



# INTERFACE

NEWSLETTER OF THE SOFTWARE ENGINEERING PROCESS GROUP

VOLUME 6, NUMBER 1 • FEBRUARY 1997

## CHIEF SCIENTIST PASSES TORCH TO ART PYSTER

Floyd Hollister's detail as Chief Scientist for Software Engineering has ended with the selection of Dr. Arthur Pyster as head of the software engineering improvement team (SEIT), AIT-5. Dr. Pyster will continue ongoing initiatives while bringing a new perspective to improving the FAA's software life cycle management practices and processes. He will report to Theron Gray, Chief Information Officer and Director of the Office of Information Technology at the Agency. Dr. Hollister will remain as a consultant to Dr. Pyster for the next month or so, helping to ease the transition.

Dr. Pyster brings over 20 years experience in software technology to the FAA. Most recently, he was vice president, chief technical officer, and chief technologist at the Software Productivity Consortium (SPC) in Herndon. He was involved in the oversight and direction of numerous programs for the member companies.



He was chief technical officer on a large, successful DARPA funded project.

As chairman of the steering group that manages the enterprise process improvement collaboration since 1994, he directed the initiation of the Systems Engineering Capability Maturity Model (SE-CMM) and the Integrated Product Development CMM.

*continued on page 7*

## MIKE DEWALT SPEAKS AT HEADQUARTERS

How much would you bet on a poker hand if you couldn't see your cards or any of the other players? Not much, especially if your safety depended on the outcome. But what if you could see two cards? What if you could see four? This is the analogy Mike DeWalt used to illustrate how increasing information about a software product, particularly about how it was developed, leads to greater assurance for its use in safety-critical systems. For airborne systems, the FAA uses a standard designed especially for avionics: ***RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification.***

Mike is the FAA's National Resource Specialist for Software from Seattle, and he was in Washington in December to present a day-long seminar to the FAA on the DO-178B standard, published in December 1992. If there is anyone who can make a dry subject interesting and

vital, it is Mike. Besides his enthusiasm and humor, his briefing has some whiz-bang graphics that are really worth seeing.

As Loni Czekalski said in the introduction, software has become a leading technology and there are more and more applications where it must work correctly and safely. What assurance does the government have that the software it acquires does this? The DO-178B standard is one model designed to assess a level of assurance for software. It lets us see some of the cards in our poker hand—as many as we are willing to pay for.

DO-178B emphasizes process, but does not neglect product attributes. Following it will provide a probability of assurance that success criteria for safe software have been met.

The main points to consider regarding DO-178B, software errors, and safety, is that:

- The standard attempts to assure that whole classes of errors are unlikely.

*continued on page 2*

## IMPROVING ACQUISITION OF SOFTWARE INTENSIVE SYSTEMS

Dr. Linda Ibrahim, AIT-5  
Chair, Corporate SEPG

The FAA Corporate Software Engineering Process Group (SEPG) is responsible for improving the processes used for the acquisition of software intensive systems. These include all processes in the FAA's new Acquisition Management System (AMS), from Mission Analysis to Service-Life Extension, which pertain to software intensive systems. They may be used by customers, suppliers, managers, engineers, and acquisition specialists.

There are several reference models which can be used to provide guidance, but three models are particularly relevant to the SEPG's purpose: the Capability Maturity Model for Software (SW-CMM), the Software Acquisition Capability Maturity Model (SA-CMM), and the Systems Engineering Capability Maturity Model (SE-CMM). These models cover the different aspects of the AMS and span the key disciplines involved.

