Kent Academic Repository
Full text document (pdf)

Citation for published version

DOI

Link to record in KAR

Document Version

UNSPECIFIED

Is object orientation an imperfect paradigm for reliable coding? Worse, does it focus on the wrong part of the life cycle? The author thinks so and explains why.

Does OO Sync with How We Think?

Les Hatton, Oakwood Computing

O f the many revolutions that have captured the hearts of programmers, object orientation has been arguably the most forceful and alluring. Central to the paradigm is that it matches how we think about the world and therefore self-evidently points the way for all future development.

Given that corrective-maintenance costs already dominate the software life cycle and look set to increase significantly, I argue that reliability in the form of reducing such costs is the most important software improvement goal. Yet, as I will show, the results are not promising when we review recent corrective-maintenance data for big systems in general and for OO systems, in this case written in C++. I assert that any paradigm that is capable of decomposing a system into large numbers of small components—as frequently occurs in both OO and conventional systems—is fundamentally wrong. Thus, because both paradigms suffer from this flaw, we should expect no particular benefits to accrue from an OO system over a non-OO system. Further, a detailed comparison of OO programming and the human thought processes involved in short- and long-term memory suggests that OO aligns with human thinking limitations indifferently at best. In the case studies I describe, OO is no more than a different paradigm, and emphatically not a better one, although it is not possible to apportion blame between the OO paradigm itself and its C++ implementation.

SILVER BULLET OR SILVER DUD?

Historically, engineering has progressed in one of two ways:

♦ through slow, incremental improvements derived from consistent measurement, or

♦ as the result of a quantum leap.

In most branches of engineering, we usually find a mixture of the two, with occasional quantum leaps interspersed between long periods of consolidation that are based on careful measurement. Examples of quantum leaps in civil engineering include the arch keystone and the Roman invention of cement.

Why quantum leaps fall short

Software engineering ought to be the same, but regrettably, our lack of emphasis on measurement-based incremental improvement leads us to expect far more from an occasional invention-powered quantum leap. We seek a silver bullet: a single product or technique that will, by itself, provide a far-reaching panacea that transforms our field overnight. Our many failures in the search for this silver bullet[1,2] do not mean that there are none, but conventional engineering teaches us that they are few and far between. Worse, without measurement, we may never find or exploit our silver bullets because we will not recognize them for what they are.

OO represents one of the latest and certainly most expensive attempts to find a silver bullet. It so dominates recent software engineering research and teaching that if it is found to be even partially unworkable, we will again have thrown away a vast amount of money on a whim. Or, even worse, we will be forced to continue using it despite its shortcomings.

Silver-bullet specs

Before we can establish if OO is indeed a valid silver bullet, we must ask what specifically we would want a silver bullet to accomplish in software engineering terms. Over the last few years, some organizations and groups have assiduously amassed enough data on software systems to uncover powerful messages we should no longer ignore. The central questions we currently ask about software concern several factors.

♦ *Reliability.* Many sources indicate that around 80 percent of all money spent on software goes to postrelease maintenance. Further, major studies such as those conducted by Bob Arnold[3] strongly suggest that corrective maintenance is the most expensive
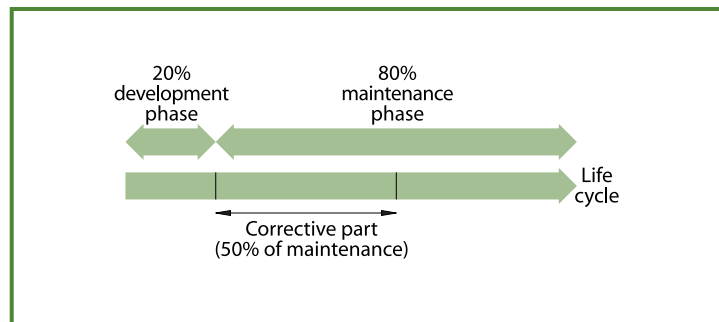

Figure 1. A simple picture indicating the relative costs of components of the life cycle. These figures leave no doubt that corrective maintenance is of strategic importance.

component, amounting to some 50 percent of all maintenance. The remaining 50 percent is ascribed to the other two recognizable maintenance components: adaptive (improvements based on changed behavior) and perfective (improvements based on unchanged behavior). Thus for every hour we spend developing a system, we spend two hours correcting it during its life cycle. As Figure 1 shows, corrective maintenance appears to be the most significant component of the life cycle at around 40 percent. Clearly, then, a valid silver bullet must have a huge beneficial effect on corrective-maintenance costs.

