"Does OO really match the way we think ? "

by

Les Hatton

Oakwood Computing, Oakwood House, 11 Carlton Road

New Malden, Surrey, U.K., KT3 3AJ

Version #1.5, 2nd. Dec, 1997

Abstract

Of the many revolutions which have captured the hearts of programmers, Object-Orientation (OO) has been arguably the most forceful and alluring. Its central premise appears to be that 'it matches the way we think about the world', and therefore self-evidently points the way forward for all future development.

This polemic paper develops the argument that reliability in the form of corrective maintenance cost is the most important improvement goal and goes on to expose the OO model to recent corrective maintenance data both for big systems in general and for OO systems in particular, in this case all implemented in C++. These data suggest that no particular advantage accrues from the use of such techniques, and that there is strong evidence to suggest that defects are harder rather than easier to correct, although it was not possible to apportion blame between the OO paradigm itself and its C++ implementation. Finally, the paper considers whether the basic premise of OO is well-supported by considering widely-accepted models of how we think in the context of programming systems, and how well the OO model matches these models.

The paper concludes that OO may have got some of this right but that any paradigm which tends to lead to a system being decomposed into large numbers of small components as is frequently the case in OO systems as well as conventional systems, is fundamentally wrong, and that we should therefore expect no particular benefits to accrue from an OO system than from a non OO system. In the case studies discussed here, it appears that at best OO is simply a different paradigm but is emphatically not better, and it is argued that significant improvement will only follow from paradigms which specifically acknowledge the strengths and weaknesses of the human reasoning system. By far the biggest concern is that corrective maintenance costs, which already dominate the software life-cycle, look set to increase significantly, without any easily quantifiable benefit accruing.

Engineering and the Quantum leap

In the history of engineering, there are two ways to make progress. Progress can be either slow, incremental and deriving from measurement-based improvement, or it can be the result of a quantum leap, the so-called 'silver bullet'. In other branches of engineering, a mixture of the two is usually found, with occasional quantum leaps interspersed with long periods of consolidation carefully based on measurement. Examples of quantum leaps in civil engineering include the arch

keystone and the invention by the Romans of cement. Software engineering ought to be the same, but regrettably, far more is expected from the quantum leap because there so little emphasis on measurement-based incremental improvement. The many failures of the search for the 'silver bullet', (c.f. (Brooks 1986), (Harel 1992) does not mean to say that there are none, but conventional engineering should teach us that they are few and far between and without measurement, we may not be in a position to exploit them as we would not recognise them for what they were.

OO represents one of the latest, perhaps the most significant and certainly the most expensive attempt to find a silver bullet, and now dominates recent software engineering research and teaching to such an extent that if it is found to be wrong, even partially, a huge amount of money would yet again have been thrown away on a whim, or even worse, we would not be able to throw it away at all.

Before we can establish its true nature, we should ask the question as to what precisely does a silver bullet mean in terms of software engineering ? Over the last few years, some organisations and groups have assiduously amassed data on software systems to the extent that such data contains powerful messages which we should no longer ignore. The central questions that we currently ask about software concern:

- *Reliability*. It is well accepted from numerous sources that around 80 ± 10% of all the money we spend on software is spent on maintenance after the software is first released to its intended users. The phase prior to its release we call development. Furthermore, major studies such as conducted by (Arnold 1983) strongly suggest that *corrective* maintenance is the most expensive component amounting to some 50% of all maintenance. The other 50% is ascribed to the other two recognisable components of *adaptive* maintenance, (improvements based on changed behaviour) and *perfective* maintenance (improvements based on unchanged behaviour). In other words, we have the unpleasant truth that for every hour we spend developing a system, we spend 2 hours correcting it in its life-cycle, *and that corrective maintenance appears to be the most significant component of the life-cycle at around 40%, as indicated in Figure 1.* Clearly, any silver bullet would have to have a huge beneficial effect on the corrective maintenance cost.
- *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, depends directly on the ability to carry out the essential tasks of development or the three components of maintenance. It does not really make sense to ask about productivity, which is merely a derived activity. For example, the very rapid production of inferior code could not conceivably be called productivity because of the nature of Figure 1.

