The use and abuse of a software engineering system

by D. J. PEARSON
Bell-Northern Research
Ottawa, Ontario, Canada

In 1969, International Computers Limited of England set about the design of its 2900 Series which was to unify the primary thrust of the company and was to provide a hardware and software architecture which was, at least, state-of-the-art. The systems also had to sell. Therefore, they had to satisfy the then market requirement for rich facilities and generally neat features. If ICL was seriously to compete with IBM (in Europe at least) the operating system, subsequently called System VME/B, would have to be comparable with the IBM products. ICL could not afford the staggering investments made in the 360 software systems. On the other hand, its recent track record had been less than outstanding for such software development. Faced with this dilemma, a project was set up to develop a system capable of minimizing the problems of software development by harnessing many of the current software engineering philosophies in order to aid management, reduce error rate and increase productivity. This system was subsequently called CADES (Computer Aided Development and Evaluation Systems). This paper describes briefly the major facets of this system and then goes on to discuss six years of product development experience with a software engineering system which was, by the standards of the day, state-of-the-art.

THE SOFTWARE ENGINEERING SYSTEM

The original CADES system was based on System 4 and was implemented between 1970 and 1972. All the original VME/B software was itself implemented on the System 4 and written in a high-level language based on Algol 68 called S3. It was not until 1973/74 that a real transition to 2900 architecture took place as far as the product development activities were concerned. CADES itself was redesigned and reimplemented on 2900 during the period 1974–76. CADES consists of the following elements:

1. Formal Design Methodology
2. Design Definition Language
3. Product Data Base
4. Formal Data Capture and Control
5. Product Data Base Applications

Formal design methodology

A methodology called Structural Modeling was defined and adopted. It was based on a formalized top-down, levels of abstraction approach with great emphasis being placed on the data-driven emergence of design, and attempting to quantify the iterative nature of design. Instead of the “design until you understand the problem, code until you realize you don’t, then iterate” approach to design, the entire process was quantified and documented, with all the possible iteration paths identified and priorities assigned to them.

Using Structural Modeling, the designer was constrained first of all to analyse the problem in terms of information flow and to structure this information analysis into a tree form, each level on the tree defining an abstract machine. One of the lower levels of abstract machine would map onto the S3 compiler data structures. This was the implementation level. After this information analysis, a function tree was constructed, compatible with the information tree, each level on this function tree representing the functional definition of that abstract machine. This was called the holon tree (holon, from Koestler, Reference 3; in this context a holon is defined as a unit of further design).

A lower level of the holon tree would map onto the concept of an S3 module. This was the implementation level. When both trees were compatible, that is, complementary from an abstract machine, levels-of-design point of view, the top-down design process was commenced, expressing the functional design of each holon, level-by-level, in terms of the information items at the corresponding level in the data tree and its interactions in terms of its peer-holons. These designs were defined in terms of the design definition language. Techniques were built into the modelling process to preserve the data and functional modularity, and to minimize the functional and data connectivity across the system.

Design definition language

The emerging design was expressed in a formal System Descriptive Language, SDL. The structure of SDL was based on that of the implementation language S3. However,
it was provided with facilities to express design concepts such as an 'event' and a 'virtual machine,' and it contained no facilities for implementation concepts such as data structures and procedure declarations. An SDL definition dealt exclusively with the items defining the abstract machine under definition, and the conditions and assertions governing its function in terms of the recognized design concepts. When the designer, during his top-down design activities, reached the implementation level and started to define that in SDL, an automatic code generator was initiated and S3 code automatically generated from an SDL definition.

The advantages of using SDL were fourfold:

1. It ensured a formal, complete definition of the entire design.
2. Design definitions were machineable and could be captured in a data base.
3. Automatic code generation increased productivity.
4. Automatic code generation dramatically reduced finger-trouble error.