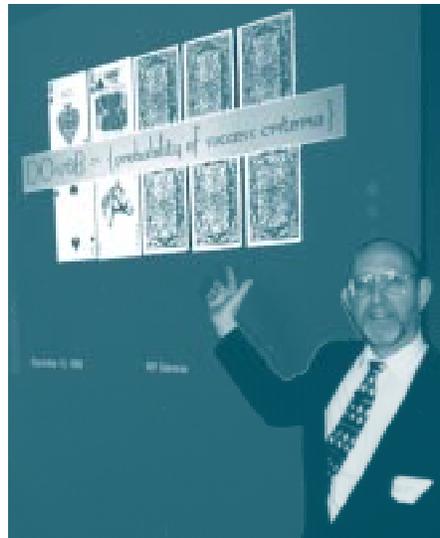
We have devised an initial combined CMM model to guide our improvement efforts. This model merges the 52 process areas of the three CMMs to derive a streamlined set of 23 combined process areas as candidates for improvement. It considers the different architectures of the models and maps processes to their occurrence in each phase of the AMS life cycle.

Over the coming months we will be working to refine and enhance this integrated "FAA-CMM." We anticipate realizing several advantages from taking this integrated approach including:

- better coordination of FAA process improvement activities by ensuring processes are defined and improved in an integrated way involving all relevant disciplines
- reduction of the number of processes and practices to be improved
- the ability to follow one unified approach, yet measure improvement against all of the models. ■

*"Mike DeWalt Speaks at Headquarters"*  
*continued from page 1*

- Safety properties are captured in the requirements and then assured by applying the objectives.
- DO-178B objectives in and of themselves are not safety sensitive.
- A best standard would ensure software safety within the target system, but we normally compromise by requiring the use of processes that reduce errors, and by including safety as part of requirements.
- Proper application of DO-178B requires knowledgeable people well versed in sound software engineering principles and digital hardware, and with extensive and varied software project experience.



For safety purposes, the regulations and advisory materials define five levels of criticality from Catastrophic (aircraft destroyed with many fatalities) to No effect (no effect on the operation of the aircraft). DO-178B defines five corresponding levels of assurance rigor from A (for catastrophic) to E (for no effect). With decreasing levels of assurance, fewer of the DO-178B criteria need to be satisfied.

Can DO-178B be used for COTS systems? **No.** It is not a standard for procurement. Without a window into the system there is no way to match COTS products to safety levels and verify the objectives. With a commercial product there is no verification of the vendors processes—no cards are shown.

Mike went on to compare and

contrast DO-178B, the Software Engineering Institute's Capability Maturity Model, the International Standard's Organization standard 9000, and other "standards." Of particular interest, despite some anecdotal reports of successful use (from obviously "successful" users), none of these standards have any scientific basis, nor is any in prospect. He particularly stressed that the standards imposed need to be matched to needs, there is a constant danger of over and underspecification, no other standards have been shown to provide the reduction in product uncertainty that DO-178B provides at delivery, and that there are, in general, no good or bad standards—only "different" standards.

Finally, designated safety team members are needed to establish the hazard categories of the system failure conditions. Software knowledgeable systems engineers must establish the contribution of software to a failure condition. The hazard category provides a requirement for a random probability of failure and design assurance levels (rigor), i.e., threshold probabilities, for software and hardware. Determination of assurance levels are left to the last. No claims for reliability measures or safety can be based on software level. Arbitrary decisions on software level may be needed (because of insufficient knowledge). The selection of appropriate assurance methods are critical—different assurance methods involve different risk exposures and tradeoffs. ■

## inter FACE

is published quarterly by SEPG

DOT/FAA/AIT-5  
800 Independence Avenue, SW  
Washington, DC 20591

▼  
Chief Scientist for Software Engineering

**Arthur Pyster (202) 267-8020**

▼  
Editor

**Norman Simenson (202) 267-7431**

▼  
FAX (202) 267-5080

## ARIANE 5 - WHY DUPLICATED SOFTWARE IS NOT REDUNDANT

[*editor's note:* the following has been severely abbreviated from the official ESA report as released to the press. A more complete version, plus comments, appears on the **FAA SEPG Interface** Web site. Bolding has been added to emphasize key findings.]

—On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. Following is a summary of some of the findings of the independent Inquiry Board.

The design of the Ariane 5 Inertial Reference System (SRI) is practically the same as that used on Ariane 4, particularly as regards the software, which has been reused largely unmodified.

