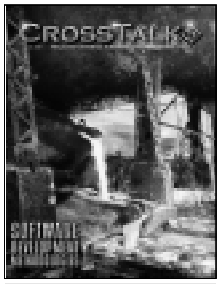
CrossTalk

June 2001 The Journal of Defense Software Engineering Vol. 14 No. 6

Software Development Methodologies


On the Cover: Kent Bingham, Digital Illustration and Design, is a self-taught graphic artist/designer who freelances print and Web design projects.

Departments

CrossTalk

Sponsor	*Lt. Col. Glenn A. Palmer*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pam Bowers*
Associate Editor	*Benjamin Facer*
Article Coordinator	*Nicole Kentta*
Creative Services Coordinator	*Janna Jensen*
Voice	801-586-0095
Fax	801-777-5633
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/Crosstalk/crostalk.html
CRSIP Online	www.crsip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail or use the form on p. 24.

Ogden ALC/TISE
7278 Fourth St.
Hill AFB, Utah 84056-5205

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at www.stsc.hill.af.mil/CrossTalk/xtlkguid.pdf. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with CrossTalk.

Trademarks and Endorsements: This DoD journal is an authorized publication for members of the Department of Defense. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the government, the Department of Defense, or the Software Technology Support Center. All product names referenced in this issue are trademarks of their companies.

Coming Events: We often list conferences, seminars, symposiums, etc. that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the CrossTalk Editorial Department.

STSC Online Services: at www.stsc.hill.af.mil
Call 801-777-7026, e-mail: randy.schreifels@hill.af.mil

Back Issues Available: The STSC sometimes has extra copies of back issues of CrossTalk available free of charge.

The Software Technology Support Center was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies to improve the quality of their software products, efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery.

Times, They Are A Changin' for Software Development

It is said that people fear change. However, we embrace change all the time when the results are known. We change what we eat (three meals a day of Wheaties doesn't sound very good). We change what we wear (the same pair of socks all month isn't a pleasant image). It isn't change that people fear as much as the unknown results of change. Have you ever watched a tape-delayed game on TV when your team won? I have. At halftime, when my team was down by 25, I wasn't nervous or afraid at all. Why? I knew the outcome.

This issue of CrossTalk focuses on the changing landscape of software development and some of the new methodologies being pioneered in the software industry. Having been successfully implemented in the commercial software industry, Object Orientation and the Universal Markup Language (UML) present some potential benefits to the real-time and embedded systems industry. From ARTiSAN Software Tools, Alan Moore suggests that some extensions to the UML are needed to address the lack of modeling techniques for real-time and embedded systems.

LTC Michael Bowman, Dr. Antonio Lopez Jr., and MAJ James Donlon from the U.S. Army War College and Dr. Gheorghe Tecuci from George Mason University present a new paradigm for software development. In their article you will discover Disciple, an artificial intelligence approach to the development of knowledge-based systems. Disciple is intended to replace the indirect transfer of knowledge and expertise from the end-user to the development team, and then into the software product with direct user development of the system with the assistance of an intelligent software agent. This is quite a revolutionary approach that may cause you to rethink traditional defense software development methodologies.

In a world where acquisition wants things better, faster, and cheaper, and software engineering requires process, discipline, and rigor, extreme methods may be the solution. From the Software Technology Support Center, Theron Leishman discusses Extreme Programming (XP) and suggests some caveats when implementing this innovative development methodology.

At a time when software process is being touted from every rooftop, Richard Duncan, a master's student at Mississippi State University further suggests that XP be designed for medium- to small-sized organizations. XP, designed with requirements drift as a fundamental occurrence, nominates coding as the key activity throughout the development process. While sounding a bit extreme, Duncan's article provides some thought provoking ideas.

From the Software Engineering Institute, Drs. Stephen Cross and Caroline Graettinger suggest that, given the rapid pace of change in software engineering, software-intensive organizations must develop a core competency for proactive change management. They also suggest that a career path in change management will be a critical need and opportunity for the 21st century software engineer.

Six valuable lessons learned about the acquisition of commercial, off-the-shelf-based systems (CBS) are at the heart of an article from Richard Adams and Suellen Eslinger of The Aerospace Corporation. Adams and Eslinger, having performed an in-depth study of actual CBS, identify some shortcomings of, and potential changes to, the acquisition and development of COTS-based systems.

Change really is a part of the software development landscape. We hope that these articles will help you gain a greater understanding of this changing landscape so that when someone suggests a *new approach*, your first reaction is not one of fear. As always, your comments and questions are welcome.

[illegible]

Kevin Richins
Software Technology Support Center

Extending UML to Enable the Definition and Design of Real-Time Embedded Systems

Alan Moore
ARTiSAN Software Tools

The complexity and size of the average real-time system is increasing rapidly. The development of detailed system designs is vital if the system is to be well understood and to function correctly. In a bid to extract maximum reuse of system software components, and to overcome previous bad software development experiences, many senior system and software engineers are looking to Object Orientation (OO) as the design paradigm for new system development. This article discusses the suitability of Universal Markup Language, the popular OO modeling language, for defining and designing real-time and embedded systems.

Object Orientation (OO) has been successfully applied to the development of commercial software, but has the potential to be even more readily embraced by engineers in the real-time domain, where familiarity with the componentized nature of hardware leads naturally into the concept of objects.

The search is on for appropriate real-time OO modeling techniques, and not surprisingly the Unified Modeling Language (UML) is the notation most often proposed as the base for the techniques. The rise in UML's popularity has been rapid since its inception some 4 years ago. The fact that it is promoted by some of the biggest 'names' in the OO arena and supported by an increasing number of tool vendors goes some way to explain why.

Although it is a well-defined and flexible modeling language its origins in the world of commercial computing show up in the lack of certain modeling techniques for real-time systems and embedded systems. For most (if not all) real-time systems, the issues of timeliness and ability to schedule are paramount. By definition, real-time systems designers are concerned with time and ensuring that certain system activities can be carried out within defined time scales. This in turn requires consideration of the system's concurrency: The number and the characteristics of the concurrent tasks and the degree of task interaction between them can significantly affect system performance. Embedded systems designers are also concerned with the physical architecture of the embedded system and the hardware/software interface. None of these concerns are well met by UML at present.

Recognizing this, the Object Management Group[1] has spawned a new initiative, the Real-time Analysis and Design Group (RTAD), specifically to recommend UML extensions in this area.

The System Development Process

The design of real-time embedded systems involves a multi-disciplined team of hardware, systems, and software engineers. System engineers design at a high level and influence both hardware and software composition of the system. They are concerned with the overall architecture and usually make trade-offs between implementing functionality in hardware, software, or both. Hardware engineers are concerned with the design of circuitry that will fulfill the system requirements as determined by the systems engineers. Software engineers usually have the largest design and implementation task, as the majority of the functionality of an embedded system lies typically within the software.

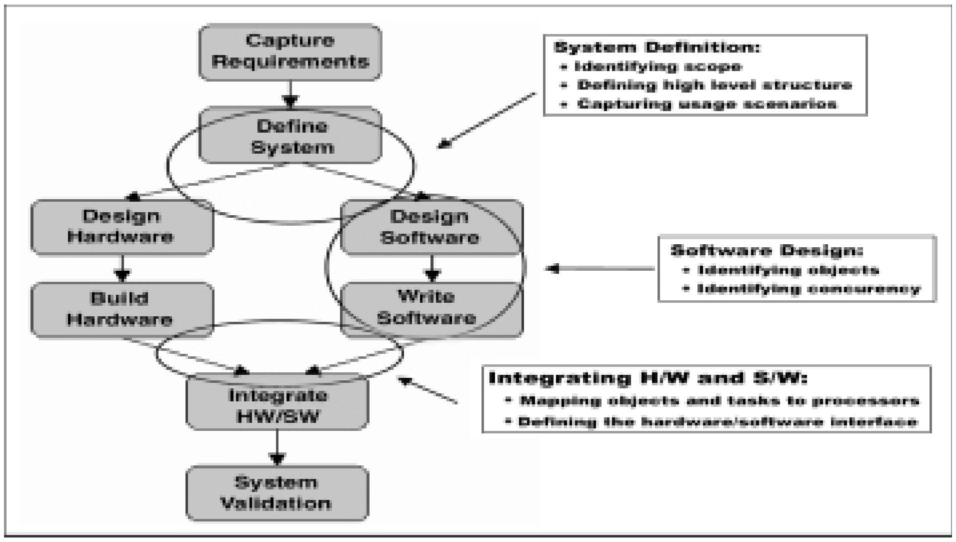
This (simplified) system development process illustrated in Figure 1, shows three critical areas of developing real-time/embedded systems: system definition, software design and hardware/software integration. Those areas are circled with some of the key activities outlined in the associated call-outs.

System Definition

The process of defining a system is often called system engineering. It involves the following:

- Identify the structure of the system, both in terms of hardware and software. For many real-time systems (e.g. mass produced embedded systems) major architectural decisions are often made early in the development life cycle. Almost all real-time developments will experience change in the system architecture during the development process. The nature and implications of the architecture, and any changes to it, need to be communicated among, and understood by, system, hardware and software engineers.
- Clarify communication between the system and the environment. Clarifying both the boundary of the system and

Figure 1: *The System Development Process*



the types of messages exchanged is important both from the point of view of scoping the system and agreeing interfaces.
- Capture system usage and therein timing and other types of system performance (often called quality of service) associated with that use. System usage requirements often include specific timing information relevant to product performance. These timing constraints are captured and decomposed during the system definition stage in order to facilitate a system design that is fit for purpose, and to support system test. Real-time systems, by definition, are constrained by some aspect of time. The right answer received late is often wrong. The requirements driving the development of these systems often state specific timing considerations. It is critical to capture this timing information in the early system definition stage.

Software Design

As well as the standard need to model objects and their interaction, real-time system models also need to address concurrency. There is an inherent concurrency in most real-time, embedded systems as they control multiple input/output devices through a variety of interfaces simultaneously. Designing a solution for this type of problem often involves a multi-tasking, and possibly multi-processor architecture. The task and object design need to be integrated, so the ability to model both together is essential.

Hardware and Software Interfaces

Real-time embedded systems contain both hardware and software elements as part of the solution. Many of the worst and most time-consuming problems with developing real-time systems manifest themselves during system integration, often through poor interface specifications. Therefore, understanding the interfaces and interdependencies between the hardware and software is a necessary step in building a correct system. Here are examples of the types of data required:
- Software/hardware interface information, e.g. port addresses, memory maps.
- Hardware characteristics such as speeds, capacities, etc.

