered only later. In reality, there is a duality between process and data, and in a certain sense they can be considered equivalent [4]. This duality of process and data is implicitly recognized in the object-oriented paradigm because, as previously stated, a class is an abstract data type. That is to say, it encapsulates both a data structure and all the actions (processes) that are performed on that data structure. Consequently, if changes are

made either to the data structure or the actions performed on it, such changes are localized to the class, its instantiations (objects), and any subclasses. This in turn reduces the chance of a regression fault. Only if the public part of a class or object is modified could this directly affect the rest of the product.

A second reason for promoting objects is that reuse is enhanced. Software reuse is virtually as old as computing itself. The machines of the 1940s were essentially bare hardware; each programmer had to write his or her own routines for such common operations as input/output or floating-point arithmetic. Soon, however, individual programmers started to share these routines among themselves. These reused routines are today part of the operating system and run-time support environment. However, not only modules can be reused in new products, but also specifications, designs, plans, test cases, and documentation of all kinds, including manuals. Nevertheless, even though reuse goes far beyond merely reusing code modules, in practice software reuse has been largely restricted to reusing modules.

The fact that time and money could be saved by reusing components written by others has long been recognized. For example, in 1974 the theory of composite/structured design was proposed [19]. The major reason for designing modules with the highest possible cohesion was stated to be that this would support future reuse of those modules. Specifically, "functional cohesion" (which is achieved when a module performs one action and one action only) was put forward as the mechanism for achieving reuse. While it is true that a module with functional cohesion performs just one action, the problem is that it performs that one action on data. Unless the new product contains the same (or highly similar) data as the original product, then reuse is hard to achieve.

On the other hand, the object-oriented paradigm promotes reuse because a class, being an abstract data type, encapsulates both data and the actions performed on that data. Thus the problems that arise when reusing a module with functional cohesion do not arise when a class is reused. But the inheritance properties of a class lead to a level of reuse beyond that which can be achieved by an abstract data type. If a new product requires a class N, say, and an existing class C is very similar to N, then instead of constructing N from scratch, N is instantiated as a subclass of C. That is to say, N inherits all the properties of C except those properties that are explicitly modified. This mechanism does not change C in any way, nor does it change any existing subclasses of C. Furthermore, the fact that N is a (sub)class means that it encapsulates an abstract data type. Consequently, modifying a subclass to fit a specific need in a new product is both rapid and easy. Instantiation, possibly followed by modification, is thus the primary mechanism through which the object-oriented paradigm can reduce development time.

The implementation of a class embodies information hiding [14]. This term is a misnomer; "details hiding" would be more accurate. What the term means is that details as to how the data structure and actions of a class are implemented are not visible outside the class. Thus, if the way that the data structure is implemented is changed during maintenance, this cannot impact the rest of the product. From the viewpoint of reuse, information hiding forces a class to be an abstraction with a well defined interface. This makes reuse far easier. In addition, the fact that information hiding makes changes during maintenance easy and safe also makes tailoring of subclasses during development easy and safe. Another way of viewing this is that information hiding results in a class being implemented as a black box, with a resulting positive impact on reuse, and hence on both development and maintenance.

Reuse of classes has a positive effect on maintenance for a number of reasons. First, when a maintenance programmer looks for the first time at a product that includes reused classes, he or she may already be familiar with part of it. Second, a component that is to be reused will, in general, have been thoroughly tested, thus reducing the chances of a residual fault ("bug"). In addition, it will probably also be comprehensively documented, thus making maintenance easier and safer. Third, a reused class may already have been reused in a number of other products. The more widely a class has been reused, the smaller the chance that there are still residual faults. Fourth, a difficulty with maintenance is that making a change to one part of a product might induce a fault in some other, apparently unrelated, part of the product, a so-called regression fault. In general, reused components must be independent and self-contained. In the case of reused classes, information hiding further enhances this independence, and hence reduces the chance of a regression fault.

There are thus strong theoretical grounds for suggesting that the object-oriented paradigm increases software productivity in two ways. The first is by making maintenance quicker, easier, and safer, and the second is by promoting reuse and thereby reducing the cost of both development and maintenance. The question is: to what extent are these theoretical arguments supported by hard experimental data?