**Failure Scenario:** The physical failure was initiated as the result of extreme rocket motor deflections commanded by the On-Board Computer (OBC) software on the basis of data transmitted by the active Inertial Reference System (SRI 2). Part of these data at that time **did not contain proper flight data, but showed an error code from SRI 2, which was interpreted as flight data.** [*editor's note:* this shows a failure of proper containment. High integrity software is supposed to be designed to isolate and contain errors, not exacerbate and propagate them. The consequences reveal a fatal flaw in the design of the SRI or OBC software, or both.]

The reason why the active SRI 2 did not send correct attitude data was that the unit had declared a failure due to a software exception. **The OBC could not switch to the back-up SRI 1 because that unit had already ceased to function for the same reason as SRI 2.** [*editor's note:* so much for using duplicate software as part of a backup.]

The internal SRI software exception was caused by an overflow during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. This resulted in an Operand Error.

[*editor's note:* down converting from a floating point number to a signed integer should *always* raise a red warning flag—it is an extraordinarily risky thing to do. Putting an error code on a data bus should have raised another red flag.]

The error occurred in a part of the software that **is based on a requirement of Ariane 4 and is not required for Ariane 5.** The error occurred due to an unexpectedly high value because **the early part of the trajectory of Ariane 5 differs from that of Ariane 4.**

**Comments on the Failure Scenario:** Not all the conversions were protected because a maximum workload target of 80% had been set for the SRI computer [*editor's note:* sounds like a problem with inadequate computer performance was being “worked around.”] Analysis indicated that the unprotected conversions were either physically limited or that there was a large margin of safety, analysis which turned out to be faulty. It is important to note that **the decision to protect certain variables but not others was taken jointly by project partners at several contractual levels.**

There is no evidence that any trajectory data were used to analyze the behavior of the unprotected variables, and it is even more important to note that **it was jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements and specification.** [*editor's note:* clearly, the consequences of an error in the evaluated variables were **also NOT** investigated. No doubt these decisions saved some minor costs.]

Although the failure was due to a systematic software design error, mechanisms can be introduced to mitigate this type of problem. **There is reason for concern that a software exception should be allowed, or even required, to cause a processor to halt while handling mission-critical equipment.** [*editor's note:* this violates a fundamental precept of high integrity system design, namely, that systems should “fail gracefully” — not catastrophically.] Indeed, the loss of a proper software function is hazardous in such a design because the same software runs in both SRI units. In the case of Ariane 501, this resulted in the switch-off of two still healthy critical units of equipment.

An underlying theme in the development of Ariane 5 is the bias towards the mitigation of random failure. [*editor's note:* see *Heisenbugs in the System* in the November, 1996 issue of *Interface*. The

probability that two identical units will fail at the same time due to random error is very small, but the probability that the two units will both fail due to the same systematic error is near one.] The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault. **The Board has reason to believe that this view is also accepted in other areas of Ariane 5 software design. The Board is in favor of the opposite view, that software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct.**

This means that critical software—in the sense that failure of the software puts the mission at risk—must be identified at a very detailed level, **that exceptional behavior must be confined, and that a reasonable back-up policy must take software failures into account.**

**Testing:** SRI testing at equipment level was conducted rigorously, beyond what was expected for Ariane 5 with regard to all environmental factors. However, no test was performed to verify that the SRI would behave correctly when being subjected to the count-down and flight time sequence and the trajectory of Ariane 5.

A large number of closed-loop simulations of the complete flight, simulating ground segment operation, telemetry flow and launcher dynamics were run. Many equipment items were physically present and exercised **but not the two SRIs**, which were simulated by specially developed software modules.

