Architecture for Software Construction by Unrelated Developers

W.M. Gentleman
National Research Council, Institute for Information Technology, Ottawa, Ontario, Canada.
phone (613) 993-9010, fax(613) 952-0074, e-mail Morven.Gentleman@IIT.NRC.CA

Abstract: Suppose one COTS (Commercial Off the Shelf) software supplier provides an interpreter for a problem oriented language, another provides an application generator for producing numerical solvers for a class of partial differential equations, and a third produces a visualization package. A team of domain specialists writes scripts in the problem oriented language to define cases to be solved, uses the application generator to produce an appropriate solver, solves the generated PDE, and uses the visualization package to analyze the results and adjust the description of cases.

Such examples illustrate that large and long lived software systems can result from the combined effort by various unrelated development organizations, organizations not even known to one another. No single design authority, to which the others report, has overall system responsibility.

Such examples also illustrate the importance for software architecture to include relationships between entities that exist and are used during the construction process, instead of focusing only on relationships between entities that exist at runtime.

The needs for software architecture for such systems are not well met by the existing literature.

Keywords: Software architecture, COTS, unrelated developers

INTRODUCTION

The literature on software architecture, for instance as surveyed in Shaw(1996), has largely focused on components in the sense of computational entities that exist at runtime, and their connections in terms of data and control transfer. Various styles of how the same computational system could be structured have been studied, considering how the structure could be analyzed, how individual components could be reused,

and how the structure could be reused in other situations. Alternatively, practitioners such as Whitney(1995), Tzerpos(1996), or Finnigan(1997), have sometimes used software architecture as a focus on definition and use relationships of entities that exist at runtime. These are, of course, interesting issues, but in many situations they are not the dominant reasons for the architectural structure adopted for the software system. Our definition of the term software architecture is that it is a high level description of a set of entities and their relationships, the understanding of which is essential to the understanding of the overall structure of the system. This is consistent with the definition other authors have used, although the entities we might consider, and especially the relationships we might consider, are broader than some other authors might take.

In some systems, physical considerations dominate the software architecture. In these systems, any single computer might run several software components, but software entities running on different computers are definitely considered distinct components. The software architecture thus reflects hardware architecture issues such as geographic locality, bandwidth, unique hardware resources, redundancy for reliability, replication for capacity, etc. It may also reflect organizational and administrative realities of the operators, such as what functionality is centralized, what functionality is replicated at each branch plant or even at each workstation, and what functionality is provided by computers belonging to the customers of the system operator, not those of the operator itself.

In this paper we consider situations where the software is implemented not by a single organization, but by a number of organizations, perhaps as a prime contractor with subcontractors, perhaps as collaborating peers with different competencies, or perhaps as suppliers and users of COTS (Commercial-Off-the-Shelf) software products (Dean, 1997; Vigder, 1997). The software development organizations contributing parts of the system may not even be known to one another. These development organizations may contribute parts of the system at very different levels of abstraction.

The situation where the development organizations are unrelated, interacting only as suppliers and customers of COTS software products, is particularly interesting, because it is so far from the traditional development model. No single design authority, to which the others report, has overall responsibility for the whole system, in that, by definition, each COTS supplier implements his product to his own specification and timetable determined by his perception as to the market demand, of which this application is typically an inconsequential portion. Detailed

specifications and source code for COTS products are rarely available, never mind possible to influence. More seriously, the maintenance and evolution of each COTS component is done to its supplier's agenda, and since obsolete versions usually become defunct, a long-lived system must adapt to the change. When a part evolves and must be reintegrated, the enhancement may not even be implemented by the supplier of the original part — indeed, sometimes a plug-compatible part of completely different design is substituted. Evolution of such systems typically results from the evolution of the different parts, although the introduction of new parts and changes in the relationship of parts can occur. The integrator who brings together all the parts must find a software architecture that can use the COTS products as they are, or as they might be in future. The integration role may be substantial, or it may be quite small, and may even be automated.

For systems of the kind considered here, there is often not simply a single runtime. Often components are run to produce entities, even source code, that will be used by other entities at a later runtime. It is thus important in the software architecture also to include relationships between entities that exist, or are used, during the construction process of other entities. Some of the contributions, for instance macro packages, may no longer be localized at runtime, although they may have been localized at some earlier stage in the build process. Potential attributes of a component generated during a run are often not determinable from specific instances generated during particular runs, but may be inferred from the generating subsystem and input it might be given. Tools used in the build process may be essential in establishing that constraints required at the runtime of the application itself are in fact satisfied. The build process itself thus must be part of the software architecture. The system architect plans how the parts are created and brought together. Fortuitously, box-and-arrow diagrams are traditionally used both for explaining the build process and for explaining the software architecture.