An Example

Throughout the remainder of this article an example of a real-time system will be used to present the basics of the UML, as well as to illustrate the extensions required to usefully define and design a solution to a typical problem.

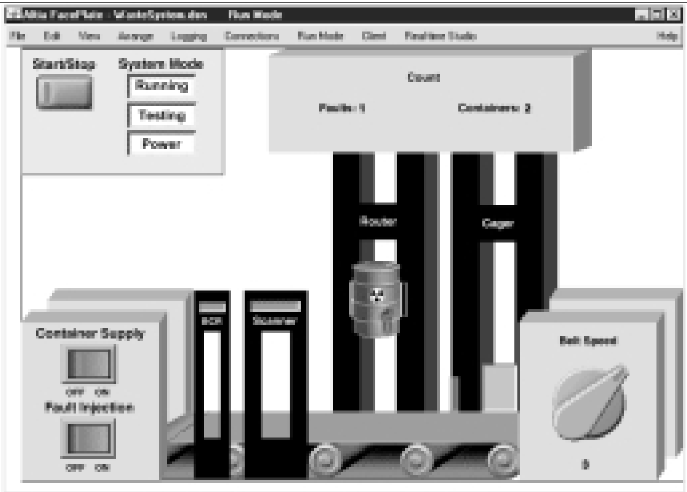


Figure 2: *Toxic Waste Processing Plant Simulation*

Figure 2 shows a simulation of a toxic waste processing plant. The plant handles containers of toxic waste, which are fed into a processing plant along a conveyor belt. Once detected and identified by an attached bar code, a container is scanned for faults before routing. If a container is badly damaged then it is routed off the belt by the first robot arm for special handling: If not, then it reaches the second robot arm where it is sent off for caging and normal processing. This simulation also provides diagnostic switches to supply containers and inject faults.

The Advantages and Deficiencies of UML

Here the three areas described in the system development process are revisited, highlighting how UML helps model these areas, and also where it is deficient. In those cases, additions or changes to UML are suggested that address those deficiencies.

System Definition

System Context (or Scope)

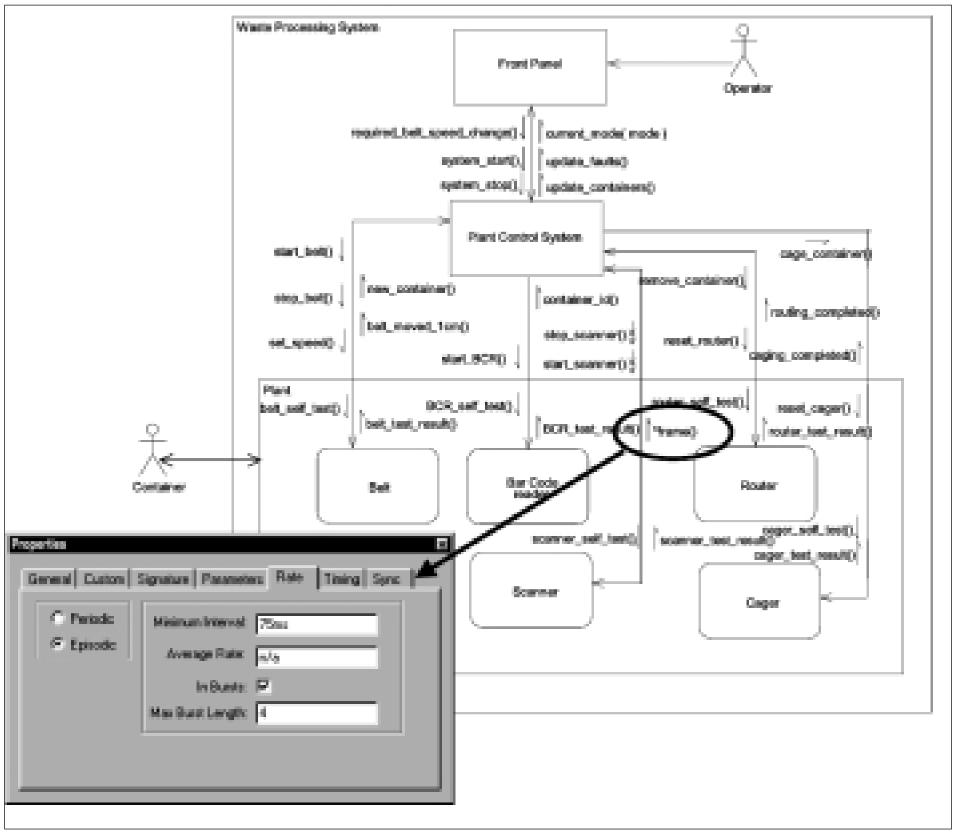
One simple and often overlooked consideration when mapping out the parts of the solution is the external interface boundary. This concept is useful in determining what is within the context of the solution and what is outside this context. UML has no good notation for supporting this; engineers often use an object collaboration diagram, but this involves faking system-level devices as objects, which can cause confusion when *real* objects are introduced.

A System Scope Diagram, an extension to UML, can be used early in the problem analysis to help isolate the boundary of the analysis activity. It provides a graphical checklist of all of the external and interface boundaries in the solution. It is an easy way to make sure that the complete problem is being addressed. It shows external actors, hardware interface devices, and the system itself, as well as the messages (often called signals) that pass between them. One of its most important roles is to define a vocabulary for describing how the system interacts with the environment when it's being used. It is very similar to the context diagram used in traditional strutured techniques. (Sadly the UML Deployment Diagram does not allow actors or message flows and so it is hard to use for this purpose.)

The System Scope Diagram in Figure 3 (see page 6) shows how all the major components of the system (hardware and software) communicate. The plant control system itself is started and stopped by the front panel subsystem, and controls the plant subsystem. Each of the messages (or signals) has a rich description as exemplified by *frame*; frames occur only while a container passes through the scanner, hence they are episodic, and occur in bursts of four with a minimal interval of 75ms.

System Architecture

Having scoped the system, a major task is to identify the high-level architecture of the system, adding major architectural elements such as subsystems, buses and disks,



( ) Periodic
(●) Episodic
Minimum Interval: 75ms
Average Rate: n/a
In Bursts: ☑
Max Burst Length: 4

Figure 3: *The System Scope Diagram*

decomposing subsystems into boards and connecting them up appropriately. The UML has the Deployment Diagram for this, but it is woefully lacking in notation for buses, boards, etc., and so cannot be used to present an easily comprehensible picture to the engineering team developing an embedded system.

The System Architecture Diagram presented in Figure 4, another extension of the UML, is a form of Deployment Diagram, but with greatly enhanced symbolism[2], and an underlying type model allowing board, bus, disk etc., types to be defined once and subsequently reused.

The plant control system is a rack-mounted PC with three boards: the motherboard, a network card, and an I/O board. The network card is there to facilitate communication with the factory bus. The I/O Board will be off-the-shelf but will need enough ports to handle the various devices attached to it, including the Front Panel devices (not shown).

Use Cases

Use case modeling is the primary UML technique for specifying functional requirements, and one of the most widely accepted and used. A Use Case Diagram illustrates the ways that elements external to the system (actors) interact with the system under development. Actors may be people, for example end users, operators, maintenance engineers, etc., or they may be other systems or items of equipment. All actors are represented diagrammatically by the stick person symbol. Lines connect the actors to one or more use cases (the ellipses). A use case represents a system service provided to one or more of the actors linked to it. Associated with each use case is a textual description of the activity involved in the delivery of that service.

Figure 4: *The System Architecture*

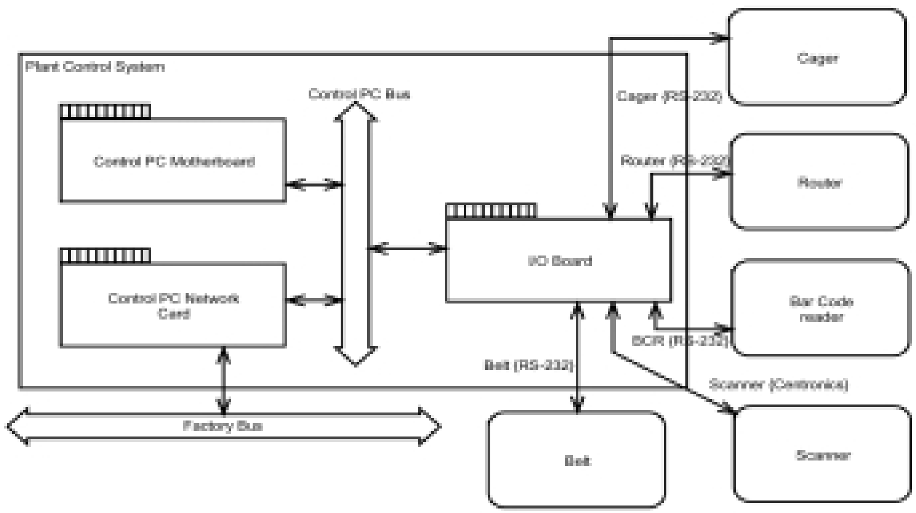


Figure 5 shows a Use Case Diagram, identifying the major functional requirements of the system. The major use case, process container, makes use of two lesser use cases, scan container and detect container, hence the <<include>> relationship. There are two exceptional cases highlighted by the <<extends>> relationship, handle defective container and emergency stop and restart which describe deviations from the standard behaviour of process container.

Use case descriptions provide a specification of functional requirements in a form that is understandable to, and therefore can be validated with, the system sponsor. However the main drawbacks of using natural language for specification (ambiguity, misconception, duplication, etc.) may be present, although not necessarily visible. UML provides a modeling technique that helps in overcoming these drawbacks: the Sequence Diagram.

Sequence Diagrams

The Sequence Diagram presents a usage scenario in a useful and intuitive way, as an ordered sequence of message exchanges (or interactions) between system elements. Although UML provides the capability for interaction modeling with Sequence Diagrams, it does not make it easy to distinguish between the different types of elements involved. The implicit assumption in UML is that this diagram exists to illustrate message passing between the actors and software objects. This is jumping the gun during system definition. It is certainly the case that, during the design stage, we will need to model object interaction.

But at the requirements stage we ideally want to express the details of use case activity by clearly indicating the interaction between the actors, hardware devices, and the entire software system; these are likely to be far more familiar to the project sponsor.