The decision NOT to include the SRIs as part of the closed-loop simulation tests was the result of a determination that the simulation needed to adequately stimulate the gyros and accelerometers of the SRI to the necessary accuracy would prove very expensive, the basic design and software of the SRI had proved itself in the Ariane 4 and previous rockets, and the unit tests of the SRI had been very extensive (but, as noted, without the Ariane 5 trajectory data). [*editor's note:* these are essentially the same arguments made for not performing system test of the mirror in the Hubble Space Telescope.] **Page 3**



©1996 ESA. All rights reserved.

## SOFTWARE TESTING OF SAFETY CRITICAL SYSTEMS – PART 2

by Patrick Brown, MITRE

What caused the Challenger accident? A failure of an O-ring in one rocket booster, right? Few of us remember what the Rogers Commission report said, but you may recall that the investigation went far beyond O-rings. In the end, the rings were redesigned and thoroughly retested. But sweeping changes resulting from the investigation rippled throughout NASA, affecting training, quality assurance, and even headquarters management. The importance of safety as the primary concern was brought home to every operation within the Shuttle program.

This example illustrates an essential point about the safety of complex systems. Safety is a *system* property, and systems include not only the hardware, software and procedures, but many other factors such as the operators, maintenance, training, management, and even corporate policies and attitudes. This article emphasizes processes for the development of software for safety-critical systems, especially testing. The safety of the software, however, should be understood in the context of total system safety. There is no such thing as “safe” or “unsafe” software, absent a system context. (Software for nuclear reactor control cannot cause a core meltdown no matter how defective, if it is running in a simulator.) Moreover, software may result in an unsafe system merely because it provides confusing information to an operator.

Evaluation of system safety begins with a system safety assessment or hazard analysis. Any state of the system which can cause an accident resulting in injury, loss of life, or economic loss is identified as a hazard. Each hazard is assigned potential accident costs and probabilities, enabling hazards to be prioritized. A hazard may be an airplane entering another airplane’s airspace, or the doors opening on the wrong side of a Metro train. A hazard does not inevitably lead to an accident, but the safety goal is to eliminate, control, or alert operating personnel to all identified hazards.

Hazard analysis traces hazardous states back to prior states in the paths

which can precipitate the hazard. This analysis may lead to design features which, suitably modified, will block the paths to the hazardous state. Hazard analysis is not a one-time event, but a continuous process starting with the system requirements and progressing through system specification, design, implementation, test, and maintenance.

Software can be categorized with respect to the level of safety criticality for a given set of operations. For airborne software, the FAA uses five categories ranging from Level A (catastrophic failure condition) to Level E (no effect on aircraft operations). These are described in *RTCA Standard DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. Software testing and approval processes are increasingly more stringent and thorough for the higher safety levels.

However, software is never certified as safe—it is merely approved for use in safety-critical systems. The approval is based on a number of criteria which vastly increase the probability that it will operate as specified and that the specifications appear to be hazard free. Indeed, no system can be certified as safe—that would require prescience on the part of the certifier. At most, systems can be certified as made safe against all known and anticipated hazards.

Testing of software systems, although necessary, is no sufficient guarantee of safety. Bugs will always remain, triggered—perhaps years after a system has gone into operations—by an unusual sequence of events, by changes to the environment, or by changes to other parts of the system. Requirements often contain mistakes, generally of omission, so that even rigorous testing against a specification cannot always reveal a problem.

At least one loss of life (and multiple injury) incident and the recent destruction of the Ariane 5 rocket can be blamed on the reuse of old software with a long history of safe use in a new, upgraded system. Because of the long previous history of safe use for the software in each case, it was inadequately tested in the new system—with very unfortunate results.

Once identification and classification of hazards have occurred, various measures can be taken in the software to improve system safety. Software design

should so partition components that safety-critical elements are isolated from the remaining elements. Strict partitioning may make it practical to apply formal methods. Safety-critical components can be subjected to especially intensive inspections during development and to more intensive testing. Designs can incorporate interlocks and other safeguards, such as safety monitors to ensure that a module’s inputs, outputs, and state parameters remain within specified bounds.