These issues will be illustrated by three thinly disguised examples of real systems, systems implemented in the past that continue to be used and to be evolved today.

A SIMPLE EXAMPLE

As a concrete example, consider a situation where one COTS software supplier provides an interpreter for a special purpose problem oriented

language, another provides an application generator for producing numerical solvers for a certain class of partial differential equations, and a third produces a visualization package. The application of the system might be, for instance, to analyze accidental fires. A team of domain specialists writes scripts in the problem oriented language to define cases to be solved in terms of geometry, fuels, atmospheric conditions, etc.; uses the application generator to produce an appropriate solver given the characteristics of a specific case; compiles the generated solver; solves the generated PDE for that case; uses the visualization package to analyze the results and adjust the description of cases; and then repeats the cycle. Because the solution of each individual case is a significant investment, and because investigation of an accident involves running many cases and similar cases may show up in future, successful results from each case would typically be stored in an object database, keyed by parameters that characterize the case in the potential search space.

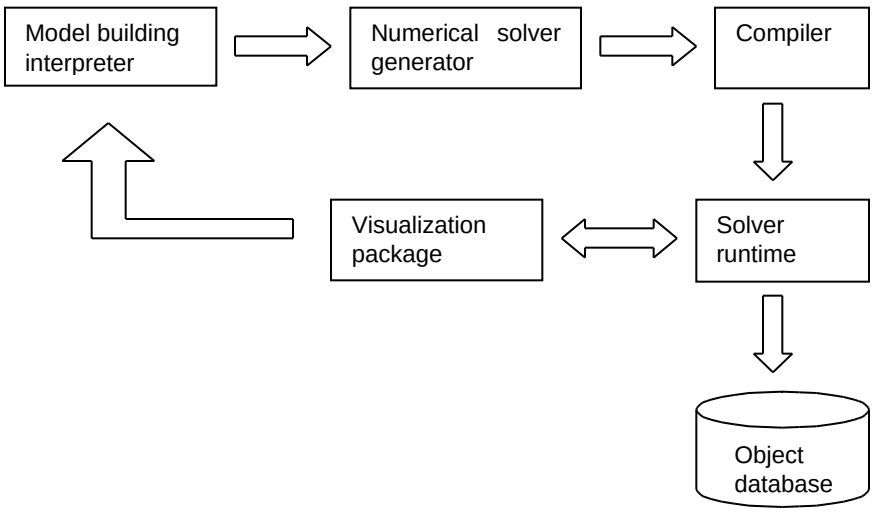

Figure 1. Superficial block-and-arrow diagram for example 1

At a sufficiently superficial level, the software architecture is simple and uninteresting: a cycle of subsystems, each producing data for its successor. A deeper level of software architecture elaborates on what connecting to the successor subsystem really entails, on how to exploit previous cases to reduce computational effort, and on how to recover from computational failures such as might result from going beyond the domain of applicability of the physical models or the numerical procedures. Because COTS components produce their output in whatever

representation and sequence that they do, and because this is unlikely to conform to the rigid representation and sequence required by the successor COTS component, insertion of, at least, a filter between them is normally required. In this example, as is often the case, more is needed. The COTS components will not work for every input with which they might be presented, and consequently the architecture must be extended to make provision for exceptions that might be raised. Moreover, a COTS component produces whatever output it produces, and some of this is not actually used by the immediate succeeding component in the notional cycle, but should be passed on through to subsequent components, in the same way that passes in the traditional compiler pipeline burned through intermediate language constructs not operated on until a later compiler pass. Unfortunately, COTS components are unlikely to make provision for simply passing through input that they do not intend to process, so the glue components must facilitate such data bypassing the COTS component. Thus the glue components are normally more general than simple filters.