The Scanning a Container diagram in Figure 6 is an extension to UML. As can been seen in the diagram the plant control system is informed of belt movements (asynchronously, indicated by the half-arrow). In this scenario it determines that a container is approaching the scanner, whereupon it starts the scanner, receives four frames of scan data from it, and then stops the scanner again. The system waits for the scanner to both start and stop before proceeding (indicated by the cross on the arrow shaft). The frame takes 50ms to capture and 25ms to transmit from the scanner. This gives a total scan time of 375ms. Note that all of the elements shown here are derived from the earlier System Scope Diagram, Figure 3.

Altough the UML allows annotations to be added to any diagram, it is more useful if timing notes, in particular, are mapped directly onto the appropriate model elements (as tagged values), making timing information more than just notes on the Sequence Diagram. There are two main categories of timing information that can be modeled, latency and duration. The amount of time it takes for two entities to collaborate (i.e., communicate) can be described as the message latency. The amount of time required for an entity to perform its task once it has received a signal is described as the duration. Through a combination of latency and duration specifications, we identify general timing constraints over a sequence of steps in a usage scenario (as an end-to-end time). These constraints map back onto specific, use cases and the original requirements.

Software Design

Class Diagrams

The UML Class Diagram is perhaps the best-known aspect of UML. Class Diagrams form the basis for the object architecture of the system. They document the static nature or architecture of the objects in the system. The Class Diagram is concerned with static structure and designing good partitioning (encapsulation) and abstraction. It works well as is for real-time systems design.

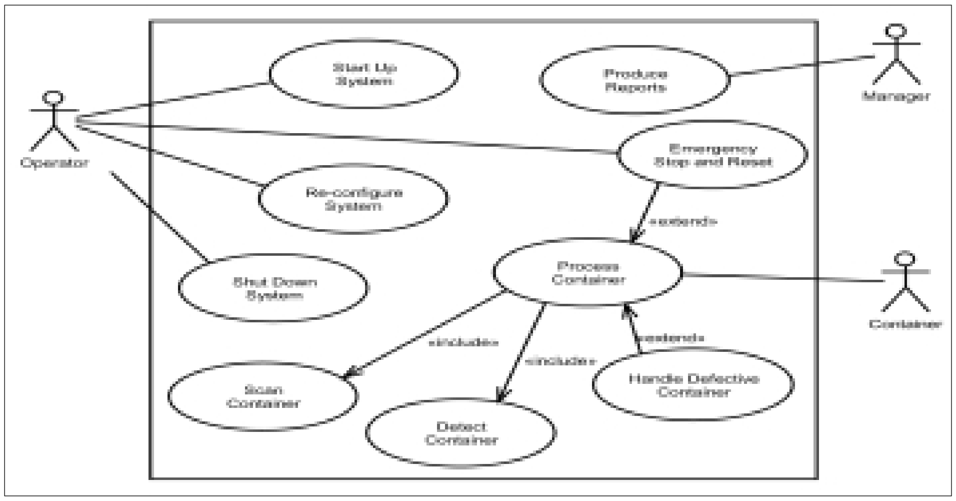


Figure 5: *Use Case Diagram*

Collaboration Diagrams

Where the Class Diagram defines a static relationship structure between the classes, the Collaboration Diagram defines a communication structure between the objects of those classes. Collaboration Diagrams can be used to detail specific scenarios (referred to as realization in UML) or as a means of synthesizing the overall collaboration of objects.

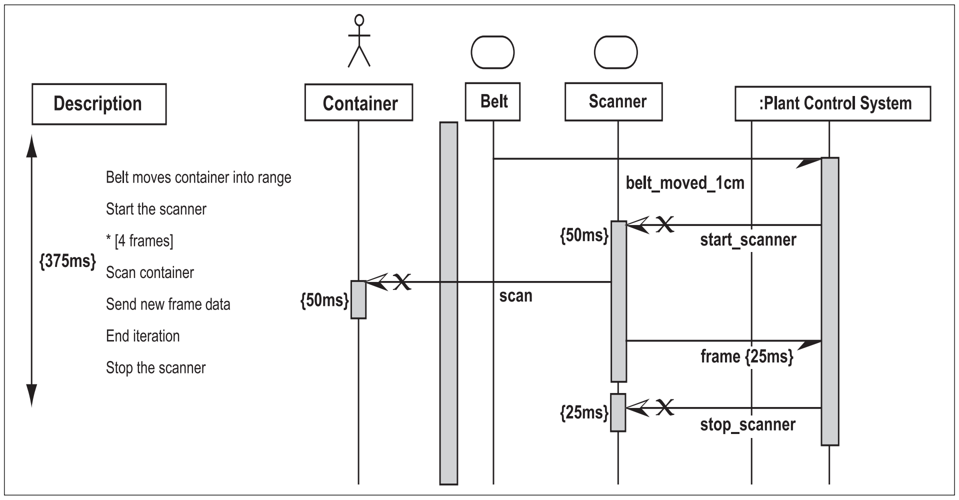
The UML Collaboration Diagram in Figure 7 (see page 8) shows the collaborating objects and the sequence of operation calls (via the sequence numbers) to scan a container. The object *theBelt* tracks all containers so is well placed to inform *theScanner* when one approaches. theScanner communicates through *theScannerDriver* to control the *Scanner* device and to receive frame data. Once all frames are processed the status of *currentContainer* is updated with the result. The messages between the drivers and interface devices provide continuity back to the earlier Sequence Diagram (see Figure 6).

By analyzing several Sequence Diagrams together as a whole and representing the overall connectivity of the objects involved in those scenarios, it is possible via a Collaboration Diagram to visualize the communication architecture of the proposed design. The Collaboration Diagram also gives some insight into the dynamic relationships between the objects. This allows us to create a more complete view of the overall communication architecture as opposed to any one specific scenario.

Concurrency Diagrams

What the Collaboration Diagram cannot describe is whether the executing objects are concurrent or not, or how they use a real-time operating system (RTOS) to communicate. Concurrency has been a concern in the real-time domain for many years and there are a variety of recognized strategies for identifying tasks and opti-

Figure 6: *Scanning a Container*



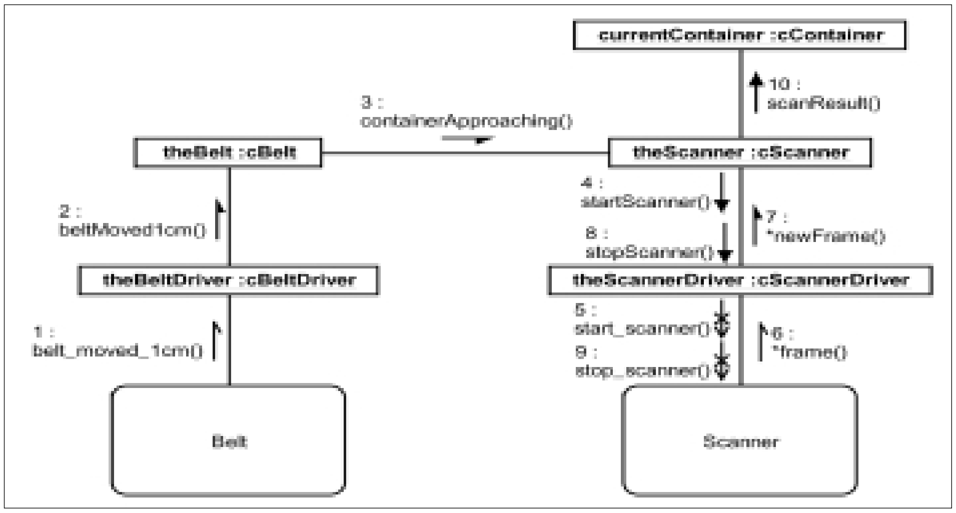


Figure 7: *The UML Collaboration Diagram*

mizing task design. In such a complex area of design, modeling can play a vital role in verifying the feasibility and correctness of a concurrency design. There is no current UML notation clearly representing tasks, or any of the RTOS mechanisms that can be used to protect shared resources or facilitate inter-task communication.

A diagram that shows the tasks, shared resources, mutexes, queues, event flags, mailboxes, etc., together with the way in which these various components interact, is required. The diagram will need to illustrate message passing in a clear and unambiguous manner so that it is clear exactly where and how the RTOS is being used. It is possible to produce a UML diagram that can represent the information shown in using standard notation Figure 8, but the UML does not have the richness of symbolism that makes this particular diagram so clear. This is not an unimportant point. Clarity is an aid to understanding; models that can be easily misinterpreted can, in some cases, be worse than no models at all.

We can now see how *cScanner* actually handles frames by looking at its internals. The identifiers of approaching containers (provided by the call to *containerApproaching*) get queued for processing by the task to scan containers. The arrival of new frame data (indicated via the call to *newFrame*) signals an event flag upon which a task (handle frame) is waiting. As frames are received (via some new calls to *theScannerDriver*), they are stored and their identities (frame ids) are passed to another task (process image) that processes the image for faults.

Hardware and Software Interfaces

Deployment Diagram Limitations

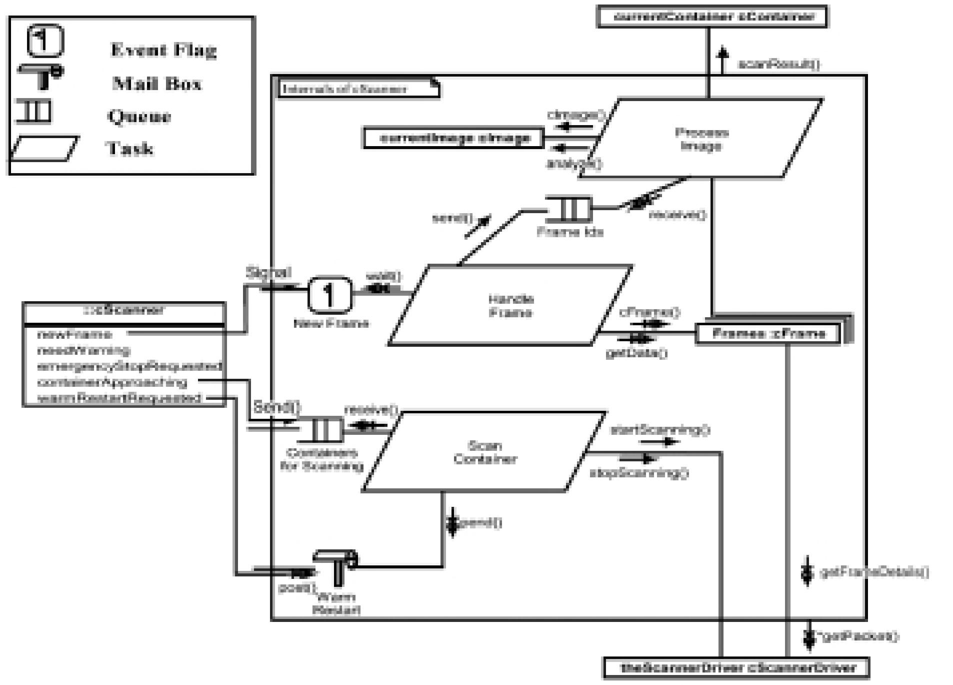
A thorough understanding and representation of the software/hardware interfaces and the allocation of software to hardware is important to construct an application that not only meets functional requirements but also can be deployed into the proper environment.