There have been three general approaches to dealing with software errors to ensure safety. First, we need to get the requirements and code absolutely correct. Structured programming and testing, formal methods, and “clean room” are techniques to do this. For many reasons, such as missed and/or constantly evolving requirements, and increasing system complexity, this has proved to be extraordinarily difficult.

Second, we can make the software fault tolerant through redundancy and auto-diagnostics. This approach offers a safeguard against both system faults and software errors, but has not worked as well as expected. For one thing, fault-tolerant software is notoriously poor at detecting errors in the fault-tolerant software itself—all designers and programmers tend to make mistakes. And this will not protect against faulty assumptions on the part of users, analysts, designers, or programmers, or against faulty algorithms or requirements. These two approaches may increase software reliability, but not necessarily its safety. Reliable software can behave in reliably unexpected and unsafe ways.

Despite the limitations of the first

*continued on next page*



“Software Testing...” continued from previous page

two approaches, systems can be made safe. The third approach to software safety is specifically to prevent or detect software states that can result in hazardous system states. Design modifications that block paths to a hazardous state can be built into the system. Such blocks are normally called interlocks. In modern systems, hardware interlocks are increasingly being replaced by “software interlocks.” Unfortunately, the design and implementation of proper “software interlocks” requires very specialized training.

It is in this context that software is called safety-critical, since failure of a “software interlock” can result in the occurrence of a hazardous state in the parent system. Safety-specific testing must be planned and integrated into the normal testing process. Not only the operational requirements must be tested, but also the safety related requirements—up to destructive testing, if necessary.

Beyond the software, the system design should incorporate judicious use of hardware interlocks and human

alerting and override capabilities. This holistic approach is what Nancy Leveson means by the name of her book: *Safeware: System Safety and Computers*.

Software safety requires the use of special techniques and disciplined processes that go well beyond the standard software engineering practices used for obtaining quality software. Safety must permeate the entire system life cycle. It must be reflected in all operational and maintenance procedures and documentation. This may best be ensured by fielding an independent system safety subteam at the inception of every IPT which has products which require it.

DO-178B was developed as a standard for avionics software. Historically, ground systems (other than ground-to-air voice systems) have been categorized as “controller aids,” and not critical for the execution of the primary mission, air traffic separation. But all systems have become increasingly interdependent and all systems have become increasingly dependent upon software. As a consequence, the application of DO-178B, or a successor, will have to be expanded.

The Voice Switching and Communications System (VSCS) is a critical component of the ground-to-air voice system. It is estimated that the embedded FAA application specific software runs to 1.5 million source lines of code. The operating system and other COTS software adds as much as 13.5 million source lines of code.

The Plan View Display (PVD) situation displays used by the controllers are generated by software. While total loss of display may be tolerable (controllers are trained to be able to “visualize” their airspace and the relative position of aircraft within it), defective software can cause incorrect information to be displayed. This may cause controllers to incorrectly visualize their airspace and allow hazardous states to occur.

The Wide Area Augmentation System involves elements located on the ground, in orbit, and on board aircraft. How should such a “mixed” system be approved for operational use? Ground and airborne system functions can no longer be kept safely separated as computers increasingly interoperate and exchange data. Development and testing for safety-related software will have to arrive at a common set of standards for ground and airborne applications to ensure the integrity of the entire system. ■



## Letter from the EDITOR

## SOFTWARE INTERLOCKS ARE...NOT!

Careless and improper use of language has been responsible for more waste of resources, including death and destruction, than almost any other source. Until prohibited by law, gasoline tanks labeled as “empty” (but filled with highly explosive gasoline vapor) were constantly being exploded by otherwise scrupulously careful smokers. After all, we all **know** that something that is “empty” cannot hurt us. Don’t we? The deadliness of that equation is as recent as the ill-fated Valuejet airliner incident, where obsolete oxygen canisters were mislabeled as “empty.”