In this example, the connectors between the components in the superficial view of the architecture are wildly different. At some level this is sufficient, because it shows where dominant relationships exist — and do not exist. At a deeper level we need to understand what they are. The output of the first component, the model builder, is three different kinds: mathematical formulae which are the partial differential equations and also the description in space of the region of integration; mathematical facts which have been proved or are to be assumed about these formulae; and large numerical arrays that represent initial values, boundary conditions, and other parameter values. Only the mathematical facts and the mathematical formulae, together with a few of the parameter values, are required by the second component, the application generator. This application generator uses these facts and formulae to select among various choices of algorithms and data structures to produce source code for a numerical solver optimized to the particular kind of problem to be solved and the kind of computational resources available to solve it. The first connector thus filters a data stream, possibly reordering typed items and changing their representation. The second connector is a very simple pipe, taking the source code produced by the application system and feeding it to a compilation system. The third connector is more complicated, for it must run the executable image produced by the compilation system, and make available to it the numerical arrays and parameters produced by the model builder. Classically, the fourth connector could be very simple, for numerical solvers wrote their results to files which were later subject to

analysis by techniques such as visualization. The visualization might also have required the full output from the model builder. Today, however, visualization is often used interactively to steer the computation as it proceeds, so in addition to inspection of stored partial or complete results, this connector must support debugger-like actions. The final connector, from the visualization package back to the model builder, is simply revising the scripts that define the problems, and is probably accomplished by a standard editor.

USES FOR A SYSTEM ARCHITECTURAL DESCRIPTION

By looking at how we might use the architectural description of a system, we can learn more about what it might contain, and how it would be usefully represented. Who is the high level description for?

1.1 High level description during planning stages

From the literature, one might conclude that the principal use of an architectural description of a system was as a high level planning document, to agree upon what must be done and what it would be nice to do, then to derive specifications for the components to be implemented. Such a top down approach can be effective where all the components have to be designed, or even when some of them pre-exist and either the others must be designed to accommodate them or glue must be specified. It can be used to establish properties such as completeness and correctness, and to analyze for properties such as capacity and concurrency. It can be studied for examining dependencies of partial results, and hence for identifying opportunities for phasing computation and so reducing instantaneous demand for memory and other resources. It can be used to study communication requirements between components and hence to assess suitability for distribution in the sense of what should run on which node of a network. If the software system was to be operated jointly by a collection of organizations, the software architecture might be used to study distribution, in the sense of suggesting which components and which responsibilities be given to which organizational units. If the system is to be sold as a commercial product to many different customers, the software architecture might suggest packaging for optional configurations. The software architecture can also serve as a documentation framework,

identifying where to record assumptions and dependencies between components.

For software to be implemented jointly by a collection of organizations, a software architecture can provide a framework for considering a number of acquisition and implementation questions which are nontechnical but with potentially technical consequences. What constraints are implied by available components that could be used? Where would separate suppliers of components possibly be effective in reducing cost or improving time to completion? Where does intimate dependency on the same technology imply that the same subcontractor should be used to avoid duplication of startup effort or to avoid errors due to conflicting interpretations? How should implementation responsibilities be divided to correspond to the competencies of different collaborators? And for systems where corporate or national security is an issue, what are the security clearance implications for the implementers of different components?

1.2 High level description during operation

During operation of the system, the primary use of a system architectural description is tutorial. Because integration of components is often not seamless, the operators of the system often need to be aware of the roles of different components in the production system, and the software architecture often is a useful framework for teaching them. For example, systems often are designed with metering for monitoring and tuning purposes. The significance of such measurements depends on the system architecture, and hence the operator needs to understand the system architecture in order to properly interpret the measurements and act on them. As another example, operational problems often arise in the production use of systems not because of bugs in the implementation, but because intrinsic limitations in the underlying science restrict the domain of applicability, or because choices made during implementation in the absence of knowledge turn out not to be consistent with operational experience. When such problems arise, the operator needs to understand the architecture well enough to recognize the situation and the source of the problem, to take corrective action, and to plan workarounds. Also, as mentioned earlier, the software architecture can be useful for establishing operational responsibilities for different organizational units. Note that operators like this rarely have programming skills.

1.3 High level description during maintenance

Day-to-day maintenance is normally finding and fixing minor bugs, mis-configurations, and interoperability conflicts. Minor enhancements may also be included. For systems that operate nonstop for extended periods, simply monitoring for outages and interpreting logs is often difficult, and the maintainers not only need to understand the software architecture generally but may need to make detailed reference to it in order to localize and eventually isolate errors. Often attempting the repair immediately is not possible, so through knowledge of the software architecture a workaround must be found. Organizing for and actually conducting the repair requires detailed just-in-time learning of the code at the site of the error, as well as at other affected sites. An understanding of the software architecture of the system is key to knowing what to study and the context in which it must be understood. Unfortunately, the skill level of staff employed for this kind of maintenance is often less than that of the initial developers or developers involved in major enhancements.