- *Ease of modification*. This is simply a measure of how efficiently maintenance of the various kinds described above can be carried out.
- *Re-use*. This involves the notion of re-using parts of a design or code in a system for which they were not originally built. It leads to many interesting questions and is primarily aimed at the development phase. Re-use has been a feature of software development since the quantum leap of separate compilation, a notion understood as early as Turing in the '40s and '50s, allowing the emergence of libraries of sub-components for specific tasks such as numerical algorithms in Fortran. Judging by detailed surveys such as that done by the ACM in (Crawford 1995), apart from GUI building, re-use is not an outstanding success with C++ in spite of this being the second most common reason for switching to the language. (In our experience, by far the most common reason is regrettably that the developer can put the experience on his or her CV (Resume), a rather unsatisfactory reason.). In contrast, we can certainly recall achieving very high levels (> 90%) of re-use using scientific library routines called from Fortran 20 years ago.

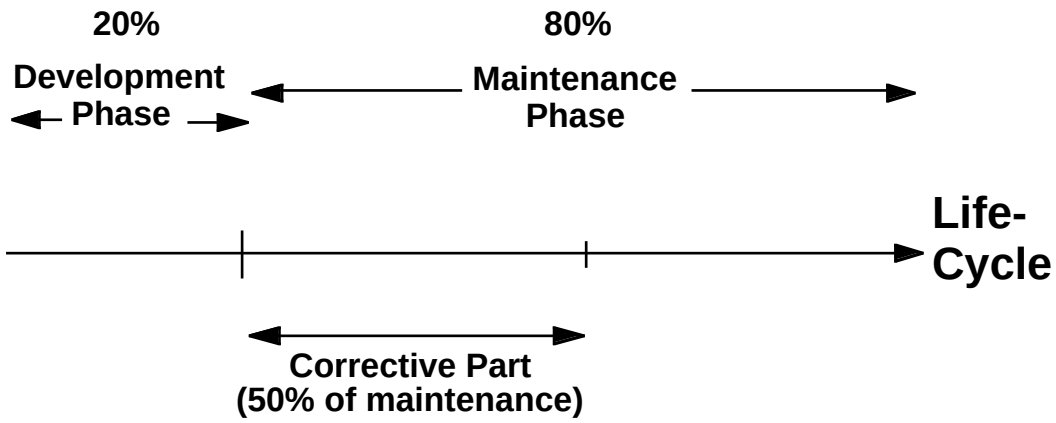

Figure 1: A simple picture indicating the relative costs of components of the life cycle. From these figures, there is no doubt that corrective maintenance is of strategic importance.

Since a silver bullet by definition should lead to a significant improvement, the primary manifestations should be on a massive reduction in maintenance, by far the biggest component of the life-cycle, and in turn its biggest component, that of corrective maintenance, rather than an unnatural obsession with the relatively minor development phase.

A case study in OO

In this section, we will present a recent case history of an OO development in which corrective maintenance cost was tracked in detail as well as other maintenance issues, and will then compare it with other recently published data.

The case study was carried out during 1995 on the 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 run-time failure. The tools are very widely used and corrective maintenance cost is an important consideration.

In this company, all software development has been orchestrated by a strict, Change, Configuration and Quality Control System since April 25th 1991 yielding a complete audit trail including defect analysis. The system is completely automated using an in-house developed product whose design goal was to automate significant parts of levels 2 and 3 of the Carnegie Mellon Capability Maturity Model, as well as providing data for some parts of levels 4 and 5. Development is carried out in a UNIX environment on Sun and HP workstations using various C and C++ compilers.

During the last 6 years, amongst other products, the company developed a deep-flow analysis tool for C and written in C, and then a similar tool for C++ written in C++. The C++ tool is based on close adherence to the emerging ISO C++ standard. The parsers for these two languages were of a similar size in line-counts, which at the time of the project measured as non-blank non-comment source lines came to:

- C parser: 44,700 lines of C.
- C++ parser: 52,500 lines of C++.

although the latter has grown very significantly since the study to incorporate the full language and is now much bigger than the C parser. Probably the single most

important feature of this case study for the purposes of this discussion, is that both products were written by the same highly-skilled staff, and yet both 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 1 token-lookahead yacc grammar along with support routines. Note that 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 intractability of parsing C++ using 1 token-lookahead grammars. In other words, they are very different products, albeit by the same authors, who would be considered expert in the implementation languages, (with several years experience in each) and also in OO design.

Another important factor in common is that safe well-defined subsets of C and C++ were used following guidelines published in (Hatton 1995) for example, although the subset for C was rather stricter than that for C++ during the course of the development. These subsets were enforced *automatically* using the toolsets themselves, and together constitute the effective programming standards. New code was required to conform absolutely and existing code was required to improve incrementally until it met the absolute standard. This also was enforced automatically.