♦ *Productivity.* Programmers cost a lot of money, so any silver bullet should increase productivity, but what does this mean? Productivity, or the rate at which a task is completed, directly depends on the ability to carry out essential development or maintenance tasks—successfully, with long-term goals of usefulness in mind. The very rapid production of inferior code could not conceivably be called productive given the corrective-maintenance costs such code must incur.

♦ *Ease of modification.* This is simply a measure of how efficiently maintenance can be carried out.

♦ *Reuse.* This involves the notion of reusing parts of a design or code in a system for which they were not originally built. Suitability for reuse is primarily aimed at the development phase. A feature of software development since the quantum leap of separate compilation, a notion first understood by Alan Turing in the '40s and '50s, reuse enables the emergence of subcomponent libraries for specific tasks such as numerical algorithms in Fortran. Judging by detailed surveys such as that done by the ACM,[4] apart from GUI building, reuse is not an outstanding success with C++—this despite reuse being the *second* most common reason developers switch to that

Table 1
Defect rates of various products

Product	Language	Faults/KLOC
C parser	C	2.4
C++ parser	C++	2.9

language. Regrettably, the most common reason for the switch is so the developer can add C++ to his or her curriculum vitae. In contrast, I recall achieving a greater than 90 percent level of reuse 20 years ago using scientific library routines called from Fortran.

Thus a valid silver bullet for software must lead to a massive reduction in maintenance, which is by far the life cycle's biggest component. OO should focus on improving maintenance's biggest component, corrective maintenance, rather than obsessing unnaturally on the relatively minor development phase.

OO CASE STUDY

The following recent OO case history details a development project in which project staff tracked in detail corrective-maintenance costs and other maintenance issues. I'll then compare those results with other recently published data.

The case study took place during the 1995 development of static deep-flow analysis toolsets. These operate by reading the source code of C, C++, or Fortran like a compiler, but instead of going on to convert this to machine-executable code, they compare it against knowledge bases of potential problems to identify significant inconsistencies or dangerous use of the language, which is likely to lead to runtime failure. The tools are widely used, which makes corrective-maintenance cost an important consideration.

In this company, from April 1991 onward all software development has been orchestrated by a strict change, configuration, and quality control system. The system yields a complete audit trail, including defect analysis. It uses an in-house-developed product that automates significant parts of the Software Engineering Institute's Capability Maturity Model levels 2 and 3, and provides data for some parts of levels 4 and 5. The company develops products in a Unix environment that runs on Sun and HP workstations and uses various C and C++ compilers.

During the last six years, the company developed a deep-flow analysis tool for (and written in) C, and

then a similar tool for (and written in) C++. The C++ tool closely adheres to the emerging ISO C++ standard. At the time of the project these parsers had similar line counts when measured as nonblank noncomment source lines: 44,700 lines of C for the C parser and 52,500 lines of C++ for the C++ parser. However, the latter has grown significantly since the study to incorporate the full language and is now much bigger than the C parser.

One staff, two paradigms

The single most important feature of this case study, for the purposes of my argument, is that both products were written by the same highly skilled staff, yet the two are radically different. The C++ parser is a true *ab initio* OO-designed parser, whereas the C parser is a traditional design originally built some years earlier using a one-token lookahead yacc grammar along with support routines. The code generated by yacc only amounts to a small fraction of the whole parser. The C++ parser, by further contrast, is a recursive-descent parser owing to the relative intractability of parsing C++ using one-token lookahead grammars.

Another important factor in common is that safe, well-defined subsets of C and C++ were used following published guidelines[5] (although the subset for C was rather stricter than that for C++ during the development). The toolsets themselves enforced these subsets automatically, effectively constituting programming standards. The company required that new code conform absolutely and that existing code improve incrementally until it met the absolute standard. This also was enforced automatically.