1.4 High level description during major evolution

Major evolution of an existing system has much in common with initial implementation, except that because it is incremental there is more incentive to maximize reuse of components from the previous release, as well as to ensure interoperability with data, including control data, produced by or for the previous release. Working out a strategy for actually carrying out the upgrade or replacement of a component is particularly important, especially in nonstop systems. Planning as to how to add a new component or to make other architectural changes is important, and requires a solid understanding of the existing software architecture. That understanding can lead to identification of required competencies and appropriate allocation of responsibilities to carry out the change. However, such changes are relatively rare. The dominant kind of change, especially for a successful architecture, is change by upgrade of a single component.

A SECOND EXAMPLE

Another example where the software architecture is dominated by pre-existing components, although not in this case COTS software, is a training system for operators of an embedded system, such as a weapons fire

control system, a SCADA (sensor control and data acquisition) system, or a command and control system for air traffic control. For such systems, it is often essential that new operators be trained on the real system, warts and all. Only that way will the new operators get an appropriate sense of the real system's capabilities and limitations, and get the feel of its responsiveness in real time. Consequently, the core component of such a training system is an instance of the real embedded system. There are three other subsystems in the training system. One other component is a debriefing subsystem. This is a subsystem that is able to record the student's actions, in real time, as the system responds to interesting situations, so that an instructor can go back through the situations with the student to point out where the student has done well, where the student has used bad judgement or made errors, and what the consequences of these have been. Because real time is an essential aspect of such situations, it is necessary not just to rely on probes into the real system to log the displays produced by the system together with the student's responses to them. It is also necessary to log video and audio of the student's off-line activity, especially where there are several operators working together simultaneously with the system. Many parts of this subsystem pre-exist. Another important subsystem is the world modeller. The real embedded system interacts with the real world through various sensors and actuators, and since use of the real sensors and actuators may be impractical for training purposes, they must be carefully simulated. The real sensors and actuators are not independent of each other, but are coupled at least through the real world, so the simulated world for the training system must properly model such interactions. Adequate simulation of the real world requires sufficiently precise modelling of the physical situation, with adequate computational power and typically with a great deal of empirically determined data. It also requires an understanding of what approximations and shortcuts can be taken to meet real time performance without losing simulation fidelity. Such a simulated world may be a valuable asset that must also be used with trainers for other embedded systems. Of course to carry out the pedagogical purpose of the training system, the world simulator has to be directed to produce scenarios illustrating situations that the students are to be taught to deal with. Thus the last subsystem is a scenario editor for the simulated world. Obviously scenario development happens at a different runtime than the students lesson. A final wrinkle in such training systems is that qualified instructors are usually in short supply, so the whole training system is partially replicated to allow several students to be trained simultaneously.

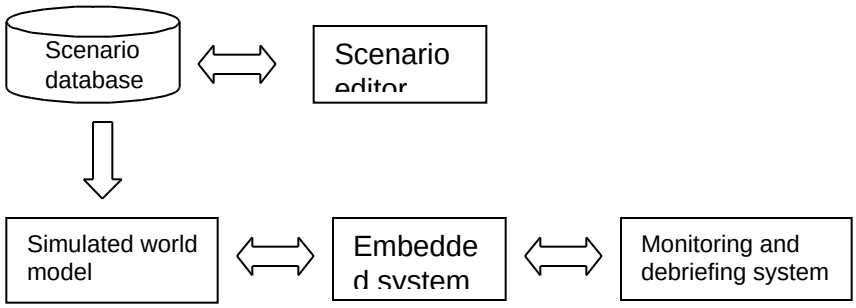

Figure 2. Superficial block-and-arrow diagram for example 2

This example is interesting because qualifications for implementing each of the four subsystems are quite different. The real embedded system was implemented, and is frequently upgraded, by whoever, typically a systems contractor expert in the sensors and actuators and signal processing. The debriefing system is best implemented by a company well versed in pedagogical techniques, so that it will be easy to capture and to replay appropriate aspects of the student's actions. The simulated world subsystem is best done by a company with strong scientific computing credentials in the appropriate science. The scenario editing subsystem is best implemented by a company that combines usability skills with a clear understanding of what scenarios will be needed.