The UML provides a Deployment Diagram for capturing this kind of representation. It allows engineers to place objects (software) on a set of connected nodes (hardware). However, its description of the processing architecture is very simple, and it allows no account of the placement of tasks at all. It offers a very limited set of symbols that make it difficult to adequately represent the range and characteristics of the physical entities inherent in many real-time systems. Nor are Development Diagrams able to easily model interface information such as the memory map and interrupt request vectors.

In order to support this type of description, a much more detailed view of the board and processor architecture is needed with specific model elements to describe them. This type of information neatly fills the gap between traditional UML and the detailed hardware schema available from Electronic Data Access (EDA) tools. Hardware related information like this enables software engineers to undertake detailed device driver development, without the need to refer to hardware schematics generated by the hardware team. Figure 9 shows an Architecture Diagram as an extension to the UML Deployment Diagram. This diagram allows both hardware and software engineers to access the same information, significantly improving communication, and thus reducing the number of errors in integration testing.

Figure 9 shows the connections at the board level. Each of the board's I/O devices is further described to the detail required by the software, as is shown in the overlay. We can see the memory map as well as any IRQs or I/O port addresses.

Figure 8: *The Internals of cScanner*



Progress in the OMG

Recognizing the need for real-time extensions to UML, two Object Management Group (OMG) task forces, the Analysis and Design Task Force and the Real-time Task Force jointly spawned a specific group to look at extending UML to support real-time systems.

The Real-time Analysis and Design Group

The goal of the Real-Time Analysis and Design (RTAD) initiative is to issue a comprehensive set of Requests for Proposals (RFPs) leading to OMG standards that will support the use of object-oriented approaches in the analysis, design, and development of real-time computing systems.

There have been three RFP's proposed by the RTAD:

- UML™ Profile for Scheduling, Performance, and Time;
- UML™ Profile for Quality of Service (QoS) other than timeliness;
- UML™ Profile for large-scale, distributed systems.

The first of these has been formally issued by the OMG; its schedule shows the following:

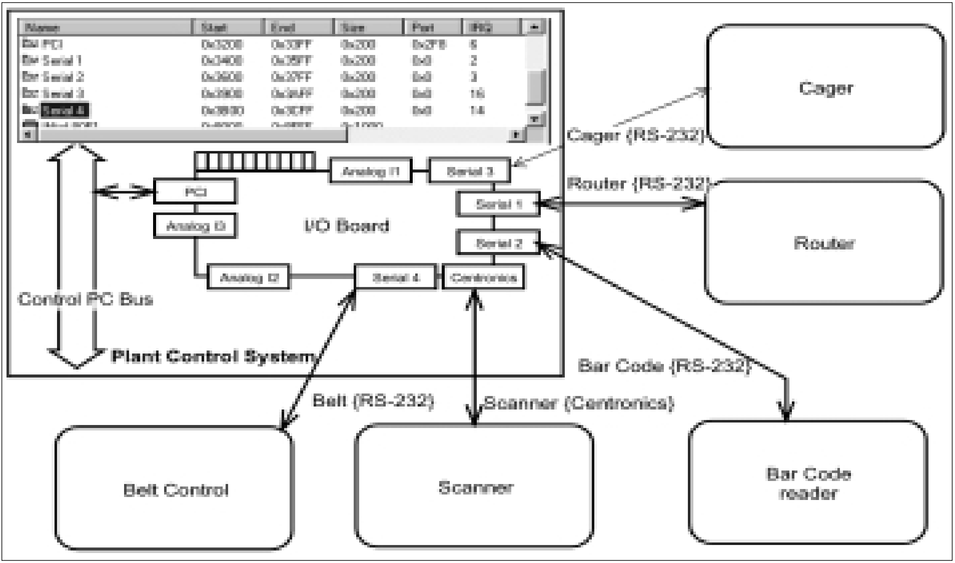
- Initial submissions received in September 2000.
- Revised submissions due in July 2001.
- Approval by the Architecture Board of the OMG by October 2001.
- Specification adoption by January 2002.

The Submission Team

There looks likely to be only one team offering a submission for the initial RFP. The team consists of modeling tool vendors ARTiSAN Software Tools, I-Logix, ObjecTime (with Rational) and Telelogic, and schedulability of tool vendors Tri-Pacific Software and TimeSys.

Conclusion

Given the popularity and notational robustness of the current version of UML, real-time developers can reasonably begin exploiting OO technology in their efforts. However, it is important to realize that certain characteristics of real-time systems may be difficult, if not impossible, to capture in most modeling tools limited to standard UML notation and semantics. Real-time development demands extensions to UML provided only by specialized tools. This article proposes some feasible extensions to UML shown in Table 1. ◆



Name	Start	End	Size	Port	IRQ
PCI	0x3200	0x33FF	0x200	0x2F8	5
Serial 1	0x3400	0x35FF	0x200	0x0	2
Serial 2	0x3600	0x37FF	0x200	0x0	3
Serial 3	0x3800	0x39FF	0x200	0x0	16
Serial 4	0x3B00	0x3CFF	0x200	0x0	14

Figure 9: *An Architecture Diagram Showing Board-Level Connections*

Notes

1. The Object management Group (OMG), www.omg.org, is a consortium of companies and other interested parties responsible for the specification and standardization of the Unified Modeling Language. OMG ratified the inital version of UML, UML 1.1, in 1996 and has more recently specified and ratified the current version, UML 1.3 (June 1999).
2. UML provides three formal extension mechanisms, namely constraints, stereotypes, and tagged values. The latter two mechanisms can be used to provide additional symbolic elements such as busses, boards, and disks as extensions to the standard notational elements called classifiers.

Table 1: *UML Collaboration Diagram*

Standard UML Diagrams Referred to in this article	Extensions to UML for Real-time development
Class Diagram Use Case Diagram Sequence Diagram Collaboration Diagram Deployment Diagram	System Scope Diagram Architecture Diagram (Extended) Sequence Diagram Concurrency Diagram

About the Author

Alan Moore has 15 years experience in the development of real-time and object-oriented methodologies, and their applications in a variety of problem domains. He has been actively involved in product development, training, and consulting related to Object Oriented Analysis and Design and structured development tools. Moore has co-authored a book on graphical user interface design and published several papers, and has lectured on a wide variety of analysis and design issues. Moore is responsible for the specification, planning and management of the ARTiSAN product strategy. He is the author of ARTiSAN Real-time Perspective, a pragmatic approach to the development of real-time systems and is an active participant in the Real-time Analysis and Design Group of the Object Management Group .

Diagrams showing the extensions discussed in this article were prepared using Real-time Studio from ARTiSAN Software Tools (www.artisansw.com), a UML-based modeling tool used by real-time and embedded systems developers.

Vice President of Strategy
ARTiSAN Software Tools
Stamford House, Regent St.,
Cheltenham, Glos., UK GL501HN
Voice: +44-1242-229-300
Fax: +44-1242-229-301
E-mail: alanm@artisansw.com
www.artisansw.com

Teaching Intelligent Agents: Software Design Methodology

LTC Michael Bowman, Dr. Antonio M. Lopez Jr., MAJ James Donlon
U.S. Army War College

Dr. Gheorghe Tecuci
George Mason University

Current software design and development methodologies have evolved slowly since the early days of computing and today almost universally include user-developer interaction for requirement determination, testing, and acceptance activities. A new paradigm for software development is emerging from U.S. government sponsored research in artificial intelligence for rapid knowledge-based systems development. This new paradigm, being pursued by the George Mason University Learning Agents Laboratory, calls for end users to interact directly with an intelligent software agent to develop their own problem solving intelligent agents. This approach is called Disciple, and it has been experimentally shown to rapidly produce high performance software agents for solving complex military problems.

Software design methodologies are intended to help develop computer programs more systematically. They are sets of procedures that people follow from the beginning to the completion of the software development process. Since the 1970s, the numbers of software design methodologies have increased significantly. Khoo [1] gives an overview of some of the major ones, and Sorenson [2] narrows the focus to a comparison of those methodologies used by Department of Defense software developers. In both presentations, the authors recognize that the selection of an appropriate methodology depends on a number of factors, including the type of problem being addressed and the qualifications and training of the people using the methodology.

It is universally recognized that software design is a creative process that cannot be reduced to a routine procedure; however, it is not devoid of structure. For the typical data processing or information derivation problem, the framework for software design and development is given in Figure 1.

The user specifies the requirements (usually poorly) and the software development team, after gathering data and clarifying some ambiguities, selects and uses a design methodology to develop the desired software product. The software development team may have many members, including systems analysts, senior and junior programmers, quality assurance personnel, and technical writers. The process is error prone.

As Humphrey [3] puts it: "The process is so complex because many different people are involved and they are all learning. At the outset, no one really understands either the requirements, the design, or the implementation. This is particularly true when the implementation finally progresses to the point where the users can first try the product (p. 313)."