Study metrics

The study involved analysis of the company's entire change, configuration, and quality control history between April 1991 and August 1995. For software experimentalists, this system is automated and has the following desirable properties.

♦ Every request for any kind of change received internally or externally—corrective, adaptive, or perfective—is entered into the system as a change request, at which point the system assigns it a unique number, available to the originator for reference. This number can be used to track the CR's progress at any time. The first operational CR formally entered was CR53, and the most recent at study completion was CR2935. Software release notes specifically reference the CRs resolved in that release.

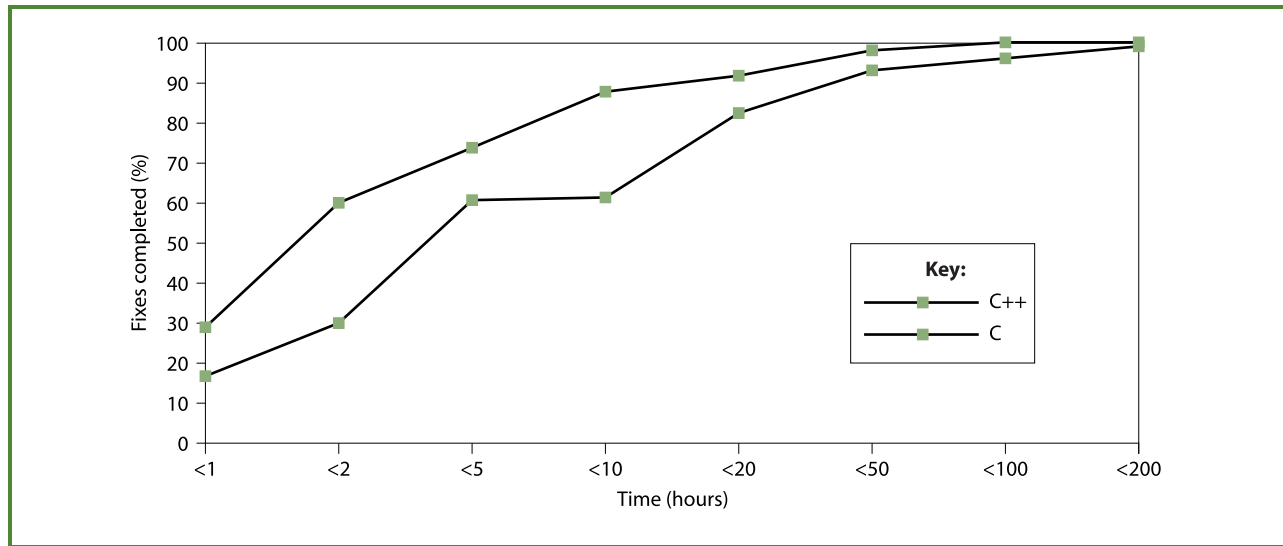♦ Corrective change accounts for only 12 per-

Figure 2. A comparison of the distributions of the times to fix faults in comparable C++ and C development projects.

cent of all CRs (349 out of 2833). The industry average is thought to be near 50 percent.[3] This low value is most likely due to a mixture of the enforced use of safer language subsets, comprehensive regression test suites, and objective test coverage targets.

♦ CRs automatically go through several states:

    evaluation

        → implementation

            → testing

                → loading

                    → dissemination

                        → closed

♦ CRs can be either accepted or rejected at the Evaluation stage. If accepted, they are specified at this point before being advanced to Implementation. If rejected, they automatically mature to the Closed state. A CR belongs to exactly one person at any one time, although this person may change as the CR matures through the above states. CRs can only be assigned by a project coordinator.

Table 1 shows the defect densities of the various products as of August 1995. All defects, however minor, were logged and included in these statistics.

The industry standard for good commercial software is around six defects per KLOC in an overall range of around 6–30 defects per KLOC.[6] In addition, very few systems have ever stayed below one defect per KLOC this far into the life cycle, even in safety-critical systems. Given that the two products shown in Table 1 are used extensively at several hundred sites around the world, and that parsing technology is considered a difficult application area, these sta-

tistics compare well with the population at large. Finally, the average age of these two products since first release is between three and four years, so the fault densities represent those for mature software.