Some experiments have been performed that support the claim that the use of objects indeed simplifies maintenance and promotes reuse; see, for example, references [5] and [9]. Unfortunately, these experiments were performed on students and, as discussed in Section 2, the results of such experiments cannot be extrapolated to software professionals.

In addition to experiments on students, there are anecdotal reports regarding the effectiveness of the object-oriented paradigm. That is to say, individuals have informally reported that they have used object-oriented techniques for software development, and that software costs appear to have decreased. More precisely, what is reported is that the first time an organization develops software in terms of objects, the cost of development using objects is higher and the size of the product is larger than when traditional methods are used. This is particularly noticeable when software with a graphical user interface is developed. However, this initial cost is recouped in two different ways. First, maintenance costs are lowered, thus reducing

Table 1. The phases of a typical software process model.

1. Requirements phase
2. Analysis (specification) phase
3. Planning phase
4. Design phase
5. Implementation phase
6. Integration phase
7. Maintenance phase
8. Retirement

the overall cost over the lifetime of the product. Second, the next time that a new product is developed, some of the objects from the previous project can be reused, further reducing software costs.

It is important to stress that these reports are anecdotal; there are no published studies with statistical data that support these informal reports. This is a consequence of a disturbing trend in the software industry. During the formative period of software engineering (approximately 1975 to 1985) not overmuch hard data were produced, but the available information was generally freely shared. This allowed researchers to determine what the real problems were, and find appropriate solutions. However, competitive pressures have prevented companies from publishing current statistics. It is understandable that, when a company has invested a huge sum of money to evaluate the object-oriented paradigm, purchase the required hardware and software, train software professionals, and conduct pilot projects, it does not wish its competitors to obtain the benefits of its investment at little or no cost. As a consequence, however, the only data that are available are generally either company confidential, or anecdotal in nature.

Furthermore, the newness of the technology means that some of the informal data relate to pilot studies. Such pilot studies are frequently on a smaller scale, and are performed by highly talented employees who are already committed to the object-oriented paradigm. In addition, the Hawthorne effect [11] can also be a factor in evaluating the reliability of pilot studies. For all these reasons, the reliability of the results of such informally reported pilot studies can be seriously questioned. The bottom line is that we simply do not have the data to determine whether the object-oriented paradigm works.

4 Management of the Object-Oriented Paradigm

The eight phases of a typical software process model are depicted in Table 1 [17]. Whereas every component of a product usually undergoes each of the phases in order, the product as a whole may follow a considerably more complex software process model, for example, if components are built in parallel, or if iterative techniques are used. There is no separate testing phase or documentation phase; testing and documentation are an intrinsic part of every individual phase.

First we turn our attention to the planning phase. While it is true that planning takes place throughout a project, the specification document has to be approved by the client before the development team can estimate project duration and project cost, and then draw up the software project management plan. That is to say, the major planning activities for a project cannot take place until the specifications are complete. On the other hand, it is foolhardy to commence the design phase until the detailed plan has been drawn up. This is why the planning phase follows the specification phase and precedes the design phase.

Two of the key work products of the planning phase are the estimates of the duration and cost of the project. These are now considered in more detail.

Duration and Cost Estimates

There are two main types of duration and cost estimation methods that can be used in conjunction with the structured paradigm. First, there are metrics based on the functionality of the target project, such as function points [20]. Second, algorithmic models such as COCOMO [1] are employed. However, neither of these two methods nor any other can take all aspects of reuse into account. There are two key problems. The first is that specific reuse decisions are made during the design phase, not the specification (analysis) phase. However, the planning phase (and more specifically, duration and cost estimation) takes place after specification and before design. Presenting the client with duration and cost estimates only after the design is complete can carry a high degree of risk. If the client rejects the proposal at that stage, the development organization will be considerably out of pocket. A conservative way to bid on projects would be to assume that there will be no reuse, and to set the price of the product accordingly. However, reuse rates of 40% have been obtained for years [8, 10, 17]. It is true that ignoring reuse cannot lead to underbidding for a project, but there is a real risk of losing contracts to other organizations that assume that they will be able to achieve a specific reuse rate, and then set their bids accordingly. On the other hand, assuming that a specific reuse rate can be achieved is also risky; the resulting bid may be set far too low.