Elsewhere in this issue, we have made the case that software cannot be the **direct** cause of any accident. Software is analogous to thought, and if the thought were the same as the act, we would all be in dire circumstances! But if we define an interlock as a physical device for preventing or blocking some physical state from being entered, then the “software interlock” can be as treacherous as the “empty” gasoline tank or the “empty” oxygen canister. Issuing instructions to bar the gate is not

the same thing as barring the gate!

Most people in the software safety business are familiar with the Therak-25, a therapeutic x-ray machine which killed or seriously injured 6 people before the problem was discovered, correctly diagnosed, and fixed. There was a bug in the software. (Well, we expected that, didn’t we?) The software that had the bug was directly ported, without change, from the Therak-20. But the Therak-20 had hardware interlocks which the Therak-25 replaced with “software interlocks.” In the Therak-20, the software bug occasionally manifested itself, but the effects were so minor (because of the hardware interlocks) that no one even realized there was a problem. The fact that “software interlocks” are anything but interlocks resulted in the tragedy. A little problem of language.

Which is not to say that software cannot be used to drive an interlock. In this day and age, such a prohibition would be foolishly self defeating. But **both** the hardware **and** the software used in an

interlock must be wholly independent of the operational system, except for the physical interface which occurs when the interlock is preventing the operational system from entering an illegal state. This includes the processor used to run the software and any input-output hardware. (Even inputs should not be shared with the operational system, except where absolutely necessary and then under very carefully constrained conditions.) Nor do we rule out a plethora of software checks embedded in the operational system and intended to block any operational path to an illegal state. But the latter must NOT be called “interlocks.” Any other less semantically loaded term, such as “software safeguard,” may be used.

*Norm*

## CAN COTS SOFTWARE BE TRUSTED?

by John Liddiard, IPL, Bath, UK

As high integrity and safety related systems become increasingly sophisticated, it can be very hard to justify building standard functionality from scratch. There are great pressures for increasing the use of Commercial Off-the-Shelf (COTS) software, such as operating systems, graphical user/windowing interface systems (GUIs), communications utilities, databases, software development tools and, most recently, commercial datasets.

COTS software is packaged as ready to use, so there is an understandable tendency to assume that it is reliable, easy to use, and well behaved. But few software product companies work to the same high development standards as those which specialize in high integrity operational systems. It is simply not cost-effective for the average product. COTS products are very rarely certified as high integrity or safe for a given application.

Moreover, the suppliers of software products target for as wide a market as possible. Individual applications generally use only a fraction of the full functionality provided. As far as a given application is concerned, the unused product functions just add unnecessary (and risky) complexity.

There is an exquisite relationship between the money expended on testing and the tolerance of the average commercial user for bugs encountered. The objective is to spend not a single cent more than necessary to obtain the largest number of users who will accept the reliability of the product as marketed. Therefore, the reliability of a product depends on the nature of the product, especially on how it is used, and the impact of bugs on the vendor (so not all bugs are perceived as equally serious by vendor and user).

On the positive side, many software products have been extensively used, by an enormous range of users as compared to the specially developed systems, resulting in products which have achieved considerable maturity. They can be considered to have been extensively tested in the field, but not necessarily in the precise way in which they may be used in a new system. Nor

are all discovered bugs fixed. Economics rules. Bugs are fixed which tend to have a large effect and cost little to fix or which affect a large number of users. "Workarounds" are usually provided for the others.

Ada is the preferred language for the development of high integrity systems. It was specifically developed to support good software engineering principles and consistent standards of verification and validation. This has obvious benefits for system integrity. Yet most software products are developed in C or C++, languages which are positively discouraged by most standards for the development of safety related software.

To summarize, COTS software products are overly complex (because of excessive functionality), insufficiently tested, and implemented using inappropriate technologies for safety related systems. But we continue to use them!

It is unlikely that any complex system can be engineered to be entirely fault free. Errors will be present in both the specially developed application software and in COTS components. The main difference is that specially developed software can be constrained to the functionality required, and system developers and system procurers have more control over the processes used for specially developed software—especially in the timely correction of discovered bugs.