Overall, then, this development environment is well-controlled, formalized, and produces lower defect densities than average products.

RESULTS

I used change requests in two ways: to compare defect densities in the C parser and the C++ parser, and to compare the distributions of the times required to correct defects in the two products.

Defect densities comparison

The two products' defect densities are similar, although that for the OO C++ development is already 25 percent higher and climbing more rapidly than the conventional C development.

Distribution of correction times comparison

This was most revealing and at the same time the most disturbing. One argument in favor of OO systems is that they are easier to change, whether for corrective reasons or otherwise. To measure this, the distribution of times to fix defects was compared between the OO C++ development and the conventional C development. Times were computed automatically from the beginning of the Implementation phase to the beginning of the Loading phase. The developers found
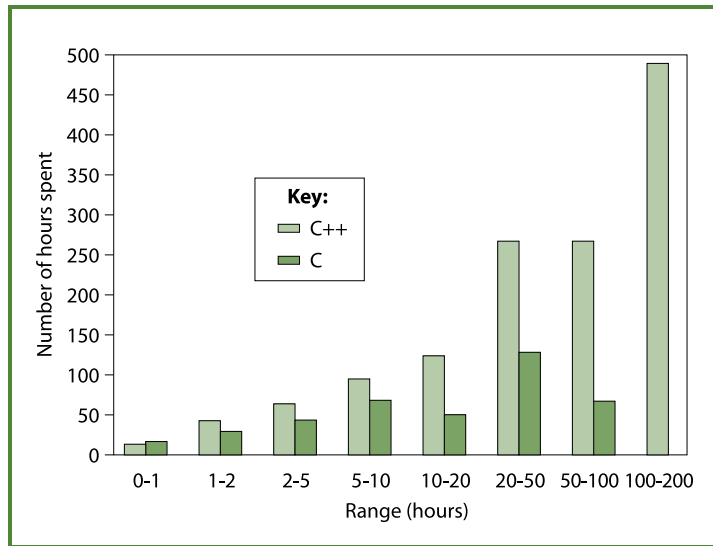
Figure 3. The total number of hours spent in corrective maintenance for each time range for each product. For example, for defects taking between 20 and 50 hours to correct, about 270 hours were spent on the C++ product and about 140 on the C product.

it much harder to trace faults in the OO C++ design than in the conventional C design. Although this may simply be a feature of C++, it appears to be more generally observed in the testing of OO systems, largely due to the distorted and frequently nonlocal relationship between cause and effect: the manifestation of a failure may be a "long way away" from the fault that led to it. To quantify this, Figure 2 shows the percentage of all fixes completed within certain times for both the C and C++ development. To read the graph, note that 60 percent of all fixes for the C development took less than two hours, compared with only 30 percent of all fixes for the C++ development. The C++ development is substantially right-shifted, which indicates that all corrections, simple and complex, take significantly longer.

Overall, each C++ correction took more than twice as long to fix as each C correction. This could simply be because the C++ application area was more difficult. However, the right shift of the C++ curve compared with the C curve shows that even simple fixes take longer on average. This suggests that the implementation and not the application area contributed most to the differing error rates. For a different perspective on the data, Figure 3 shows the total number of hours spent in each of the categories for each of the two products.

Overall, project staff spent 1,341 hours correcting 94 defects in the C++ product, compared with 375.1 hours correcting 74 defects in the C product. The overall picture is shown in Table 2, including a calculation of the average time taken to fix a defect—first for all defects, then for defects taking no more than 40 hours. This was done to remove the effects of a small number of very difficult defects, which occurred in the C++ product but were never seen in the C product. Even after this adjustment, defects still took some 37 percent longer to fix in C++ than in C. For all defects, the average correction time is 260 percent longer in C++.