The example is also interesting because at a superficial level, understanding the relationships between the four subsystems is simple. Any attempt to provide a complete and correct description of all the interactions becomes mired in detail.

A THIRD EXAMPLE

In next example, the COTS software merely provide a platform on which the system is built rather than performing substantial parts of the computation itself, but limitations of the COTS components are the major cause of architectural choices, with anticipated implementation churn during evolution of these components also playing a role. The system itself is a small but long-lived interactive exhibit for displaying to the public current information about air quality. Two different kinds of information

are presented in the exhibit. The first is descriptive material which is generally static but changes occasionally, for instance when administrative or legal actions affect what is being described. The second kind of information displayed is trends in recent measured data from a network of online monitoring stations. The core of the exhibit is a program written in the proprietary language of a commercial authoring system. This provides facilities from user dialogs to visual effects, and allows the exhibit designer to focus on effective communication with users instead of on implementation. Unfortunately, the authoring system has functionality deficiencies. The first is that it cannot generate and display the multicolour time series graphs required to display trends. This is solved by a plug-in available from a third-party supplier, together with some glue to remap data structures. The second deficiency is more serious: the network of monitoring stations must be polled by dial-up modem and the measurements accumulated to be shared by several instances of the exhibit, but the language of the authoring system, even with plug-ins, is too weak to support the error handling or concurrency control to do this. The solution is

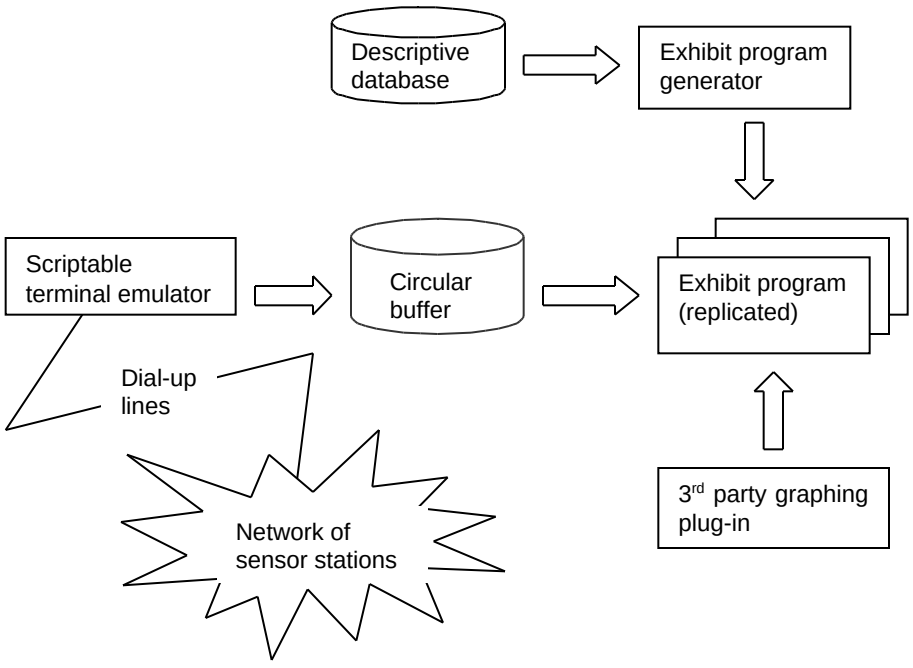

Figure 3. Superficial block-and-arrow diagram for example 3

that the exhibit uses read-only optimistic concurrency control to read from a shared database (conceptually a circular buffer of records) maintained by another program. The program maintaining the database is written in another proprietary language, this one being the communications control language of a terminal emulator. The problem of being able to keep the descriptive material up to date without manually updating the whole exhibit each time some fact changes is addressed by keeping all the relevant descriptive material in a database, and using scripts in the database language to walk the database and generate the pages for the authoring system whenever a change is needed. Since the descriptive material includes multimedia items such as pictures, sound and video, an appropriate commercial product is used.