The study involved the analysis of the company's entire change, configuration and quality control history between 25th April, 1991, when it was formally instigated, and 1st. August, 1995, when the study terminated. For software experimentalists, this system is automated and has the following desirable properties:

- *Every* request for any kind of change, corrective, adaptive or perceptive received internally or externally, is entered into the system as a Change Request, (CR), where it is assigned a unique number which is available for reference to the originator. This number can be used to track the progress of the CR at any time. The first operational CR formally entered was CR 53 and at the time of completion of the study, the most recent was CR 2935. Software release notes specifically reference the CR's resolved in that release.
- Corrective change accounts for only 12% of all CR's, (349 out of 2833). The industry average is thought to be near 50% as was described in the overview, (Arnold 1983). 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.
- CR's automatically go through a number of states as follows:

EVALUATION

-> IMPLEMENTATION

-> TESTING

-> LOADING

-> DISSEMINATION

-> CLOSED

- CR's 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. CR's can only be assigned by a project co-ordinator.

The defect rates of the various products as of August 1995 are shown in Table 1.

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

Table 1. Defect rates of various products in the case study described here.

Note that all defects are logged however minor and are included in the above statistics.

It may be noted that the industry standard for good commercial software is around 6 per KLOC in an overall range of around 6-30 per KLOC, e.g. (Musa, Iannino. et al. 1987). In addition, very few systems have ever got below 1 per KLOC this far into the life-cycle, even in safety-critical systems. Given that the above products are in extensive use at several hundred sites around the world and that parsing technology is considered to be a difficult application area, these statistics certainly compare very well with the population at large. Note finally that the average age of the above products since first release is between 3 and 4 years so the fault densities represent mature software.

The overall impression of this development environment then is that it is well-controlled, formalised and produces rather lower defect densities than average products.

Results

The Change Requests were used in two ways:

- To compare defect densities in the C parser and the C++ parser.
- To compare the distributions of times to correct defects in the two products.

Comparison of defect densities

The defect densities are very similar although the defect density of the OO C++ development is already a little higher and is climbing rather more rapidly than the conventional C development. It is currently around 25% higher.

Comparison of distribution of correction times

This was most revealing and simultaneously the most disturbing. One of the arguments in favour of OO systems is that they are easier to change 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 data from the beginning of the IMPLEMENTATION phase to the beginning of the LOADING phase. It was observed that failures were much harder to trace to 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 non-local relationship between cause and effect. In other words, the manifestation of a failure may be ‘a long way away’ from the fault which 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.