You can also see the right shifting of C++ corrective-maintenance costs compared with other languages in Watts Humphrey's data comparing C++ with Pascal.[7] Humphrey independently analyzed more than 70 Pascal and C++ programs and found the behavior shown in Figure 4, which is quantitatively similar to the case history presented here. Qualitatively similar comments comparing C++ and C were made to me in a personal communication by Walter Tichy.

To summarize, these two independent data sources suggest that corrective-maintenance costs are significantly larger in OO systems written in C++ when compared to conventional systems implemented in C or Pascal. Although two cases do not make a theory, the results' similarity seems significant. Given that the whole of OO appears to have existed so far in a measurement vacuum, these cases raise some disturbing questions that can only be answered by further measurement. If this pattern were to be repeated, then the claims certainly for C++ based OO systems, particularly with regard to ease of corrective change, look dubious.

OO AND HOW WE THINK

The data I've presented clearly give cause for concern and, considering the difficulty of corrective maintenance in C++ based OO development, call into question the validity of the paradigm's fundamental principle: that it is a way to think about systems that mirrors our natural thought processes.

Essential OO properties
Studying the copious OO literature, I find that the central features that comprise an OO system are somewhat ill defined. Although such systems differ in detail, all appear to encompass at least the following.

♦ *Encapsulation.* The essential properties of an

Table 2
Comparison of Defect Correction Times in the C and C++ Products

Implementation Language	Total Hours	Total Defects	Hours/defect
C, all defects	375.1	74	5.5
C++, all defects	1,341.0	94	14.3
C, all defects taking no more than 40 hours	311.1	73	4.6
C++, all defects taking no more than 40 hours	575.0	86	6.3

object can be isolated from the outside world. Only those properties that the outside world must know about are made visible.

♦ *Inheritance.* An object can inherit properties from another object. In OO, we think of a base object from which subobjects can inherit certain features, as well as adding their own. This can be extended to multiple inheritance, whereby a subobject can inherit properties from more than one base object.

♦ *Polymorphism.* The behavior of an object is context-sensitive: it acts differently in different circumstances.

OO practitioners assert that these properties mimic how we think about the world by defining it in terms of objects and their attributes. We must, however, clearly distinguish between the logical properties of a paradigm and the degree of error that stems from how we think about those properties.

Modeling human reasoning

So how do we reason logically? As with engineering, we have considerable empirical evidence from physiology to guide us in building basic models of the thinking process. In particular, the nature of logical thought, memory, and symbolic manipulation are highly relevant to programming.

There is very considerable physiological evidence, from studies of Alzheimer's disease for example, that favor the existence of a dual-level reasoning system based around a short- and long-term memory that exhibits significantly different recovery procedures.[8]

Short-term memory is characterized by rehearsal, finite size, and rapid erasure. Its effective size is, however, governed by the degree of rehearsal. As you
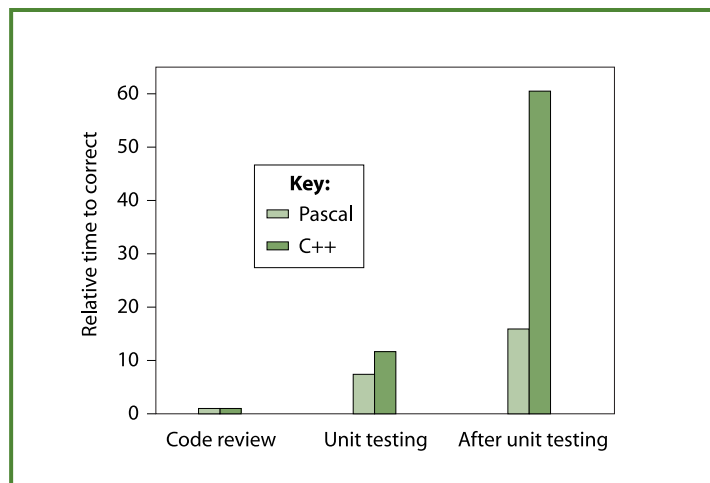

Figure 4. A comparison of the distributions of the times to fix faults in C++ and Pascal development. The C++ development is again right-shifted, indicating that both simple and complex corrections take significantly longer.

continually rehearse concepts, their representation becomes more compact preparatory to or perhaps coincident with their transfer to long-term memory, so that more can fit into the short-term memory.