There is a second complication that arises with software reuse. When an item is reused, it may be reused unchanged, or with minor changes, or with major changes. However, it is difficult to estimate in advance the extent of the changes that will have to be made. For example, in a NASA study of reuse of code modules [18], the following reuse rates were obtained: 28% of modules were reused unchanged, 10% were reused with minor changes, and 7% with major changes. In that study, a reused module was considered to have undergone major changes if 25% or more of the module was altered. It is difficult and time-consuming to make major changes to a code module. Accordingly, if the NASA developers had known in advance that about one-sixth of the reused modules would require major changes, it is likely that they would have preferred to have constructed those modules from scratch. Of course, software developers do not have the gift of hindsight. The fact that one-sixth of the reused modules in the NASA study underwent major changes implies that,

in those cases, the developers simply made a mistake in trying to reuse an existing module. This is not meant to be a criticism of the developers. Rather, it is an indication of how difficult it currently is to estimate in advance just how much of an existing module will have to be changed to tailor it for use in a new product. And we know even less about reuse of classes than we do about code modules.

There is a fundamental difference between the two ways in which reuse impacts duration and cost estimation. The fact that specific reuse decisions cannot be made earlier than during the design phase means that there will always be a risk when estimates are made at the start of the planning phase. With experience, the magnitude of the risk can possibly be reduced, but the underlying risk itself is intrinsic to the process of estimating a reuse rate; there is no way to resolve this risk. On the other hand, it may be possible in the future to develop accurate metrics which will provide a measure of how easy it would be to transform an existing software component for use in a new software product. Thus, it is possible that, in the future, the chances of having to make unanticipated major changes to a reused component will decrease. But until such a metric is developed, there is a nontrivial risk that reuse of an existing component may in fact increase the cost of the project.

Even without reuse, there are difficulties when metrics for the structured paradigm are applied to the object-oriented paradigm. For example, consider COCOMO [1]. Underlying this model is the assumption that the effort E to build a product of size S is given by the equation

$$E = aS^b$$

where a and b are constants that are obtained from previous project data. There are two problems that arise if COCOMO is applied to a project that uses the object-oriented paradigm. First, there is no reason to believe that the values of constants a and b are unchanged when the COCOMO equations are applied to the object-oriented paradigm, but no data one way or the other has yet been published. Second, there is a wide variety in the ways that object-oriented software is developed. Some organizations are essentially using the same hardware and software tools that they used for the structured paradigm, while others are developing object-oriented software on graphical workstations using state-of-the-art CASE (computer-aided software engineering) tools. COCOMO includes "software cost multipliers" to handle such differences [1], but these multipliers have not been updated to incorporate integrated CASE, GUIs (graphical user interfaces), and workstations. In short, structure-oriented estimation techniques are all that we have to enable us to come up with duration and cost estimates, but there is no reason to believe that they will be accurate when applied to object-oriented projects.

The Development-to-Maintenance Ratio

A major factor influencing software management is the percentage of the software budget that is devoted to maintenance. Specifically, studies on mainframes have shown that, on average, 67% of the total software budget is spent on maintenance [3, 16]. Bearing in mind that some organizations do nothing but development, the percentage of money devoted to maintenance in a given organization can exceed 70% or even 80%. The significance of the development-to-maintenance ratio is as follows: If the ratio is indeed 33-to-67, then for every $1 spent on development, $2 is spent on maintenance. This in turn colors every aspect of the software development process. For example, suppose management decides to build a rapid prototype first. Once satisfied that the rapid prototype captures the key functionality of the target product, the next step is to draw up the specifications. In many instances, a possible alternative to the specification document is to use the rapid prototype as the basis for the specifications, and simply to draw up a list of additional features that the target product must support. The advantage of this scheme is that it is quick, and a specification in the form of a rapid prototype cannot contain the ambiguities, omissions, and contradictions that so frequently arise in a written specification document. But that advantage is far outweighed by the major disadvantage of this approach, which is that the specification document is necessary for ease and safety in maintaining the product. In other words, the specification document is needed for maintenance far more than for development. Taking this argument further, it might be argued that if it were known for a fact that a specific product would never be maintained, then it would be unnecessary to produce not just the specification document, but all documentation of any kind for that product. This is, of course, an extreme position; for example, it is difficult to conduct either execution-based or nonexecution-based testing in the absence of documentation. But the underlying idea, namely that the primary need for documentation is to support maintenance, is valid.