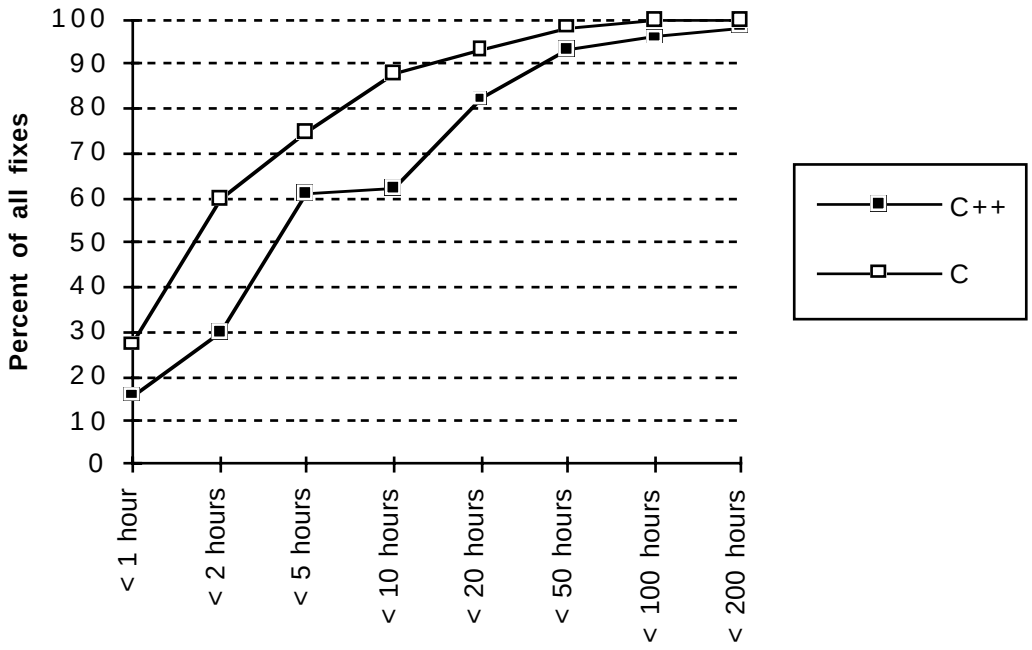

Figure 2: A comparison of the distributions of the times to fix faults in comparable C++ and C development. To read the graph, note that for the C development, 60% of all fixes took less than 2 hours compared with only 30% of all fixes in the C++ development. The C++ development is substantially right-shifted, indicating 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. Now this could simply be because the C++ application area was more difficult. However, the figure shows clearly that the C++ curve is substantially *right shifted* compared with the C curve which shows that even simple fixes take longer on average. In other words, it is less likely to be the application area than its implementation. 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.

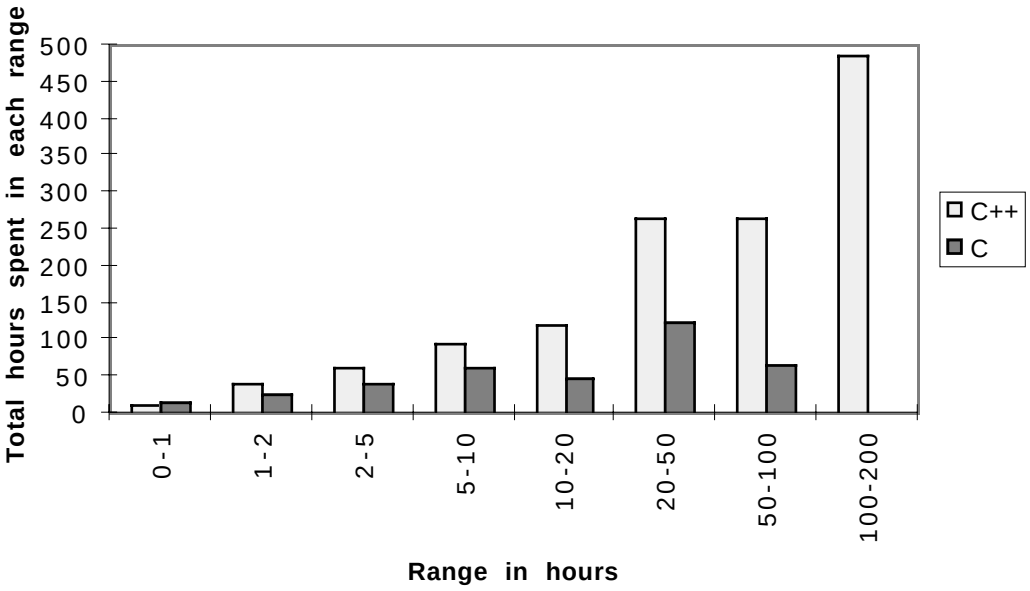

Figure 3: The total number of hours spent in corrective maintenance for each time range for each product. For example for defects taking between 20 and 50 hours to correct, about 270 hours was spent on the C++ product and about 140 on the C product.

In all 1,341 hours were spent correcting 94 defects in the product written in C++ compared with 375.1 hours correcting 74 defects in the product written in C. The overall picture is shown in table 1 including a calculation of the average time taken to fix a defect for all defects, and also for defects taking no more than 40 hours. This was done to remove the effects of a small number of very difficult defects which tend to occur in the C++ implemented product but were never seen in the C implemented product. Even when this is done, defects still took around 37% longer to fix in C++ than in C. For all defects, the average correction time is 260% longer in C++.

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

Table 1: Comparison of defect correction times in the C and C++ implemented products.

The right shifting of C++ corrective maintenance costs compared with other languages can also be seen in data comparing C++ with Pascal shown by (Humphrey 1995) who independently analysed more than 70 Pascal and C++ programs and found the behaviour shown in Figure 4, which is quantitatively very similar to the case history studied here.