Product data base

One of the primary intentions of the CADES system was the construction of a product data base which would contain the entire set of information defining the VME/B product, from earliest design statement, S3 code and loadable versions, product releases, bug reports and fixes. One major reason for this was in order to be able to relate problems discovered in the field to the appropriate earlier design decision and hence define accurately the scope of such problems, rather than adopting the fire-fighting, piecemeal approach to product maintenance. The greater the contents of the product data base, the more valuable it was in terms of information inversions. It reached maximum value at product release, and subsequently proved to be invaluable throughout all product maintenance activities. The data base itself is able to contain all types of VME/B product definition, as shown in Figure 1.

Updates were made to the data base using the formal data capture mechanism and expressed in SDL. Similarly, retrieval requests were expressed in SDL. The code generator, compiler, construction, loading and maintenance tools interacted directly with the data base. This interaction enabled the mapping between ‘types’ of operating system representation to be carried out consistently. Note that the mapping between high- and low-level design representations is carried out by the designer. When a change was made to the data base, entries were recorded in queues for action by the appropriate tools. The tools themselves return information such as sizes and indications of success to the data base. This return information itself might then cause further entries to be made on the queues for appropriate processing. In this way a degree of automation was achieved, human interventions, and hence error rate, decreased.

Formal data capture and control

As will be explained in the next section, the size of the project team was large. Data base information integrity in such an environment is very important, and yet very difficult, to achieve. In order to achieve the necessary level of integrity we were forced to implement a semi-batch interface for data capture, with a rigorous control scheme to validate every piece of information to be inserted into the data base. This level of control was achieved by associating an au-
The Use and Abuse of a Software Engineering System

Figure 2 gives an idea of the CADES system as a whole. The four applications shown, Input Analysis, Product Information System, Code Generation and Compilation, and Construction and Maintenance were the most significant in terms of impact, and certainly in terms of success. Less successful applications such as simulation and test program generation were attempted and then subsequently fell into disuse due to lack of support.

The Input Analyzer was responsible for the syntax and semantic analysis of the design. It ensured that standards had been adhered to, that correct versions had been used, that the connectivity of the proposed design update was acceptable, etc. Less than full use was made of this checking stage. It would have been possible to code very rigorous and comprehensive checks in the Analyzer, and for the project manager to choose what levels of validation he was prepared to accept in a trade-off against expediency. This would have been possible and would have led to some interesting results. However, time always seemed to preclude this level of 'luxury.'

The Product Information System presented an easy to use retrieval interface to the designers, in either interactive or bulk-data mode. The service project was the main users of this interface to regularly supply managers and designers with current product details and statistics. The interface also provided answers to 'what if ..., ' 'who uses . . . ' and 'how did this happen' type of questions.

The Code Generation and Compilation application had a great impact throughout the product development. The application started from design specification in SDL at the implementation level, formed an S3 source code module...
from macros and primitives in the product data base, compiled it, and stored the subsequent S3 object code in the data base for subsequent collection. Because the designer did not deal with any sort of declaration and because he was constrained to make maximum use of the primitive macros stored in the data base, code production rates were increased very considerably and error rates dramatically reduced. Other advantages were that the code generator automatically inserted error checking and tracking code, and if, of course, imposed its own standards on the entire product coding.

The Construction and Maintenance application was driven by the master product version/release plan built into the data base. From this the system was able to construct from the object code modules, architectural details, etc., a set of load modules for export to a specific user site. The system kept track of user configurations, trouble reports, etc., and automatically correlated the most recent updates and fixes with the version currently in use on site.

THE ENVIRONMENT

Before discussing the results of using this CADES system on the VME/B product development, it will be useful to look at the target product and project involved.