Long-term memory is characterized by a storage mechanism based on chemical recovery codes, effectively limitless capacity, and longevity, in which recall is impaired only by breakdown in the recovery codes themselves, rather than in the embedded memory itself. Under certain conditions based on short-term rehearsal behavior, memories are transferred from short-term to long-term memory by a mechanism that encodes important properties in a more easily representable and diverse form, leading to the phenomenon whereby, for example, an entire summer vacation can be recalled from the stimulus of a particular fragrance. This mechanism is at the heart of accelerated learning techniques that exploit such encoding to improve the accuracy and persistence of recollections, and may have significant relevance to programming.
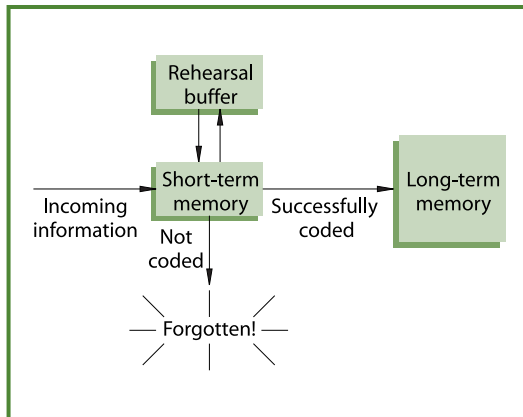
Figure 5. A simple schematic of the physiological properties of the human memory system.

Figure 5 shows this relationship.

A simple mathematical model of this system[9] suggests that any manipulation that can fit entirely into short-term memory will be far more efficient and error-free than those requiring access to long-term memory. Further, it appears to pay to keep short-term memory as full as possible by partitioning the problem into pieces that just fill it, without overflowing into long-term memory. This, it turns out, leads directly to a U-shaped defect density vs. component size curve as shown in Figure 6, a phenomenon widely observed in extraordinarily diverse software systems.[9,10] My work[9] also reinforces the observation previously made by others that defect density does not appear to be strongly dependent on language, with mainstream languages like Ada, C, C++, Fortran, Pascal, and assembler all yielding broadly similar defect densities when measured in defects per KLOC. These results also support the notion that defect introduction is strongly related to models of human reasoning, rather than to more abstract concepts.

The U curve simply tells us that, in any system, defects tend to accumulate in the smallest and largest components, leaving the medium-sized components most reliable. The existing data does not, however, tell us how to design a system, since the systems were all measured after the fact. No experiments have yet been done whereby a system has been deliberately constructed to exploit the U curve. Behavior models, such as the one based on human memory, can predict how to exploit the curve, but we must conduct further experiments to confirm or reject such predictions.

OO and the human mind

How do the principles of OO fit this model? First, there appears to be a natural fit with encapsulation. Encapsulation lets us think about an object in isolation and can thus be related to the notion of manipulating something in short-term memory exclusively. Further, the finite size of short-term memory suggests that objects should also be of limited size. However, encapsulation does not explicitly recognize the need to use short-term memory most efficiently by keeping it nearly full, which corresponds to the development of components in the medium-sized range. Programmers express surprising agreement about when they should use formal methods—that point at which a problem becomes sufficiently complex to require formal notational support. We can speculate that this indicates the point at which short-term memory overflows.

It is less easy to see how inheritance fits our memory model. The functions of long-term memory provide the best comparison. If an object has already been transferred to long-term memory, the transfer mechanism will have encoded important object properties in a more compact representation. If an object is then manipulated in short-term memory that possesses inherited properties previously encoded more efficiently in long-term memory, the properties' recall from long-term memory will be more efficient. However, access to long-term memory breaks the train of thought and is inherently less accurate. So this property of OO seems likely to be problematic. This is supported by a detailed study[11] of a large commercial C++ system consisting of some 133,000 lines. The study's results showed that components involved in inheritance contained six times more defects than components not thus involved, even though this system contained single inheritance only.