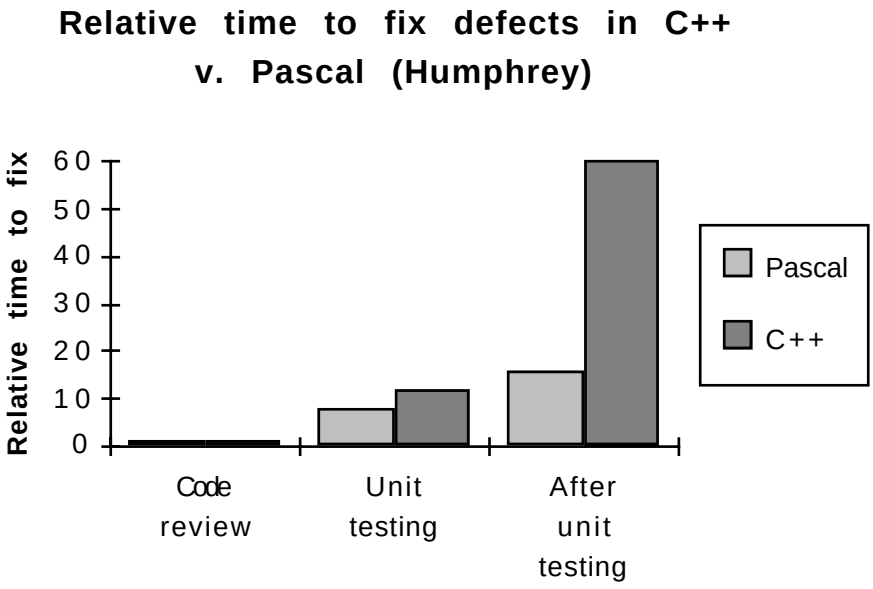

Figure 4: A comparison of the distributions of the times to fix faults in C++ and Pascal development reported by (Humphrey 1995) who independently analysed a considerable number of Pascal and C++ programs. The C++ development is again substantially right-shifted, indicating that all corrections, simple and complex take significantly longer.

Finally, the effect has also been noted by (Tichy 1996) comparing C++ and C.

To summarise, we have described two independent sources of data, one of which is presented in detail, which suggest that corrective maintenance costs are significantly larger in C++ implemented OO systems than in conventional systems implemented in C or in Pascal. Clearly two independent cases do not make a theory and we do not claim otherwise, however the similarity of the results is interesting and given that the whole of OO appears to have existed so far in a measurement vacuum, there are some disturbing questions which can *only* be answered by further measurement. If this pattern were to be repeated, then the claims of OO particularly with regard to ease of corrective change look very dubious indeed.

OO and the way we think

The nature of the data presented in the previous section is clearly of concern and so in this section, we will speculate on the validity of the fundamental principle of OO that it is a more natural way to think about systems.

Studying the copious literature of OO, the central features which define an OO system seem a little ill-defined. There seems to be an absence of agreement about detail, however they appear to encompass at least the following:

- *Encapsulation.* This concept simply means the ability to isolate the essential properties of an object from the outside world. In other words, only those properties which the outside world needs to know are made visible.
- *Inheritance*. This refers to the ability of an object to inherit properties from another object. Thus in OO, we think of a base object from which sub-objects can inherit certain features, as well as adding their own. This can be extended to multiple inheritance whereby a sub-object can inherit properties from more than one base object.
- *Polymorphism*. This is the notion whereby the behaviour of an object is context-sensitive. In other words, it can behave differently in different circumstances.

These properties are said by practitioners to mimic the way we think about the world, i.e. in terms of objects and their properties. We must however, clearly distinguish between the logical properties of a paradigm and the degree of error-proneness in the way we think about those properties.

So how do we reason logically ? Well, in common with engineering, there is 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 of relevance to the activity of programming.

There is very considerable physiological evidence, for example from studies of Alzheimer's disease, in favour of the existence of a dual level reasoning system in humans based around a short-term memory and a long-term memory with significantly different recovery procedures, (Craik and Lockhart 1990). *Short-term memory* is characterised by rehearsal, finite size and rapid erasure. The effective size is however governed by the degree of rehearsal. As concepts are continually rehearsed, their representation appears to become more compact preparatory to or

perhaps coincident with the transfer to long-term memory, so more can fit into the short-term memory. *Long-term memory* is characterised by a storage mechanism based on chemical recovery codes, effectively limitless capacity and longevity, where recall is impaired only by breakdown in the recovery codes themselves rather than the embedded memory itself. Under certain conditions based on the short-term rehearsal behaviour, memories are transferred from short-term to long-term memory by a mechanism whereby important properties are encoded in a more easily representable and diverse form, leading to the observed phenomenon whereby an entire summer vacation can be recalled from the stimulus of a particular fragrance for example. This diverse mechanism is at the heart of accelerated learning techniques which exploit such diversity to improve the accuracy and persistence of recollections, and may have significant relevance to programming. The relationship is exemplified by Figure 5.