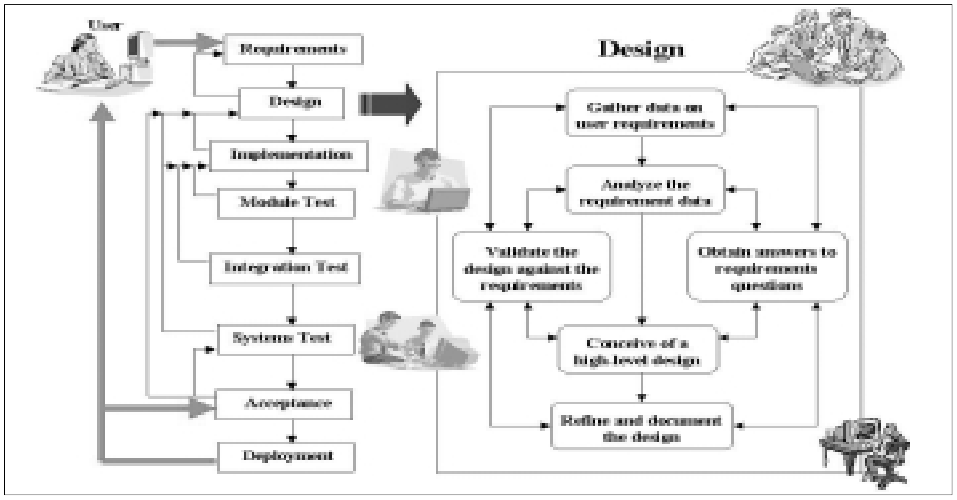
There is a class of difficult problems and a group of highly qualified and trained people who routinely solve such problems, but these people are not software designers or developers. We categorize the problems as knowledge-based problems and the people as subject-matter experts (SMEs). The military has SMEs in many critical areas, and they possess priceless knowledge in how to solve problems of great importance. The skills, insights, and creativity of such individuals make them true craftsmen.

For several decades artificial intelligence (AI) researchers have addressed the problem of developing knowledge-based software agents that incorporate the expertise of the SMEs and exhibit a similar problem-solving behavior. Generally, the agent includes two main components: a knowledge base and an inference engine. The knowledge base contains the data structures representing the entities from the expert's application domain such as objects, relations between objects, classes of objects, laws, actions, processes, and procedures. The inference engine consists of the programs that manipulate the data structures in the knowledge base in order to solve problems in a way that is similar to how the expert solves them.

Typically, developing an intelligent software agent that assisted or replicated the SME required the SME to work closely with a knowledge engineer who would actually build the agent [4]. In this case, the framework for the software development is analogous to that presented in Figure 1 with the software design effort of the knowledge engineer being based more on gathering and representing knowledge than on data and information. Here we view information as interpreted data, but knowledge is information in action or information transformed into capability for effective action [5].

The process of acquiring knowledge from an SME and representing it in the knowledge base of the agent has been found to be difficult and labor intensive, and is what has come to be known as the *knowledge acquisition bottleneck*. The process is slow and difficult because knowledge engineers and domain experts do not initially speak the same language. During the process of indirect knowledge

Figure 1: *A Typical Framework for Software Development*


User
Requirements
Design
Implementation
Module Test
Integration Test
Systems Test
Acceptance
Deployment
Design
Gather data on user requirements
Analyze the requirement data
Validate the design against the requirements
Obtain answers to requirements questions
Conceive of a high-level design
Refine and document the design

transfer from the SME through the knowledge engineer into the agent's knowledge base, the SME and knowledge engineer must achieve a common understanding of the domain and how problems are solved in the domain. They must also produce a mutually understood representation of the domain and problem solving. There is in a sense, a cross leveling of language and expertise.

A Different Paradigm

A contradiction exists in the classical knowledge-based agent development process where the presence of the knowledge engineer is both part of the solution and part of the problem. An additional difficulty with this paradigm is that the development of each new knowledge base or agent starts from scratch, with no knowledge reuse in spite of the fact that knowledge acquisition is such a difficult process. The Learning Agents Laboratory (LALAB) at George Mason University (GMU) is developing a new approach to agent development that addresses these problems and significantly alters the framework in Figure 2.

The goal is to enable an SME that does not have prior knowledge engineering experience to develop a knowledge-based agent by himself or herself, with limited or no support from a knowledge engineer. This approach, called Disciple [6], is based on developing a very capable learning agent that has two main characteristics: 1) it uses a type of knowledge representation and organization that facilitates knowledge reuse and learning, and 2) can be directly taught by an SME how to solve domain-specific problems in a manner that resembles the way the SME would teach a human apprentice – by giving the agent examples and explanations as well as by supervising and correcting its behavior.

Over the years, the LALAB has developed a series of increasingly more capable learning agents from the Disciple family. The general architecture of a Disciple agent is shown in the upper right side of Figure 2. The problem-solving engine is based on the general problem reduction paradigm of problem solving and is therefore applicable to a wide range of domains. In this paradigm, a problem to be solved is successively reduced to simpler problems until the problems are simple enough to be solved immediately. Their solutions are then successively combined to produce the solution to the initial problem.

The learning and knowledge acquisition engine integrates several learning strategies synergistically, such as learning from examples, learning from explanations, and learning by analogy, in order to acquire the knowledge from the SME. The knowledge base is structured into two distinct components: an object ontology and a set of reduction and composition rules. The object ontology is a hierarchical representation of the objects and types of objects from a particular domain, such as military or medicine. That is, it represents the different kinds of objects, the properties of each object, and the relationships existing between objects. The object ontology provides a representation vocabulary that is used in the description of the reduction and composition rules. Each reduction rule is an *if-then* structure that expresses the conditions under which a problem P_1 can be reduced to the simpler problems $P_{11}, \ldots, P_{1n}$. Similarly, a composition rule is an *if-then* structure that expresses the conditions under which the solutions $S_{11}, \ldots, S_{1n}$ of the problems $P_{11}, \ldots, P_{1n}$ can be combined into a solution S_1 of P_1.

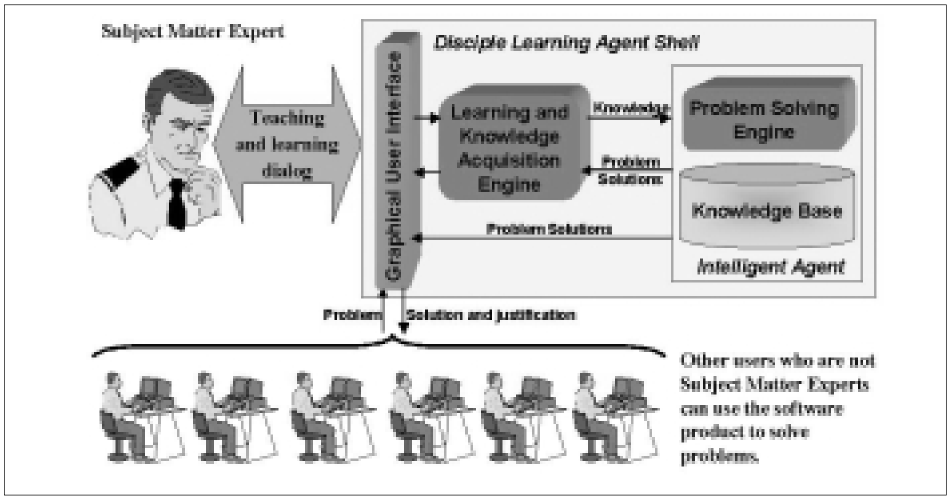
Dividing the knowledge base into an object ontology and a set of rules is very important, because it clearly separates the most general part of it (the object ontology), from its most specific part (the rules). Indeed, an object ontology is characteristic to an entire domain. In the military domain, for instance, the object ontology will include descriptions of military units and of military equipment. These descriptions are most likely needed in almost any specific military application. Because building the object ontology is a very complex task, it makes sense to reuse these descriptions when developing a knowledge base for another military application, rather than starting from scratch.

In the case of Disciple, the ontology reuse is further facilitated by the fact that the objects and the features are represented as frames, based on the knowledge model of the Open Knowledge Base Connectivity (OKBC) protocol. OKBC has been developed as a standard for accessing knowledge bases stored in different frame representation systems [7]. Therefore, importing an ontology from an OKBC compliant knowledge server, such as Ontolingua [8] or Loom [9], does not raise translation problems.

The rules from the knowledge base are much more application-specific than the object ontology. Consider, for instance, two agents in the military domain, one that critiques courses of action with respect to the principles of war, and another that plans the repair of damaged bridges or roads. While both agents need to reason with military units and military equipment, their reasoning rules are very different, being specific not only to their particular application (critiquing versus planning), but also to the SMEs whose expertise they encode. Therefore, the rules are generally not reusable from one application to another, and have to be defined for each new application.

Several decades of knowledge engineering attests that the traditional process

Figure 2: *A New Framework for Software Development*


Subject Matter Expert
Teaching and learning dialog
Graphical User Interface
Disciple Learning Agent Shell
Learning and Knowledge Acquisition Engine
Knowledge
Problem Solving Engine
Problem Solutions
Knowledge Base
Problem Solutions
Intelligent Agent
Problem
Solution and justification
Other users who are not Subject Matter Experts can use the software product to solve problems.

by which a knowledge engineer interacts with a subject matter expert to manually encode his or her knowledge into rules is long, difficult and error-prone. The Disciple approach to rule development does not involve a knowledge engineer. Initially the SME teaches the Disciple agent how to solve problems by showing it each problem reduction step necessary to solve a specific problem, and helping the agent to understand it. From each such problem reduction step, the Disciple agent learns a general problem reduction rule that will allow it to perform similar problem reductions in the future.

As Disciple learns from the SME, their interaction evolves from a teacher-student interaction to an interaction where the SME and the agent collaborate in solving new problems. During this joint problem solving process, Disciple continues to learn new rules from the contributions of the expert, and to refine previously learned rules based on its own problem solving attempts.

Under this agent-building paradigm, knowledge engineers support the SME's creation of a specialized Disciple agent in these ways:

- By customizing the graphical user interface.
- By helping the SMEs to learn how to teach the Disciple agent.
- By facilitating the re-use of ontological knowledge found in established knowledge repositories such as the CYC knowledge base [10].

The ultimate software development framework for the SME-driven Disciple learning agent paradigm is given in Figure 2. This paradigm eliminates much of the error generated by the many different people involved in the typical framework for software development (Figure 1). Who knows the specific problem domain better than the SME – the requirements, the data, and the problem solving techniques?

Part of Disciple's output when it has solved a problem is an explanation of how it derived that solution. Thus other people who are not as familiar with the specific problem domain as the SME can use the trained version of Disciple to solve problems, understand the problem solving reasoning used, and learn themselves.

Because the problem solving approach of a Disciple agent is based on the general problem reduction paradigm, this agent-building methodology is applicable to a wide rage of domains. For instance, we have applied it to develop agents for the following:

- Planning the repair of damaged bridges and roads.
- Critiquing military courses of action.
- Identifying and testing strategic center of gravity candidates in military conflicts.
- Designing configurations of computer systems.
- Generating test questions for assessing a student's higher order thinking skills in history and in statistics, etc.