The third property, that of polymorphism, is potentially even more damaging. Objects with chameleon-like properties are intrinsically more difficult to manipulate, as they will by necessity involve pattern-matching similar behavior in long-term memory. The crossover of properties in short- and long-term memory relates to the apparently nonlocal relationship between cause and effect observed in an OO implementation, such as the complex set of matching rules that take place invisibly with function overloading in C++ through the mechanism of implicit actions. This mechanism has caused programmers difficulty in program comprehension and reading.

Overall, it seems that encapsulation at least partially fits how we think, but neither inheritance nor polymorphism do so as well or as naturally. The world may well be described conveniently in terms of objects and their properties. But OO is not naturally and self-evidently associated with the least error-prone way of reasoning about the world and should not be considered a primary candidate for a more effective programming paradigm. Given the simple model I've described, any paradigm that favors the natural behavior of the relationship between short-term and long-term memory, by exploiting the U curve in inspections for example, will likely lead to a significant reduction in defects, although this must still be tested experimentally.[9]

LESSONS LEARNED

We can derive the following lessons by reviewing the case history I've described.

♦ Owing to its measurement-confirmed role as the leading consumer of programmer resources, reliability in the form of corrective-maintenance cost is a strategically important goal and a significant indicator of engineering improvement.

♦ In the study I cited, an OO C++ implementation led to a system with somewhat poorer defect density when compared to an equivalent non-OO system in C. More significantly, comparable defects took between two and three times as long to correct because the whole distribution was right-shifted.

♦ The way defects appear in programming-language usage is qualitatively similar to the known properties of human short- and long-term memory, and the generally accepted principles of OO do not appear to match this model well.

Given the preceding points, you could only expect real corrective-maintenance gains from the higher defect density of the OO C++ implementation if a similar development in C would have taken much more code—that is, if C++ could lead to a corresponding compaction in code size for a given functionality. The data showed no evidence for this, however, partly due to the large amount of framework code that must be written in C++ to establish the object relationships. As a coda, the survey by Leach[12] showed that few users felt they achieved more than 20 percent reusability. In the study described here, both systems have comparable levels of reuse: around 40 percent. I have heard of C++
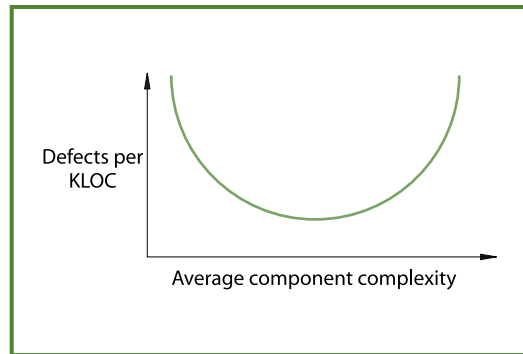

Figure 6. The U-shaped defect density vs. average component complexity curve observed in many software systems. Lines of source code is the most commonly used complexity measure in these experiments.

systems that claim to have reached much higher compaction levels, but the evidence is anecdotal and such claims rare.

Thus both the experiment I've presented and other independently derived data collectively cast considerable doubt on the belief that OO systems implemented in C++ offer a quantum leap forward. Mine is not the only study that shares such doubts. Michelle Cartwright and Martin Shepperd[13] raise several important and general concerns about OO in general, including the following.

♦ Despite many claims about OO software's ease of maintenance, supporting data is sparse and conflicting.

♦ The potential for complexity in OO systems gives rise to concern over their maintainability in practice.

♦ You cannot directly apply current maintenance research to the OO paradigm without adaptation. Because OO is so poorly defined in general, many conflicting views and hybrid approaches exist.

Given that improved maintainability is in question and that some 50 percent of all maintenance is corrective, Cartwright and Shepperd's study casts further doubt on OO's ability to provide improved corrective-maintenance costs. Finally, the study cites compelling evidence that most OO systems do not achieve their promise. The authors conclude with another survey of actual OO users who, perhaps surprisingly, still view the paradigm in a favorable light. Hope does indeed shine eternal.