VME/B is a facility-rich operating system with great emphasis placed on data management and job control language facilities. It supports a virtual machine architecture and is protected by 15 levels of hardware-implemented protection. A current typical release of VME/B represents, in total, between two and three million lines of S3 source code. Detailed design of the product was started in 1971 and the first version was running on 2900 architecture in 1973. The first customer release was made in 1975. The software was initially developed on System 4 machines, and then converted to run on a 2900 hardware simulator (i.e. 2900 architecture, old technology), before finally appearing on a true 2900 machine in 1974.

Until 1974 the project, about 200 people, was split over two sites which were two hundred miles apart. As the size of the project peaked the software engineering disciplines proved insufficient to handle this geographic split and the two teams were moved into one site. At any one time the CADES development and service group, including compiler development, represented about 15 percent of the total project team.

In the view of the above environment, and in view of the late 60s experiences by ICL in terms of operating systems development, the software engineering system was initially developed and controlled in order to promote the following VME/B attributes:

1. Highly structured, that is high modularity and low connectivity. This requirement initially dominated the performance requirements.
2. Ease of maintenance and enhancement.
3. High predictability and reliability.
4. Strong management and technical control over a large project team.
5. Subsequent performance improvements.

The next section describes how these attributes were achieved in practice.

THE PRAGMATICS

(Pragmatic—‘practical as opposed to idealistic’ (Webster’s New Collegiate Dictionary).)

When we set out to formulate and create the CADES philosophy and system we attempted from the start to be as practical as possible. We certainly adopted a rather basic engineering, rather than scientific, approach. This approach is described in detail in Reference 5. For instance, the form of the design language was based more on the facilities we wanted to remove from the programming languages as far as the designer was concerned than on more esoteric considerations concerning concise, formal definitions of the evolving design. It had to be usable, capable of quick, error-free production and easy to learn. Hence it ended up as being rather inelegant. But we could express emerging design in it, and we could generate S3 from it.

This approach to life pervaded the entire CADES system—methodology, language, data base, applications and service. The goal was to save the company as much product development money and lead times as possible. We set out early in 1970 not knowing a great deal about how we were going to do this and learned a great deal during the next seven years—both in a controlled learning way and also by bitter and expensive mistakes and experience. Some of the lessons are discussed below.

Economics and productivity

Over the seven-year period, the CADES system produced significant increases in programming productivity. At the end of the seven year period we were much better at producing code than at the beginning. Fred Brooks1 quotes ‘typical’ IBM code production rates classified by the complexity of the product being generated. These are:

Very few interactions—10,000 instructions per man-year.
Some interactions—5,000 instructions per man-year.
Many interactions—1,500 instructions per man-year.

He also quotes Corbato’s (MIT’s project MAC) overall production rate on MULTICS of 1200 lines of debugged PL/I per man-year. During the last three years of VME/B production when CADES represented a fully familiar, established and stable product, the production rate for the entire development team was around one module per man-month (four weeks). After generation, the average S3 source module was around 350 lines of code, derived from, perhaps, 100 lines of SDL. This represents a debugged programming rate of around 4500 S3 lines per man-year. This compares
favourably with the MULTICS rate and with the IBM rate for complex software. Hence, in terms of individual productivity CADES would appear to be at least a qualified success.

However, there is a more fundamental point here. In view of the ultimate size of the product and of the project, the experiences of the other manufacturers on products of similar size, and the problems we ourselves had during those seven years, it is very unlikely that the VME/B product could have existed at all without the CADES system, or an approach very akin to it. The power of CADES was that it forced software development activities, and in particular quantified design activities, into close proximity with rigorous management mechanisms. They presented sociological problems which will be discussed later. It did, however, provide us with an extremely well controlled and regulated large-scale software development.

Connectivity controls

One of the initial intentions in the CADES philosophy was to minimize and then subsequently control the evolution of the connectivity of the system. Connectivity is a major factor influencing the structure of a software product, the other being its modularity. Lehman has compared software structure to negative entropy in his Program Growth Dynamics studies. The Law of Increasing Entropy states: "The entropy of a system (which is the cross-product of connectivity and the reciprocal of modularity) increases with time unless specific work is executed to maintain or reduce it."