We consider that the main factors that contribute to the success of the Disciple approach are: 1) the synergistic collaboration between the SME and the Disciple agent in developing the knowledge base, 2) the multi-strategy learning methods of Disciple that are based on a learnable knowledge representation, and 3) extensive knowledge reuse. Nevertheless, this technology is still in its research phase, and significant work remains to be done before it will be available for mainstream DoD use.

Is software developed using an intelligent learning agent such as Disciple a new software design methodology? We claim that it is.

Recorded Successes

The Defense Advanced Research Project Agency (DARPA) has sponsored a sequence of two programs for the development of software systems that will solve knowledge-based problems. The first is the High Performance Knowledge Bases program (HPKB), which ran from fiscal year 1997 to 1999. The second, Rapid Knowledge Formation program (RKF) is currently funded for fiscal years 2000 to 2003. The goal of the HPKB program was to produce the technology needed to enable system developers to rapidly construct large knowledge bases that provide comprehensive coverage of military specific domains of interest, are reusable by multiple applications that use diverse problem solving strategies, and are maintainable in rapidly changing environments. The HPKB program used challenge problems to evaluate the competing software technologies.

In the first set of challenge problems, one of the problems dubbed *the workaround* problem consisted of assessing how rapidly and in what way a military unit might reconstitute or bypass damage to an infrastructure such as a bridge. The Disciple learning agent shell was customized to solve this problem. The SMEs supported by knowledge engineers identified many of the concepts that needed to be represented in Disciple's ontology (e.g. military units, engineering equipment, types of damage, and geographical features of interest). Some of these concepts were imported from existing ontology repositories. The SMEs and the knowledge engineer using Disciple's specialized browsing and editing tools defined others. Knowledge engineers also explained to the SMEs problem reduction modeling. This is an iterative process of stating a problem to be solved, asking a relevant question about solving the problem, and answering the question either conclusively, in which case that portion of the problem is solved, or by creating sub-problems that need further consideration.

We compare the problem-question-answer process to the Army's task-condition-standard testing methodology. Army SMEs had little difficulty in assimilating the idea. Further detail regarding the training inputs and reasoning outputs of Disciple for the workaround problem can be found in [11]. Cohen et al. [12] presents measured results for the first year of HPKB, including the workaround problem where the trained version of Disciple (Disciple-WA) was judged to perform at an expert level. It was also noted that Disciple-WA's knowledge actually doubled during the 17 days of experimental period, thus demonstrating the agent's ability to learn rapidly.

The second part of the HPKB program was based on even more complex challenge problems. The course of action (COA) critiquing problem called for rapid development of knowledge bases containing comprehensive battlefield knowledge. These include terrain characteristics, force structures, troop movements, and tactical strategies to assess COA viability, correctness, and strengths and weaknesses with respect to the principles of war and the tenants of Army operations. To address the challenge problem, the Disciple learning agent shell was customized for the domain by knowledge engineers who imported an initial ontology built by Teknowledge and Cycorp, and by SMEs who further developed the ontology using Disciple's built in tools. The SMEs then taught Disciple how to evaluate a COA, and thus Disciple-COA was developed.

Tecuci et al. [13] presents the results of Disciple-COA evaluation vis-à-vis its competitors. In total scores on the metrics of recall and precision, Disciple-COA outperformed the competition. Furthermore, during the eight days of experimental period, Disciple-COA's knowledge increased by 46 percent, from the equivalent of 6,229 simple axioms to 9,092 simple axioms. This represents a higher daily rate of knowledge acquisition than in the first experiment.

To further test the ability of Disciple-COA to amass knowledge, it was used in a knowledge acquisition experiment at the U. S. Army Battle Command Battle Laboratory (BCBL) in Fort Leavenworth, Kansas. Four SMEs with no prior knowledge of engineering experience received approximately 16 hours of training on Artificial Intelligence and the use of Disciple-COA. Then using the knowledge base containing the previously developed ontology (but with no rules and no significant assistance from knowledge engineers), each SME taught Disciple to critique COAs. This was done with respect to a modeling of the principles of offensive and security that was discussed with them before the experiment.

In about three hours Disciple-COA learned 26 rules. Upon his evaluation, LTC John N. Duquette, chief of the Experimentation Division of BCBL, wrote [14], "The potential use of this tool by domain experts is only limited by their imagination – not their AI programming skills."

The Next Step

DARPA's follow-on RKF program emphasizes the development of knowledge bases by the domain expert. Its central objective is to enable distributed teams of SMEs to enter and modify knowledge directly and easily without the need for prior software development or knowledge engineering experience. The Knowledge Engineering Group (KEG) in the Center for Strategic Leadership at the United States Army War College (USAWC) recommended that the GMU LALAB be given the vexing Center of Gravity Determination problem as the RKF challenge problem for the next step in Disciple's evolution. The KEGs recommendation went to Murray Burke, the RKF program manager at DARPA, and was approved.

Joint Publication 3.0 Doctrine for Joint Operations [15] states, "Identification of enemy centers of gravity require detailed knowledge and understanding of how opponents organize, fight, make decisions, and their physical and psychological strengths and weaknesses. (pp. III-21)." In 1995 KEG researchers and SMEs in various departments at USAWC, along with interested USAWC students and International Fellows taking the elective course "Case Studies in Center of Gravity (COG) Determination," worked to capture the *current* thinking on determining strategic and operational centers of gravity. That thinking was published in a monograph with an accompanying process flow diagram [16]. Attempts by KEG researchers to build an intelligent system that could assist in center of gravity determination and analysis were constrained by the technology and theory of the day.

In the fall of 2000, KEG researchers began ontology development for Disciple-COG [17] and GMU LALAB researchers worked on new tools for Disciple that would aid SMEs in scenario building and domain modeling. In the winter 2001 term, USAWC students in the COG course used these tools to begin the process of building their own agent for the scenario that they have been given to analyze. In the spring 2001 term, another USAWC COG class will use the tools to analyze additional scenarios. Through a process of iterative refinement and with more examples being given to Disciple-COG with each new USAWC COG class, it is expected that the intelligent learning agent will become increasingly adept at determining and analyzing centers of gravity for opposing forces.

Conclusion

Software design and development methodologies have evolved slowly since the early days of computing. Today's methodologies call for extensive user-developer interaction for requirement determination, testing, and acceptance activities. New research and experimental results presented in this paper indicate that this may be about to change.

Supported by DARPA, the U.S. Air Force Office of Scientific Research, and the U.S. Army, the George Mason University Learning Agents Laboratory has taken a novel approach to the creation and use of intelligent agents to solve complex military problems. The goal of this research is to develop methods and tools that allow users with minimal computer skills to easily build, teach, and maintain high performance intelligent software agents. In this approach, the user teaches the agent to solve various problems in a way that resembles how they would teach an apprentice or student – providing examples and explanations, supervising and correcting behavior.

Created this way, the intelligent agent represents a new paradigm for software development. In this paradigm, the inefficient, often ineffective indirect transfer of knowledge and expertise from the end-user through the development team, to the resulting software product, is replaced by direct user development of the system, assisted by intelligent agent technology. This paradigm has already been experimentally shown to be effective in tackling selected military problem-solving at the tactical and operational level, and is now being evaluated at the strategic level.◆

References

1. Khoo, Benjamin, An Integrated Software Design Framework for the Design of Information Systems, Masters Thesis, Andrews University, Michigan, 1996. userpages.umbc.edu/~khoo/survey2.html
2. Sorenson, Reed, A Comparison of Software Development Methodologies, *CrossTalk*, January 1995, pp. 12-18.
3. Humphrey, Wyatt, *A Discipline for Software Engineering,* Addison Wesley, Reading, MA, 1995.
4. Russell, Stuart and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice Hall Upper Saddle River, NJ, 1995.
5. Smith, Reid and Farquhar, Adam, The Road Ahead for Knowledge Management: An AI Perspective, *AI Magazine*, December 2000, pp. 17-40.

6. Tecuci, Gheorghe, *Building Intelligent Agents*, Academic Press, San Diego, CA, 1998.
7. Chaudri, Vinay; Farquhar, Adam; Fikes Richark; Park, Peter; and Rice, James, *OKBC: A Programmatic Foundation for Knowledge Base Interoperability*, Proceedings of the Fifteenth National Conference on Artificial Intelligence, 600-607, AAAI Press, Menlo Park, CA, 1998.
8. Farquhar, Adam; Fikes, Richard; and Rice, James, *The Ontolingua Server: A Tool for Collaborative Ontology Construction*, Proceedings of the Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada, 1996.
9. MacGregor, Robert, Retrospective on LOOM, 1999, www.isi.edu/isd/LOOM/papers/macgregor/LoomRetrospective.html
10. Lenat, Douglas, CYC: A Large-scale Investment in Knowledge Infrastructure, *Communications of the ACM*, November 1995, pp. 33-38.
11. Tecuci, Gheorghe; Boicu, Mihai; Wright, Kathryn; Won Lee, Seok; Marcu, Dorin; and Bowman, Michael, A Tutoring Based Approach to the Development of Intelligent Agents, *Intelligent Systems and Interfaces*, Kluwer Academic Publishers, Boston, Mass., 2000.
12. Cohen, Paul; Schrag, Robert; Jones, Eric; Pease, Adam; Lin, Albert; Starr, Barbara; Gunning, David; and Burke, Murray, The DARPA High-Performance Knowledge Bases Project, *AI Magazine*, December 1998, pp. 25-49.
13. Tecuci, Gheorghe; Boicu, Mihai; Bowman, Michael; Marcu, Dorin; preface by Burke, Murray, An Innovative Application from the DARPA Knowledge Bases Programs: Rapid Development of a Course of Action Critiquer, *AI Magazine*, June 2001.
14. Bowman, Michael; Tecuci, Gheorghe; and Boicu, Mihai, Intelligent Agents, Tools for the Command Post and Commander, to appear in *Acquisition, Logistics, and Technology*, 2001.
15. Joint Publication 3.0, Doctrine for Joint Operations, Joint Chiefs of Staff, Washington, DC, 1995.
16. Giles, Phillip and Galvin, Thomas, Center of Gravity: Determination, Analysis, and Application, Center for Strategic Leadership, United States Army War College, Carlisle Barracks, Pa., 1996.
17. Bowman, Michael; Lopez, Antonio; and Tecuci, Gheorghe, Ontology Development for Military Applications, Proceedings of the Thirty-Ninth Annual ACM Southeast Conference, March 2001.