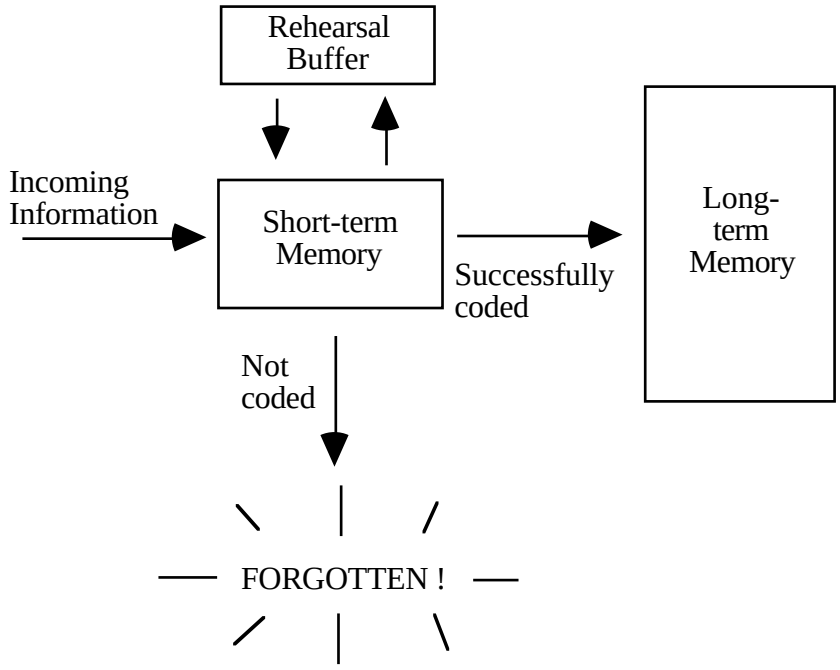

Figure 5: A simple schematic of the physiological properties of the human memory system.

A simple mathematical model of this system (Hatton 1997), suggests that any manipulation which can entirely fit into the short-term memory will be far more efficient and free of error than those which require access to the long-term memory. Furthermore, it appears to pay to keep the short-term memory as full as possible by partitioning the problem into pieces which just 'fill' short-term memory without overflowing into long-term memory. This it turns out, leads directly to a U-shaped defect density v. component size curve as shown in Figure 6, a phenomenon widely observed in extraordinarily diverse software systems, (Card and Glass

1987) , (Hatton 1997). The latter paper 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 measured in defects per KLOC, (1000 lines of code), which also supports the notion that defect introduction is strongly related to models of human reasoning, rather than more abstract concepts.

The U curve simply tells us that in any system, defects will tend to accumulate in the smallest and largest components, leaving the medium-sized components as the most reliable. Note that the existing data does not 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. The existing data does however tell us how to *inspect* a system after the fact in the knowledge that these are the conditions under which the U-curve was observed. Models of this behaviour such as the one based on human memory described here can of course make predictions but these predictions will have to wait for further experiments to be confirmed or rejected.

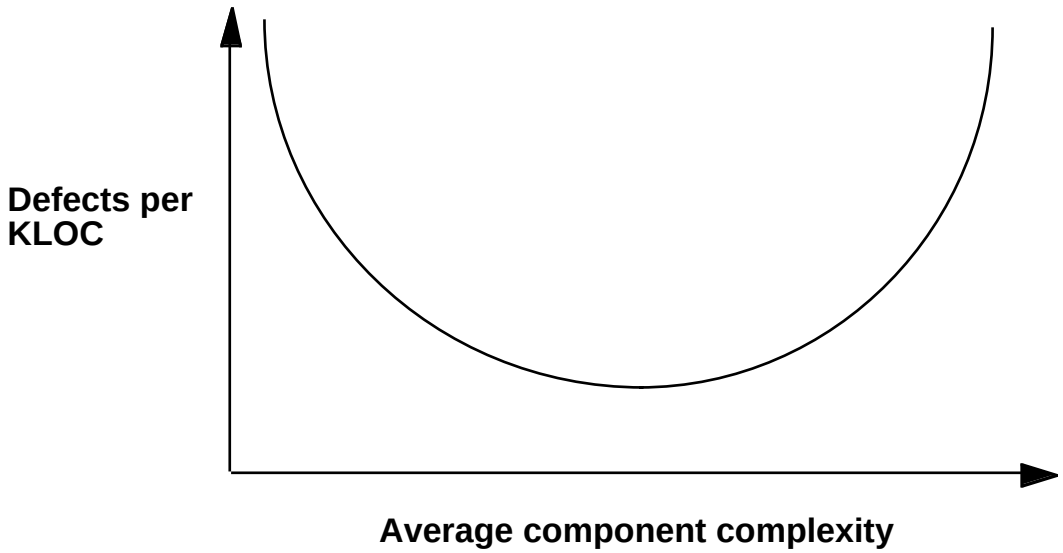

Figure 6: The U-shaped defect density v. average component complexity observed in many software systems. Note that number of lines of source code is the most commonly used complexity measure in these experiments.