Problems with COTS software can impact the integrity of a system in a variety of ways. COTS failures may result in the loss of key functions or even the entire system. Problems can cause the system to give unreliable or incorrect output.

To illustrate, consider just one area of frequent trouble, dynamic memory management. Most software products include mechanisms to allocate and deallocate memory as required, but deallocation and/or reclamation is nearly always imperfect.

Memory can become fragmented or even permanently "lost." Over a period of time, system performance deteriorates as a system has to search harder to find memory available for allocation, then deteriorates even further as the use of secondary (disk) virtual memory escalates. Eventually, a system can grind to a complete halt due to lack of

free memory. There are operational systems which have to be re-booted on a regular basis to recover memory which has been "leaked" by COTS software. (On the other hand, there are systems which rarely see this bug because they have to be frequently re-booted due to other problems long before memory leakage and fragmentation become significant!)

Similar problems can occur with the allocation and deallocation of disk space—for example, creating and deleting entries in a database. The difference is that fragmented and "leaked" disk space are **not** recovered when a system is re-booted. There are operational systems which have to be partially re-installed on a regular basis to recover disk space which has been "leaked" by COTS software.

For these reasons, many standards for safety related software place a total prohibition on the use of dynamic memory allocation which, if rigorously applied, would preclude the use of many COTS software products.

It is not just the executable COTS software which can cause problems. Extensive data is also available as COTS products. For example, map data is frequently procured as a COTS product for use as part of a larger system development. Operationally, defective or incorrect data can be just as big a problem as errors in executable software, an issue which is frequently overlooked.

The message is that developers need to take measures to control the risk to system integrity which may be introduced by the use of software products, including developing and applying their own extensive tests to COTS software and providing comprehensive mechanisms for limiting the potential impact of COTS failures during operations.

Prior experience of a product provides essential knowledge of solutions and workarounds for faults in the product. However, such experience may only be valid for one version of a product. A newer and less mature version may have a completely new set of problems. This is a particular problem in safety related systems. Organizations using such systems must either allocate disproportionate resources to constantly recertifying the frequent new COTS versions, or risk

*continued on next page*

*"Can COTS Software Be Trusted?"  
continued from previous page*

becoming dependent on an increasingly incompatible, obsolete early version.

Despite problems with COTS software components, many developments have led to usable systems. The operational success of these systems has shown that there are measures which can be taken to reduce the risk associated with COTS components and to produce reliable systems using COTS components. But there have also been unanticipated impacts on cost and time, both during initial system development and during later operation and maintenance of the systems.

The use of COTS components to reduce costs and shorten development times may be a valid design aim, but such products must not be allowed to compromise the integrity of systems. Would you entrust your life to a commercial off-the-shelf software component which averages one failure per day? Possibly, if you can insure that failure of such a component will not compromise system integrity.

*The technical background for this article is drawn from a study into the growing use of COTS software conducted by IPL on behalf of the National Air Traffic Services (NATS), part of the UK Civil Aviation Authority. This article represents the personal views of the author and is not a statement of NATS policy or practice.*

For further information, contact:  
**Eveleigh House, Grove Street  
Bath, BA1 5LR, U.K.**

email: johnl@iplbath.com  
Tel: +44 1225 475000  
Fax: +44 1225 444400

*"FLASH! Art Pyster replaces Floyd Hollister  
FLASH!" continued from page 1*

Prior to his ten years at SPC, Dr. Pyster was involved in pioneering the use of computers for commercial voice processing, including a very early voice mail system. At TRW, he was instrumental in launching the effort to build a standard, high productivity software engineering environment (SEE). This effort served as the basis for later work by others elsewhere in industry. He has been involved with hard real-time software for safety-critical systems, such as nuclear reactor control.