Another example of the impact of the development-to-maintenance ratio also concerns rapid prototyping. A common recommendation is to discard a rapid prototype after use, and not to refine it into the final product. The reason is that the essence of a rapid prototype is that it is quickly built without specifications or design, and is therefore essentially unmaintainable. In the long run, then, it is better to discard the rapid prototype and carefully build the target product from scratch than to attempt to refine the rapid prototype, warts and all, into a production-quality software product. However, if maintenance becomes less important, then the maintainability of the rapid prototype also decreases in importance, and refinement of the rapid prototype may become an economically viable development technique.

Suppose that the development-to-maintenance ratio were to increase to (say) 75-to-25. The primary thrust of the software industry would then be development, and software processes would be oriented towards the rapid delivery of software products; the subsequent maintenance would be less important. Thus, the development-to-maintenance ratio has a major impact on the software process, and hence on the management of software.

The relevance of these remarks to management of the object-oriented paradigm is as follows. If the claims re-

garding the object-oriented paradigm are true, then the percentage of resources devoted to maintenance will decrease. That is to say, the use of the object-oriented paradigm will lead to an increase in the development-to-maintenance ratio. (The use of appropriate CASE tools could lead to an additional increase.) Management must be aware that this change in the ratio will take place, and must be in a position to take advantage of it. Bearing in mind that many software professionals disdain maintenance, the change in the development-to-maintenance ratio could be used to shift disgruntled maintenance programmers into development positions.

Time Devoted to Each Phase

Many software engineering textbooks include a pie chart depicting the percentage of time spent on each phase of the process. A typical pie chart of this kind is shown in Figure 1. In that figure, 6% is devoted to the specification

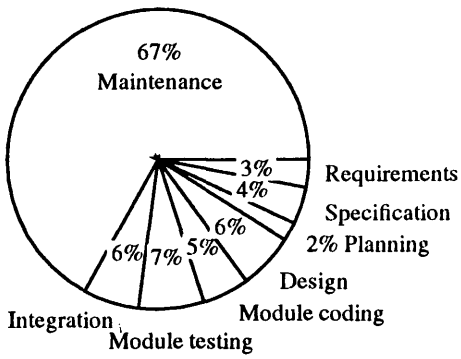

Figure 1. Approximate relative costs of the phases of the software process model. (from [17,p11], used with permission)

(analysis) phase, 7% to the design phase, and so on. While the exact figures will vary from book to book, depending on the specific data utilized by the author, they all essentially reflect the process of software development on a mainframe without the use of CASE tools of any kind. Now suppose that a code generator is used to generate code directly from the detailed design. In this case, zero time is needed for the module coding phase. While use of the object-oriented paradigm is unlikely to have as drastic an impact as that, there is no reason to believe that the percentages in the diagram will stay fixed; reuse, in particular, is likely to have a significant impact. At this stage it is difficult to forecast exactly what the impact of the new paradigm will be. Nevertheless, management has to be aware that staffing changes may be inevitable in the future.

One such change is likely to arise in the area of analysis. A considerable amount of modeling is done during the object-oriented analysis (OOA) phase. In addition, many steps are carried out in OOA that correspond to steps of the design phase of the structured paradigm. It is therefore likely that, relative to the structured paradigm, the object-oriented paradigm requires considerably more time during the specification (analysis) phase, and considerably less during the design phase. If this is indeed the case, it will be necessary to acquire additional analysts and dismiss some designers, or perhaps to retrain some designers as analysts.

As previously stated, it is virtually certain that, when the object-oriented paradigm is used, the percentage of time devoted to each phase of the software process model will change from what appears in Figure 1. As soon as experimental data become available, new software process models can be developed that stress those phases that turn out to be the most time-consuming, in much the same way that current process models stress the maintenance phase.

5 Conclusions

We currently simply do not know enough about the object-oriented paradigm to be able to make important management decisions. In particular, we do not yet know whether the object-oriented paradigm is more cost-effective than the structured paradigm. There are strong theoretical indications that the object-oriented paradigm is superior, and this is backed by informal, anecdotal reports, but there is no statistical proof yet available that this is the case.