Given these comments, how do the principles of OO fit into this model ? First of all, there appears to be a natural fit with encapsulation. Encapsulation implies the ability to think about an object somewhat in isolation and can be related to the notion of manipulating something in short-term memory exclusively. Furthermore, the finite size of the short-term memory suggests that this requires that objects

should also be of limited size. However encapsulation does not explicitly recognise the need to use the short-term memory most efficiently by keeping it nearly 'full', which would correspond to the development of components in the medium-sized range. It is interesting to note in discussions with programmers about when they should use formal methods that there is surprising agreement when a problem is sufficiently complex to require some form of formal notational support, and we can speculate that this is evidence of the notion that the short-term memory is 'over-flowing'.

It is a little less easy to see how inheritance fits, although the following argument can be used. If an object has already been transferred to long-term memory, the mechanism of the transfer will have encoded important properties of the object in a more 'compact' representation. If an object is then being manipulated in short-term memory and it possesses inherited properties previously encoded in a more efficient form in long-term memory, their recall from long-term memory will be more efficient. However, access to long-term memory, breaks the 'train of thought' as discussed above and is inherently more inaccurate. So this property of OO seems likely to be problematic. This is supported by the results of (Cartwright and Shepperd 1997), who in a detailed study of a large commercial C++ system comprising some 133,000 lines, found that *components involved in inheritance contained 6 times more defects than components not involved in inheritance*.

The third property, that of polymorphism, is potentially even more damaging. Objects with chameleon-like properties, i.e. polymorphic objects, are intrinsically more difficult to manipulate as they will by necessity involve pattern-matching similar behaviour in the long-term memory. This is related to apparently non-local behaviour in an OO language, for example, the complex set of matching rules which take place invisibly with function overloading in C++ through the mechanism of implicit actions, which has been observed to cause programmers difficulty in program comprehension and reading.

To conclude, it would seem that encapsulation does at least partially fit with the way we think, but that neither inheritance nor polymorphism fit so naturally. In other words, even though the world may well be described conveniently in terms of objects and their properties, OO is not naturally and self-evidently associated with the least error-prone way of *reasoning* about the world, and should not necessarily be considered a primary candidate for a quantum leap. We can go on to conclude from this simple model that any paradigm which favours the natural behaviour of the relationship between short-term and long-term memory, for example by

exploiting the U-curve in inspections, *is* likely to lead to significant improvement in the sense of reduced defects although this will have to be tested experimentally. Note that the paper by (Hatton 1997) made predictions which could be tested experimentally.

Conclusions

The following conclusions can be made:

- owing to its measurement-confirmed role as the leading consumer of programmer resource, reliability in the form of corrective maintenance cost is a strategically important goal and a significant indicator of engineering improvement,
- in the study cited here, an OO C++ implementation led to a system of somewhat poorer defect density to an equivalent non OO system in C, and more significantly, comparable defects took between two and three times as long as long to correct as the whole distribution was right-shifted. Both systems were the work of the same experienced staff with expert language skills in both languages. Very similar results were reported in studies comparing Pascal against C++.
- the way defects appear in programming language usage are qualitatively very similar to known properties of the human short-term / long-term memory, and the generally accepted principles of OO do not appear to match this human memory model particularly well,

Given the above, real corrective maintenance gains in terms of the overall number of faults would only be gained from the higher defect density of the OO C++ implementation if a similar development in C would have taken much more code, in other words, if C++ could lead to a corresponding compaction in code size for a given functionality. There was however no evidence for this, partly due to the large amount of framework code which has to be written in C++ to set up the object relationships. As a coda, it is interesting to reflect in the survey by (Leach 1995), that very few users felt they had got beyond 20% re-usability. In fact, in the study described here, re-usability is tracked and both systems have comparable levels of re-use at around 40%. We have heard of C++ systems which claim to have reached much higher compaction levels, however this evidence is anecdotal and they seem few and far between.