Acknowledgements

Research at the GMU Learning Agents Laboratory is sponsored by the DARPA, Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-00-2-0546, by the Air Force Office of Scientific Research (AFOSR) under grant no. F49620-00-1-0072, and by the U.S. Army. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFOSR, the Air Force Research Laboratory, the U.S. Army or the U.S. government. Mihai Boicu, Dorin Marcu, Bogdan Stanescu, Catalin Balan, Elena Popovici, Cristina Cascaval, and other members of the LALAB contributed to successive versions of Disciple. The CrossTalk Editorial Board provided insightful comments and suggestions that helped us improve this paper.

About the Authors

LTC Michael Bowman is a student at the U.S. Army War College and a Ph.D. candidate at George Mason University. He was the Army product manager for Communications and Intelligence Support Systems, and has had a variety of acquisition, automation, and tactical assignments at the Defense Intelligence Agency, the U.S. Military Academy, and in several Army field artillery battalions. He received a bachelor's degree from Ouachita Baptist University and a master's degree from the Naval Postgraduate School.

Knowledge Engineering Group
Center for Strategic Leadership
United States Army War College
Carlisle Barracks, PA 17013
Voice: 717-245-3252
Fax: 717-245-4600
E-mail: michael.bowman@carlisle.army.mil

Antonio M. Lopez Jr., Ph.D., holds the chair of Artificial Intelligence in the Knowledge Engineering Group of the Center for Strategic Leadership at the U. S. Army War College. He also holds the Conrad N. Hilton Endowed Chair in Computer Science at Xavier University of Louisiana. He is a colonel (retired) in the U. S. Army Reserve having held assigned positions with the 377th Theater Support Command as assistant chief of staff G6, G3, and chief of staff.

Knowledge Engineering Group
Center for Strategic Leadership
United States Army War College
Carlisle Barracks, PA 17013
Voice: 717-245-3251
Fax: 717-245-4600
E-mail: lopezt@csl.carlisle.army.mil

MAJ James Donlon is the director of the Knowledge Engineering Group at the U. S. Army War College. He conducts applications engineering, education, and applied artificial intelligence research as an Army systems engineer. His Army background is as an infantry officer. He received a bachelor's degree from the University of Delaware and a master's degree from Northwestern University.

Knowledge Engineering Group
Center for Strategic Leadership
United States Army War College
Carlisle Barracks, PA 17013
Voice: 717-245-3265
Fax: 717-245-4600
E-mail: donlonj@csl.carlisle.army.mil

Gheorghe Tecuci, Ph.D., is professor of computer science, director of the Learning Agents Laboratory at George Mason University, and member of the Romanian Academy. He received his doctorate and master's degrees in computer science from the Polytechnic University of Bucharest, and a doctorate in computer science from the University of Paris-South. Tecuci has published more than 100 scientific papers and five books, most of them in the artificial intelligence areas of intelligent agents, machine learning, and knowledge acquisition.

Learning Agents Laboratory
Department of Computer Science
George Mason University
4400 University Drive
Fairfax, VA 22030
Voice: 703-993-1722
Fax: 703-993-1710
E-mail: tecuci@gmu.edu

"Computation offers a new means of describing and investigating scientific and mathematical systems. Simulation by computer may be the only way to predict how certain complicated systems evolve."

Stephen Wolfram

Extreme Methodologies for an Extreme World

Theron Leishman
Software Technology Support Center

The world we live in today demands greater availability of decision-making information. The use of cell phones, hand held computers, online banking, and internet stock trading are only a few examples of this demand for timely information. Information Technology organizations are feverishly scrambling in an attempt to gain ground on the ever increasing demand for their services. This article takes a glimpse into the software development methodologies that are being applied in an attempt to catch up in the rapidly changing world in which we live. A brief insight into traditional development methods will be followed by an analysis of the principles underlying the new, less traditional methods. These newer, extreme methodologies share many common characteristics, which will be identified and discussed. An analysis will also be made of the ability of extreme methodologies to meet the rigor demanded by the Software Capability Maturity Model.

While enjoying a day on the slopes at one of the local Utah ski resorts, I had the chance to ski on the course that has been developed for the downhill ski races in the upcoming 2002 Winter Olympic Games. As I spent the day in my feeble attempt to navigate the course, I could only image the adrenaline rush experienced by those who will be competing in a sport in which skiers race at speeds in excess of 70 miles per hour.

Another winter sport not as well known is speed skiing. Called the fastest non-motorized sport on earth, skiers plunge down a steep ramp of ice propelling themselves from 0 to 154 mph in about 10 seconds. Is this insanity? Why would anyone in his or her right mind participate in this sport?

Growing numbers of people are enjoying high risk, *extreme* activities. Everything in our world today seems to be following this extreme theme. Daily we are bombarded with the demand for things to be bigger, better, and faster! What was once viewed as adequate is now considered substandard and lack-luster.

Business and government organizations are not immune from this phenomenon. Developments in recent years have resulted in an environment where information is needed now! Decisions must be made quickly! Fortunes can be made, governments can fall, and wars can be won or lost based on the availability of information to make sound decisions.

This rapid pace has had substantial impact on computer resources. The boom in web technologies is an example of this demand for information. For companies to remain competitive, the amount of information available on the web has skyrocketed. Information technology organizations have had to scramble to respond to this demand. New techniques are evolving in an attempt to keep pace with growing demand. Into this environment has sprung *extreme software development methods*.

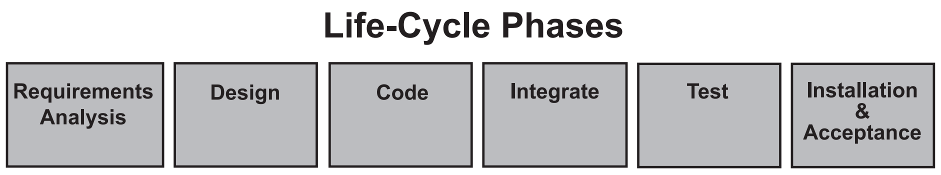


Figure 1: *Traditional Software Development Phases*

Is an extreme programmer a wild and crazy thrill seeker, pounding out code with no concern for his own safety or the security and stability of the organization? What about the project lead that leaps from tall buildings in a single bound with no thought of standards, best practices, or the future maintainability of the application? These are the visions that come to mind when thinking of extreme development methods. However, before validating or dispelling this stereotype, a review of traditional development methods comes first.

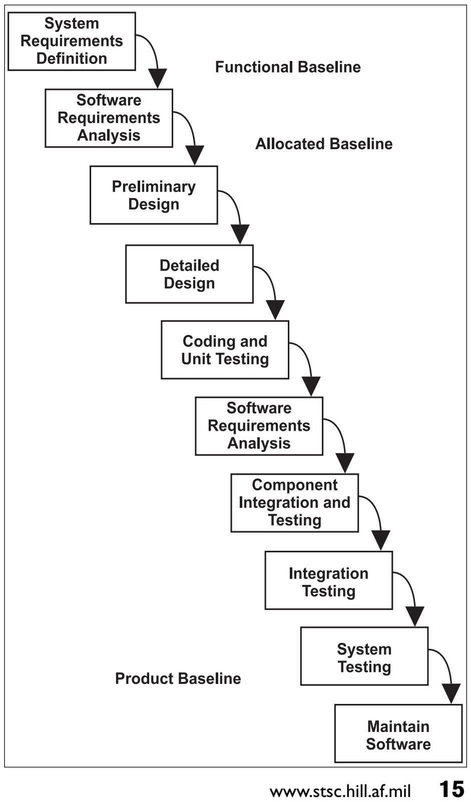
Traditional Development Methods

Traditional software development methodologies were developed to standardize the procedures used to develop and maintain software. These methodologies follow the general phases outlined in Figure 1. These methodologies are usually based on one of the following three basic models:

The Waterfall Model: The Waterfall model for software development is a step-by-step process. The requirements for one phase are completed prior to the next phase beginning. This is a onetime through approach. The phases build upon each other toward the completion of the development effort. Figure 2 illustrates the sequential nature of a typical waterfall methodology.

The Incremental Model: Using the incremental approach, user needs are determined and system requirements are defined. Then the rest of the development is performed in a sequence of builds. The first build incorporates part of the planned

Figure 2: *Waterfall Model*



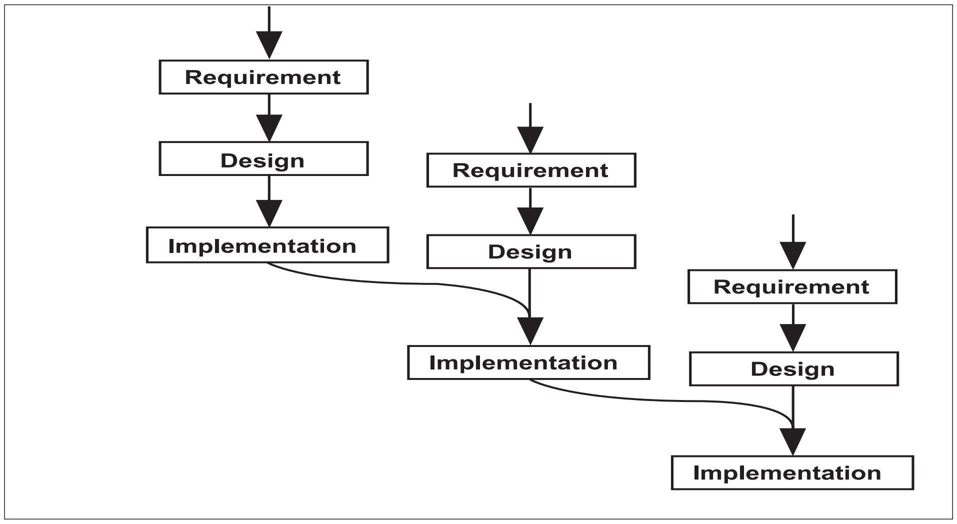


Figure 3: *Incremental Model*

capabilities. The next build adds additional capability. This incremental process continues until the system is complete. An example of the incremental model is presented in Figure 3.

The Evolutionary Model: An evolutionary, or spiral, model also uses a system of builds as in the incremental model. The variation comes in the acknowledgement up front that the user needs and requirements are not fully understood at the inception of the project. Using this strategy, the user needs and system requirements are partially defined at the inception of the project. They are then refined in each succeeding build. A pictorial representation of the evolutionary model is in Figure 4.