This is another statement of the well known Structural Decay phenomenon in which the difficulty of system modification increases with system release number. I believe that the secret of maintenance and enhancement, and hence economic product longevity, lies in mechanisms to monitor and control connectivity evolution. CADES had such mechanisms built into its methodology and database applications.

A measure of how successful we were can be gained by looking at a system attribute called Ripple Factor (RF) and calculating this for VME/B release sequences. The term Ripple Effect was defined by F.M. Haney and occurs most noticeably in systems with a large number of components. An enhancement or fix in one component causes, as a side effect, further necessary changes in other components, and possibly itself. The RF is the factor by which the work increases relative to that planned. Haney, via various sample systems, quoted RF of around 10 as being representative for a maturing software system of medium-to-high complexity. When a similar study was made of VME/B releases in 1975 the derived RF was found to be 1.4. It represents a remarkable improvement if Haney's figures are taken as representative. Subsequent experience in VME/B maintenance and enhancement would tend to confirm the fact that it is indeed relatively economic to enhance and develop now that it has reached a steady-state maturity. It remains to be seen whether the management controls will remain sufficiently rigorous to present the Law of Increasing Entropy finally dominating. The current signs are, however, that connectivity is still under control.

Law of management futility

Lehman states a law which he calls the Law of Statistically Smooth Growth. I prefer to call it the Law of Management Futility, and to state it thus:

Growth trend measures of global system attributes may appear stochastic locally in time and space, but statistically are cyclically self-regulating with well-defined longer-term trends.

I believe that this law is a very fundamental one and explains why so many large-scale software developments go astray because of too much misdirected, rather than too little, management attention. Lehman's law, and my experience, state that management fire-fighting at any stage in the software development process is likely not to work. The fighting and extinguishing of a local bush fire will only raise the temperature of, and the pressure on, another area, probably more critical and expensive, downstream in the development process.

CADES recognized this phenomenon and prevented its abuse in two ways. Firstly, the product database was based on a comprehensive, complex schema which defined the entire development process, from statement of market requirements through to steady-state product maintenance, and defined in great detail the entire set of interrelationships existing throughout the life-cycle. As a result, little management action could be taken in isolation without its downstream impacts being automatically defined in a suitable degree of detail. I believe this schema, and its evolution in future systems, is fundamentally important and says very significant things about the very nature of the software development process.

Secondly, the CADES system itself, and its control over VME/B production, had a noticeable inertia to violent changes in direction by management dictate. The greater the proposed change, the more noticeable was the inertia of the system. This may not seem to be a positive attribute of the system. However, Lehman's law says otherwise. Management actions should be smooth, controlled and fully cogniscent of the downstream implications of current action. The CADES system ensured this was true to a marked degree.

Sociological implications

It was obvious early in our formulation of the CADES principles that a large team would ultimately be employed on the VME/B development program. This one fact had a strong influence on the entire CADES system—its methodology, language and software. A decision was made that all aspects of the system would impose a rigid discipline and control over the project team, even at the expense of flexi-
多版本

在第一版的 CADES 中，我们非常重视解决大型系统开发生命周期中的多版本问题。作为结果，在新版本的 CADES 上，我们有了一个包含综合和强大机制的版本管理系统。这确保了多版本的处理能够准确地反映软件系统的实际状态。我本人非常欣赏这一机制的现实意义。在第一版 CADES 上，我们确实解决了一些问题，但同时也面临了其他挑战。例如，如果我们能够更好地了解这些版本的相互影响，那将有助于解决多版本问题。

软件工程系统

从实践的角度来看，软件工程系统已经成为软件开发的核心。这些系统不仅包括质量控制，还涵盖了软件设计和实施。例如，对于高性能、关键任务系统，软件工程系统必须能够保证软件的可靠性和可伸缩性。在软件开发过程中，需要对输入语言进行多次迭代，以确保软件的正确性。