The case history I've presented reveals that in comparable, parallel developments, no signif-

icant benefits accrued from the use of an object-oriented technique in terms of corrective-maintenance cost, and that the company views the resulting C++ product as much more difficult to correct and enhance. Further, this data is not unusual and compares closely to other published data.

Given OO's rather vague definition, it is easy to criticize developers for not understanding the paradigm properly, but such criticisms tend to come from those who do not measure their own efforts. It is often noted anecdotally that C++ class systems take time to perfect. With this in mind, the developers in the study I've presented were given considerable leeway to rewrite parts of the C++ product as it evolved. They made liberal enough use of this freedom to ensure that the experiment was not simply an inappropriate first design.

Finally, we must recognize that this case history does not distinguish between how much the reported problems relate specifically to OO and how much they relate to C++. Equivalent data on how other widely used OO languages—such as Smalltalk, Java, and Eiffel—compare to conventional systems seems rare to nonexistent. However, combining

♦ a significant study that shows inheritance appears to attract defects,

♦ the similarity of the data comparing C++ systems against two widely differing languages, C and Pascal, reported here;[11] and

♦ the reasoning models presented here,

the problem seems at least partially ascribable to OO itself. If so, significant progress may only be made in the key area of corrective-maintenance costs if we use paradigms that are sympathetic to limitations in human reasoning ability rather than to abstract paradigms. A good example of this might be the explicit exploitation of the empirically established U curve of defect density. Whatever direction we take, any attempt to improve in the absence of measurable feedback seems doomed to fail, however much fun it may be. ❖

REFERENCES

1. F.P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Apr. 1987, pp. 10-19.
2. D. Harel, "Biting the Silver Bullet: Toward a Better Future for Software Development," *Computer*, Jan. 1992, pp. 8-24.
3. R.S. Arnold, *On the Generation and Use of Quantitative Criteria for Assessing Software Maintenance Quality*, doctoral dissertation, Univ. of Maryland, College Park, Md., 1983.
4. D. Crawford, "Object-Oriented Experiences and Future Trends," *Comm. ACM*, Vol. 38, No. 146, 1995.
5. L. Hatton, *Safer C: Developing for High-Integrity and Safety-Critical Systems*, McGraw-Hill, New York, 1995.
6. J. Musa et al., *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
7. W.S. Humphrey, *A Discipline of Software Engineering*, Addison Wesley Longman, Reading, Mass., 1995.
8. F.I.M. Craik and R.S. Lockhart, "Levels of Processing: A Framework for Memory Research," *Key Studies in Psychology*, R.D. Gross, ed., Hodder & Stoughton, London, 1990.
9. L. Hatton, "Re-Examining the Fault Density–Component Size Connection," *IEEE Software*, Mar./Apr. 1997, pp. 89-97.
10. D. Card and R.L. Glass, *Measuring Software Design Complexity*, Prentice Hall, Upper Saddle River, N.J., 1986.
11. M. Cartwright and M. Shepperd, *An Empirical Investigation of an Object-Oriented Software System*, tech. report TR96/01, Dept. of Computing, Bournemouth Univ., U.K., 1997.
12. E. Leach, "Object Technology in the U.K.," *CSM '95*, CSM, Durham, England, 1995.
13. M. Cartwright and M. Shepperd, "Maintenance: the Future of Object-Orientation," *CSM '95*, CSM, Durham, England, 1995.

About the Authors
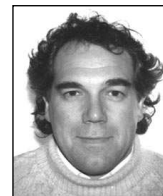
Les Hatton is a managing partner at Oakwood Computing. Formerly he was director of research for Programming Research Ltd. As a geophysicist he was awarded the European Conrad Schlumberger award in 1987, but now specializes in software safety. He is the author of *Safer C: Software Development in High-Integrity and Safety-Critical Systems* (1995) and is currently working on a new book, *Software Failure: Avoiding the Avoidable and Living with the Rest*.

Hatton received a BA and an MA from King's College, Cambridge, and a PhD from Manchester, all in mathematics. He is a fellow of the British Computer Society.

Address questions about this article to Hatton at Oakwood Computing, Oakwood 11, Carlton Road, New Malden, Surrey, KT3 3AJ, UK; lesh@oakcomp.co.uk.