With regard to management of software development, we have had sufficient experience with the structured paradigm to be able to draw up a software project management plan, incorporating not just duration and cost estimates, but also a detailed schedule incorporating the effort that will be required for each phase or task. Whereas such a plan is almost never totally accurate, an experienced management team is usually able to come up with reasonable estimates that are generally not too far off. Furthermore, metrics are available for monitoring the development process. If it should happen that a major deviation from the planned schedule is inevitable, then management is generally sufficiently familiar with the structured paradigm to be able to take appropriate action. In particular, it is usually possible to determine what went wrong, and why. An amended schedule can then be drawn up, incorporating this information.

In contrast, there is a dearth of management information regarding the object-oriented paradigm. Nevertheless, software is currently being developed using this paradigm. Until such time as data are available that could assist management in this regard, the best that can be done is to use the management techniques of the structured paradigm, while constantly being aware of the absence of appropriate management tools.

References

1. B W Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.
2. T DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New York, 1978.
3. C Ghezzi, M Jazayeri, and D Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, Englewood Cliffs, NJ, 1991.
4. R Goldberg. 'Software engineering: An emerging discipline'. *IBM Systems Journal,* **25**:334–353, (1986).

5. S M Henry and M Humphrey. 'A controlled experiment to evaluate maintainability of object-oriented software'. In *Proceedings of the IEEE Conference on Software Maintenance*, pp. 258–265, San Diego, CA, (1990). IEEE Computer Society Press.
6. M A Jackson. *System Development*. Prentice-Hall, Englewood Cliffs, NJ, 1983.
7. D E Knuth. 'Structured programming with goto statements'. *ACM Computing Surveys*, **6**:261–301, (1974).
8. R G Lanergan and C A Grasso. 'Software engineering with reusable designs and code'. *IEEE Transactions on Software Engineering*, **SE-10**:498–501, (1984).
9. J A Lewis, S M Henry, D G Kafura, and R S Shulman. 'An empirical study of the object-oriented paradigm and software reuse'. *ACM SIGPLAN Notices*, **26**:184–196, (1981). Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '91.
10. Y Matsumoto. 'A software factory: An overall approach to software production'. In P Freeman, ed., *Tutorial: Software Reusability*, pp. 155–178. IEEE Computer Society Press, Washington, DC, (1987).
11. E Mayo. *The Human Problems of an Industrial Civilization*. Macmillan, New York, 1933.
12. T J McCabe. *Structural Testing*. IEEE Computer Society Press, Los Angeles, 1983.
13. B Meyer. 'Genericity versus inheritance'. *ACM SIGPLAN Notices*, **21**:391–405, (1986). Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.
14. D L Parnas. 'On the criteria to be used in decomposing systems into modules'. *Communications of the ACM*, **15**:1053–1058, (1972).
15. J Rumbaugh *et al. Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
16. H Sackman, W J Erikson, and E E Grant. 'Exploratory experimental studies comparing online and offline programming performance'. *Communications of the ACM*, **11**:3–11, (1968).
17. S R Schach. *Software Engineering*. Richard D Irwin/Aksen Associates, Homewood, IL, second edition, 1993.
18. R W Selby. 'Quantitative studies of software reuse'. In T J Biggerstaff and A J Perlis, eds., *Software Reusability. Volume II: Applications and Experience*, pp. 213–233. ACM Press, New York, (1989).
19. W P Stevens, G J Myers, and L L Constantine. 'Structured design'. *IBM Systems Journal*, **13**:115–139, (1974).
20. C R Symons. *Software Sizing and Estimating: Mk II FPA*. John Wiley and Sons, Chichester, UK, 1991.
21. E Yourdon and L L Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1–3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Further instructions on how to reduce page charges can be obtained from the production editor.
3. In camera-ready format – a detailed page specification is available from the production editor;
4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R30-00 per final page for LaTeX or camera-ready contributions and the maximum is R120-00 per page for contributions in typed format (charges include VAT).

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 2 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972). North-Holland.
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, **9**:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African Computer Journal

Number 11, May 1994
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 11, Mei 1994
ISSN 1015-7999

Contents