It can be inferred that the above experiment along with other independently derived data casts considerable doubt on the belief that OO systems implemented in C++ are a uniform quantum step forward. This is not the only study which shares such doubts. The paper (Cartwright and Shepperd 1995) raises a number of important concerns on OO in general. They cite these as:

- There are many claims concerning the ease of maintenance of OO software, but such data as there is, is conflicting.
- The potential for complexity in OO systems gives rise to concern over their maintainability in practice.
- Current maintenance research is not directly applicable to the OO paradigm without adaptation. In fact OO is so poorly defined in general, that there are many conflicting views and many different hybrid approaches.

In other words, since improved maintainability is at question and since some 50% of all maintenance is corrective as discussed in the overview to the current paper, then improved corrective maintenance costs are also questioned by their study. Finally, it also quotes that the majority of evidence on OO systems suggests that they do not achieve their promise and finishes up with another survey of actual OO users who perhaps surprisingly, still view it in a favourable light. Hope does indeed shine eternal in the human breast.

We can conclude by saying that this case history revealed that in comparable parallel developments, no significant benefits accrued from the use of an object-oriented technique in terms of corrective maintenance cost, and the resulting C++ product is viewed within the company as *much* more difficult to correct and enhance. Furthermore, the data is not unusual and compares well with other published data. Although it is easy to criticise developers for not understanding the OO paradigm properly in view of its rather vague definition, such criticism tends to arise from people who do not measure their own efforts, and we look forward to seeing much more data published on this vexed issue. Finally, it is often noted anecdotally that C++ class systems can take a little time to get ‘right’. With this in mind, the developers were given very considerable leeway to re-write parts of the C++ development as it evolved. This freedom they made use of so that the experiment described here was not simply an inappropriate first design. Of course any design OO or otherwise can be made better by successive refinement.

Finally, it must be recognised that this case history is unable to distinguish the degree to which the reported problems are specifically related to OO itself or to the

implementation language used in the case studies described, which was C++ in each case. Comparable data on other widely used OO languages such as Smalltalk compared against conventional systems seems rare to the point of non-existence and we are aware of a certain amount of gentlemanly dispute in this area, even though OO is no longer a 'new' idea. However, taken with the significant study showing that inheritance appears to attract defects, the similarity of the data comparing C++ systems against two widely-differing languages, C and Pascal, and finally the reasoning models presented here, we would argue that the problem is at least partially ascribable to OO itself and that significant progress will only be made in the key area of corrective maintenance cost using paradigms which are human memory-oriented rather than object-oriented, for example, by explicit exploitation of the empirically established U-curve of defect density. Whatever direction is taken, any attempt to improve in the absence of any feedback from measurement, seems doomed to fail, however much fun it is.

Acknowledgements

We would like to acknowledge the unfailing co-operation of the development staff at the case history company for their assistance in compiling the data. Walter Tichy also shared many important insights from his wide experience in this area. Finally, the paper had an unusually large number of anonymous reviewers each of whom we would like to thank.

References

Arnold, R. S. (1983). On the Generation and Use of Quantitative Criteria for Assessing Software Maintenance Quality, University of Maryland.

Brooks, F. P. (1986). No Silver Bullet: Essence and Accidents of Software Engineering. Info Proc' 86, Elsevier.

Card, D. and R. L. Glass (1987). Measuring Software Design Complexity, Prentice-Hall.

Cartwright, M. and M. Shepperd (1995). Maintenance: the future of object-orientation. CSM'95, Durham, England.

Cartwright, M. and M. Shepperd (1997). An Empirical Investigation of an Object-Oriented Software System, Dept of Computing, Bournemouth University, U.K.

Craik, F. I. M. and R. S. Lockhart (1990). Levels of processing: a framework for memory research. Key studies in psychology. R. D. Gross. London, Hodder & Stoughton.

Crawford, D. (1995). Object-Oriented Experiences and Future Trends. Comm. ACM., ACM Press. **38:** 146.

Harel, D. (1992). "Biting the Silver Bullet: Toward a better future for software development." IEEE-CS **25**(1): pp 8-24.

Hatton, L. (1995). Safer C: Developing for High-Integrity and Safety-Critical Systems., McGraw-Hill.

Hatton, L. (1997). "Re-examining the fault density - component size connection." IEEE Software **14(2)**(March/April 1997): p. 89-97.

Humphrey, W. S. (1995). A discipline of software engineering, Addison-Wesley.

Leach, E. (1995). Object technology in the U.K., 1995. CSM'95, Durham, England.

Musa, J., Iannino., et al. (1987). Software Reliability: Measurement, Prediction, Application., McGraw-Hill.

Tichy, W. (1996). Personal communication.