The principal use for the system architectural description here is to explain to the front line nontechnical maintenance staff what actions to take when needed. Regeneration of the pages of descriptive material works well as long as maintenance personnel understand they need to update the database, and do not attempt to change the pages directly. Reorganizing the pages calls for different skills, but happens rarely. The monitoring stations have been a continuing source of operational problems: changed passwords block access, station identification is arbitrarily changed, modems go offline for periods stretching into months, stations are shut down and new ones are opened, data format is changed, manual editing of data at the monitoring stations produces records out of sequence, etc. Since the monitoring stations are operated by a different government agency, changes occur without notification and they are not responsive to requests for explanation, much less remediation. Accommodating such situations frequently requires manual intervention, but bullet-proofing the system, so that it reports on detected problems and continues to operate, is mandatory and has architectural implications. The most troublesome problems however have been upgrades to the platforms: the hardware on which the exhibit runs, the operating system on that hardware, and the versions of the various COTS components. These are typically upgraded without notice, and not infrequently the newest versions no longer interoperate successfully. The conflicts are usually easy to resolve, but require technical support. Since technical support is hundreds of miles away and on a time and materials basis, front line support must have a sufficient understanding of the software architecture to localize the problem, perform simple corrective procedures such as reinstalling components, and report symptoms.

CONCLUSIONS

Systems with characteristics similar to the examples cited are being developed all the time. The prime purposes of the architecture descriptions of such systems have been for communication with, and analysis by, other people — automated analysis has not been a priority. Architectural styles are not a central issue. For communicating with people, excessive formalism is not necessarily more effective, and text-only descriptions have also proved to have shortcomings. While not entirely satisfactory, the use of block-and-arrow diagrams, supplemented by text, has proved sufficient for the uses cited. What shortcomings have been apparent relate to having consistent presentations of the software architecture at various depths and from different points of view. Too much detail irrelevant to one's current interest is obfuscating.

Perhaps the flaw lies in thinking of the system architectural description as a single document, manually composed, and viewed in its entirety. Instead, we could think of a set of reports generated from a common database (Finnigan,1997), in the way some re-engineering tools present facts gleaned from existing source code. The central focus would be the cognitive psychology focus of how to make the presentation comprehensible, rather than the computer science focus of how to make the basis general and precise.

In practice, the decomposition into CSCI (Computer Software Configuration Items) for projects constructed under 2167a, and indeed the description of the individual CSCI themselves, often reflected more the competencies of, and relationships between, the prime and the various subcontractors than it did functionality, data access, or allocation of software to hardware. Perhaps this was not so wrong!

REFERENCES

Dean, J.C. and Vigder, M.R. (1997) System Implementation Using Off-the-shelf Software, *Proceedings of the 9th Annual Software Technology Conference*. Department of Defense, Salt Lake City, Utah, 27 April - 2 May 1997.

Finnigan, P. J., Holt, R. C., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H. A., Myloupoulos, J., Perelgut, S. G., Stanley, M., and Wong, K. (1997) The Software Bookshelf, *IBM Systems Journal*, **Vol. 30, No. 4**, 564-593.

Shaw, M. and Garlan, D. (1996) Software Architecture. Prentice Hall, Upper Saddle River, NJ.

Tzerpos, V. and Holt, R.C. (1996) A Hybrid Process for Recovering Software Architecture, *Proceedings of CASCON '96*, Toronto, ON, 12-14 November 1996, 1-6.

Vigder, M.R. and Dean, J.C. (1997) An Architectural Approach to Building Systems fromCOTS Components, *Proceedings of the 22nd Annual Software Engineering Workshop*. National Aeronautics and Space Administration - Goddard Space Flight Center, Greenbelt, Maryland, 3-4 December 1997

Whitney, M., Kontogiannis, K., Johnson, H. J., Bernstein, M., Corrie, B., Merlo, E., McDaniel, J., De Mori, R., Müller, H. A., Myloupoulos, J., Stanley, M., Tilley, S., and Wong, K. (1995) Using an Integrated Toolset for Program Understanding, *Proceedings of CASCON '95*, Toronto, ON, 7-9 November 1995, 262-274.

BIOGRAPHY

Dr. W. Morven Gentleman, interim Director General of the Institute for Information Technology, is also Head of the Software Engineering Laboratory of the National Research Council of Canada.

Before NRC, he was professor of Computer Science and of Statistics at the University of Waterloo, and before that was at Bell Laboratories in Murray Hill. He has also spent sabbaticals at the National Physical Laboratory in the UK.

He has published in many areas of computer science. He has been responsible for delivering commercial products, as well as being involved in building large systems.

Dr. Gentleman holds a Ph.D. in Mathematics from Princeton University.