未来展望

我认为，软件工程系统应该在未来的开发过程中保持重要性。在软件开发过程中，我们需要持续改进软件工程系统。例如，我们需要改进软件的可伸缩性，以满足高性能、关键任务系统的需求。

要求工程

在商业、工业或政府环境中，与高质量、复杂软件系统相关的开发和实施是关键。我们需要有一个明确和正式的软件要求定义，以及在其生命周期中的总体集成。在开发过程中，我们需要有适当的工具来支持这些需求。)

没有一个正式的方案来定义和接受产品需求。即使在某些情况下，产品的要求定义和其生命周期过程之间仍然存在缺乏统一的问题。在客户接收产品时，或者拒绝产品时，产品的需求定义和其生命周期过程之间存在缺乏统一的问题。
Graphical design language

I identified one of the “weaknesses” of CADES as being its gross approach to the control of Chinese-Army-type operations. The industry has moved on. Large teams become less and less the norm. So must software engineering evolve. Textual design languages are weak for describing the back-of-envelopes design activity. They demand completeness of expression. During the early stages of conceptual design completeness tends to be very low on the list of priorities, rightfully so. In this context, we need to be able to support the expressionism associated with the highly-dynamic, creative thought processes and decision making of our most experienced, creative designers. To do this, ISES provides the designer with a powerful abstract symbolic interface supported by an intelligent color raster graphics system. With this interface the designer can sketch out his design and architectural ideas in close to a random fashion whilst the system interprets these into more orthodox design representation, placing this into the global context of the current ISES data base.

Software metrics

The more one formalizes and captures the total software life-cycle process with systems such as ISES and CADES, the more able one is to develop the physics of large software systems, or Software Metrics. A large proportion of the software engineering community is striving to understand the nature of large-scale, complex software systems. We need to be able to define the parameters of these systems, the relationships governing these parameters, and how best to optimise them. The power of a system such as ISES in this context is that it controls and contains the total flow of information associated with a product’s life-cycle. Hence, there is a level of activity-monitoring architecture within ISES—the system monitors its own usage and is able to evaluate and refine in-built metric models as a result of this day-to-day usage. It is a powerful step towards combining the worlds of theoretical and practical software engineering.

Hardware CAD compatibility

In many product development situations the position of interfaces between the levels of hardware and software implementation is arbitrary to a large extent. The factors which dictate the divisions are tactical ones and include flexibility, maintainability and timing criteria. In hardware engineering this arbitrary nature is well illustrated in the division of a design between custom and silicon chips and printed circuit boards. Indeed, several manufacturers now adopt the technique, with the aid of advanced computer-aided design systems, of prototyping their products in PCB technology and then, once proven, reducing these PCB’s each to a single custom chip. This flexibility should also extend to the outer software-implemented levels of a product. In order to aid efficiency in critical areas a manufacturer may want to re-implement a software function, or series of functions, in a chosen hardware technology. Today, this is a highly complex, risk-laden task. The transfer into hardware usually stops at the firmware level. ISES, on the other hand, is being implemented as part of a total-technology CAD system. Eventually this system will be driven exclusively by technology-independent requirement and problem statement languages. The ultimate system will provide complete flexibility to the designer in terms of how his design is implemented, and freedom to move between technologies for a single piece of design. Thus this CAD system, which contains ISES, will support the fundamental concept of the Implementation Technology Independence of the holon design unit.

ACKNOWLEDGMENTS

I should like to express my gratitude to the entire CADES team, both past and present members, for making the last seven years such enjoyable and entertaining ones. I should also like to express my admiration of ICL management for having the courage to follow through with the CADES program during times when the benefits were less obvious than they are today.

Permission to publish extracts from a paper delivered to the Second International Symposium on Operating Systems in Paris in October, 1978, is acknowledged by the author.

REFERENCES