Dr. Pyster is a Senior Member of IEEE and a Distinguished Alumnus of the Engineering College of Ohio State University, where he earned his Ph.D. in Computer and Information Sciences. ■

### FAA SOFTWARE ENGINEERING PROCESS GROUP

Art Pyster Chief Scientist for Software Engineering	AIT-5
Linda Ibrahim SEPG Chairperson	AIT-5
Tanae Gilmore SEPG Secretary	SETA
Rebecca Deloney	AOS-1
Rob Hanes	AUA-200
Bob Laws	ASU-250
Tom Marker	ASU-250
Tom Pearson	AND-630
Natalie Reed	ACT-24
Ross Ridgeway	AMI-1
Art Saloman	ASD-130
Raghu Singh	AIR-200
Cindy King Skiles	AUA-7
Tom Skiles	ATR-300
Marie Stella	AND-8
Rebecca Taylor	ASD-420

## WHAT IF IT HAPPENED DURING DESERT STORM?

Halfway through a huge training exercise in Florida in March of 1995, using the Contingency Theatre Automated Planning System (CTAPS), Air Force operators found they could not open a crucial application. As part of a joint service exercise, based on a hypothetical conflict in the Persian Gulf, users were rushing to complete an air tasking order in the morning of March 21 involving detailed flight plans for 2700 simulated aircraft sorties. Desparate calls to CTAPS operators at Air Force bases across the country quickly proved that the lockout was systemwide.

The problem was not isolated and fixed (using the Internet to distribute the software fix) until fourteen frantic hours later. The trouble was traced to a COTS text editor embedded deep within CTAPS, used for cutting and pasting text into CTAPS as the air tasking order is built. The COTS editor contained an automated "license manager" which had mistakenly been set to shut it down at the end of two years instead of twenty. The text editor had never been considered "critical," since CTAPS does not need the editor to run, but the operators had no other immediate means of manipulating text.

*Editor's Note:* A full report of the above incident, by Paul Constance, GCN staff, appears in the July 8, 1996 issue of *Government Computer News*. ■

## Training Opportunities

The FAA SEPG has developed a training program consisting of the following topics. Classes are to be offered periodically throughout the year. Please contact your organization's SEPG member for schedule and enrollment information or discussion of your software training needs.

- Cost Estimates & Economic Evaluation of Projects
- Software Capability Evaluation Training
- Software Project Planning & Tracking with Delphi Estimation Techniques
- Practical Software Measurement (PSM)
- Software Development Cost and Schedule Estimation
- SLIM

## In This Issue

.....

**1**

Chief Scientist Passes Torch to  
Art Pyster

**1**

Mike DeWalt Speaks at Headquarters

**2**

Improving Acquisition of Software  
Intensive Systems

**3**

Ariane 5 - Why Duplicated Software  
is NOT Redundant

**4**

Software Testing of Safety Critical  
Systems - Part 2

**5**

Letter from the Editor:  
Software Interlocks are...NOT!

**6**

Can COTS Software Be Trusted?

**7**

What If It Happened During  
Desert Storm?

# interFACE

NEWSLETTER OF THE  
SOFTWARE ENGINEERING  
PROCESS GROUP

VOLUME 6, NUMBER 1  
FEBRUARY 1997



## CONFERENCE CALENDAR

**Software Engineering Institute (SEI)  
Software Engineering Process Group Conference**  
March 17 - 20, 1997  
San Jose, CA  
*Registration: (412) 268-7388*

**SEI Conference on Risk Management**  
April 7- 9, 1997  
Virginia Beach, VA  
*Registration: (412) 268-7388*

**Software Technology Conference**  
April 27 - May 2  
*Contact: (801) 521-9055 or (801) 521-2822*

**The Software Engineering Symposium**  
August 25 - 28, 1997  
Pittsburgh, PA  
*Registration: (412) 268-7388*

**Air Traffic Control Association**  
September 28 - October 3, 1997  
Washington, DC

**Federal Software Process Improvement Working Group (FEDSPIWG)**  
Held Monthly at NOAA  
*Contact: Martha Morphy at NOAA (301) 713-3345*

DOT/FAA/AIT-5  
800 INDEPENDENCE AVENUE, SW  
WASHINGTON, DC 20591