Extreme Development Methodologies

Extreme methodologies take a different approach from traditional software development methodologies. These methodologies accept the notion that, change happens during all phases of the development process. Further, it is anticipated that the system users may not know exactly what they want the final product of the development effort to be.

A number of non-traditional methodologies have evolved in recent years. Methodologies such as Extreme Programming (XP), Adaptive Software Development (ASD), SCRUM, and Crystal Light are a few of the methodologies attempting to achieve notoriety in these extreme times.

Extreme methodologies tend to be tailored after the evolutionary or spiral development model. They use phases like inception, elaboration, construction, and transition or speculation, collaboration, and learning to streamline the development approach of traditional methodologies. These methods attempt to satisfy customer requirements by speeding development and delivering useful products in rapid iterations. Full functionality is achieved over time, while partial functionality is achieved quickly. Figure 5 presents a comparison of the phases of traditional life-cycle methodologies to those of typical extreme methodologies.

Each of the extreme methods have developed it's own approach to conquering the brave new world of speedy development. Underlying each of these methods is a foundation of basic characteristics common to each. A summary of these similarities follows.

An Iterative Approach: These methods are designed for use in uncertain conditions. They accept the fact that change is inevitable, therefore the system evolves through various iterations. Requirements are identified and incorporated into each iteration of the system. The application is presented to the customer in releases providing increased value to the customer with each iteration until the desired system evolves.

Risk Driven: With each development iteration, risk is evaluated and acts as a driving factor in the evolution of the system. If additional requested functionality causes system risk, or increases the complexity of the application to the point of impacting developmental cost or schedule, the risk is identified and dealt with as it arises.

Studies of software development projects have shown that projects are often over budget, behind schedule, and do not provide the desired functionality. All too often these projects are never used in a production environment. The risk of this type of waste is greatly reduced by extreme methodologies. As iterations of the prod-

Figure 4: *Evolutionary/Spiral Model*

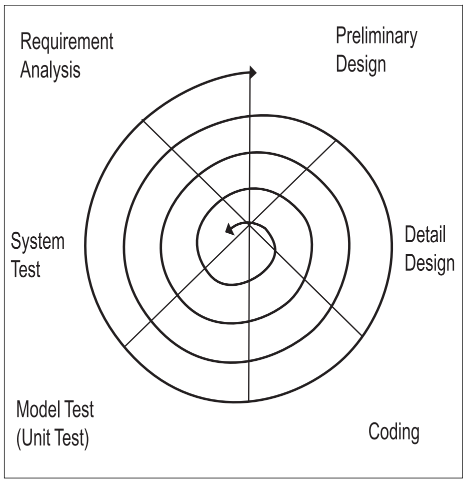
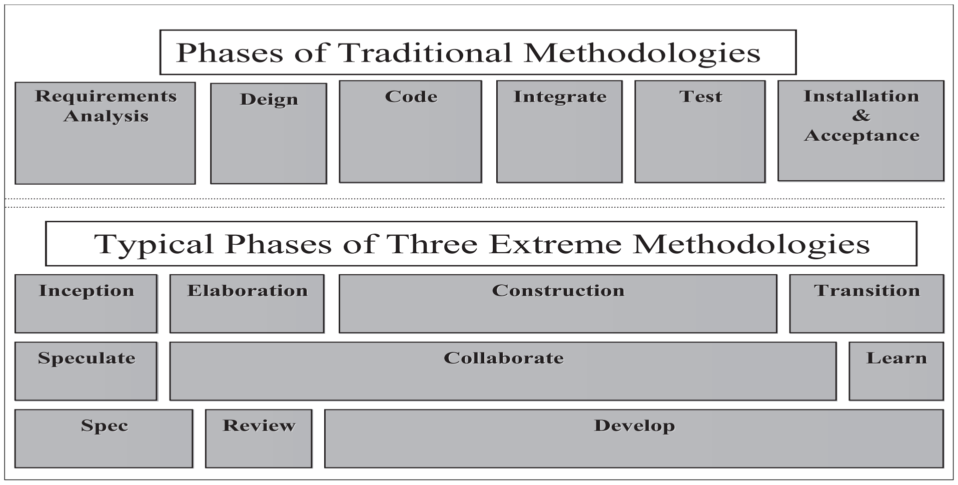


Figure 5: *Traditional Life-Cycle vs. Extreme Methodologies*



uct are completed, design errors and omissions are identified and corrected. Large investments in applications that provide little or no value can be eliminated. Systems evolve that either meet user requirements, or the development is terminated prior to large investments being made in applications that do not warrant the investment.

Results Oriented: Extreme methodologies are intended to achieve results that are often not possible using other approaches. These methodologies portray a *movers and shakers* attitude by focusing on results. These methods build in processes intended to clear the tracks for the express development train to move at extreme speeds. Extreme methods get results in these ways:

- Removing impediments to progress.
- Assuring prompt and timely decision making.
- Isolating the project and team members from irrelevant issues.
- Utilizing all resources and expertise required to achieve success.
- Rewarding individuals and teams involved with successful projects as the success occurs.
- Focusing the attention of team members on the extreme project and nothing else.

Use Sound Development Processes

In an article published in *Software Testing & Quality Engineering*, Jim Highsmith stated that the Old World was one of optimization where efficiency, predictability, and control dominated. The New World is one of adaptation in which change, improvisation, and innovation rule. This dichotomy – optimization versus adaptation – provides a distinct way of viewing the future of software project management.

A study of extreme methodologies soon leads to the determination that these methods do not promote out-of-control, free-for-all programming. These methodologies recognize value in the lessons learned from the history of software development. Although they emphasize the need to rethink the traditional methods and eliminate processes that are not value added, they recognize the need for sound development processes. For example, the extreme methodologies studied encourage the following:

- Testing at the unit and functional levels.
- Project planning.
- Developing and adhering to sound development standards.
- Using pair programming.
- Developing and following coding standards.
- Releasing software in frequent intervals.
- Establishing configuration management process.
- Providing onsite dedicated customer project support.

Emphasize Collaboration

Extreme methodologies all share the idea of collaboration. The idea of empowered, high-performance teams is key to the success of projects using extreme methodologies. Team members must have complementary skills and be dedicated to the common purpose, performance goals, and approach to which they hold themselves mutually accountable. These empowered teams include all required technical disciplines and user involvement to allow decisions to be made as problems arise.

Having the right people dedicated to a project will ensure that stakeholder and developer alike have agreement on issues from primary requirements through final implementation. Issues of scope, cost, schedule, priorities, risk, and trade-off's between these are well understood by all parties. Surprises are eliminated and successful projects are the result.

A recent study on the cost and benefit of pair programming revealed that a pair of programmers working on a project took 15 percent more time to complete the programming effort than a single programmer working alone. This cost was easily offset by the resulting 15 percent reduction in coding defects. History has shown that the cost to correct defects is exponentially greater than the cost of original coding [1].

A team of programmers developing simulator software for the Israeli Air Force also realized results similar to those identified in the above study. By using pair programming, and being located in the same facility as their customer, the team has been able to greatly reduce the time between software releases. The team is seeing great benefits by using XP principles in their development efforts [2].

It has been said that the number one killer of software development projects is time. Collaboration allows extreme projects to remove barriers and eliminate delays that kill projects.

How Extreme is Extreme?

Upon consideration of the so-called extreme methodologies, the question arises, "What is so extreme?" The characteristics of the extreme methods that appear to make these approaches appealing and successful are not glitzy new tools or earth shaking new discoveries. They are things that contribute to the success of any project following any sound methodology.

Ward Cunningham, one of the fathers of XP has indicated that, "Extreme Programming is a lot of simple little things. It's lots of things that have been done before, some of which have even been discredited. We have reconstructed a collection of practices that support each other in a way that is startling [3]."

The Capability Maturity Model

The Software Engineering Institute developed the Capability Maturity Model® (CMM®) for software, which organizations can use as a base to measure their software development and maintenance processes. It provides a standard for software process improvement to assist organizations desiring to make conscious improvement in their software development efforts. The CMM lays out a path to lead an organization from an ad hoc, immature development organization to a mature, disciplined organization. The CMM contains five levels of maturity. It covers practices for planning, engineering, and managing software development and maintenance [4].

The Key Process Areas (KPAs) of CMM Level 2 and Level 3 cover many areas of sound software development methodologies. Below is a summary of these levels along with a list of the KPAs for each level.

Level II – The focus of CMM Level 2 is on software project management. Software project management processes are to be documented and followed.

Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark office.

Organizational policies guide projects in establishing management processes. Successful practices developed can be repeated. These are the key process areas of Level 2:

- Requirements Management.
- Software Project Planning.
- Software Project Tracking and Oversight.
- Software Subcontract Management.
- Software Quality Assurance.
- Software Configuration Management.

Level III – The focus of CMM Level 3 is the software engineering process. At Level 3 emphasis moves toward the organization. The organization has processes in place that empower the individuals doing software development. Processes are defined, documented, and understood by individuals within the organization. These are the key process areas of Level 3:

- Organization Process Focus.
- Organization Process Definition.
- Training Program.
- Integrated Software Management.
- Software Product Engineering.
- Intergroup Coordination.
- Peer Reviews.

CMM and Extreme Methodologies

The CMM is intentionally not prescriptive in nature. An organization choosing to follow the framework of the CMM should develop policies and procedures that make sense for its environment. The CMM should be used as a model of the best practice in software development and maintenance.

An established software development methodology is an essential element of a mature software development organization. In addition, standardized development methodologies span many of the KPAs identified within the CMM. However, the CMM does not advocate one development methodology over another. The model does recommend that a sound methodology be followed.

An extreme development methodology can provide the elements necessary for a development organization to be successful in their development efforts. As noted earlier in this article, extreme methodologies advocate sound practices that are defined in the CMM. Principles of planning, requirements management, quality assurance, and project tracking and oversight are all essential elements of a sound extreme methodology. These are all KPAs within the CMM. They are practices essential to the ongoing success of a development and maintenance organization.

Conclusion

The demands of the world we live in require more information to be available faster than any time in history. To remain competitive, an organization must be able to respond to this demand. Extreme software development methodologies have demonstrated their ability to help information technology organizations respond to this growing demand for their services.

These methods are not simple solutions to an age-old problem. Just as the participants of extreme sports have tools and equipment to allow them to reduce the risk of participation in their chosen sport, so also should an extreme methodology reduce risk to the organization. A mature development organization will have sound policies and procedures in place to guide the selection and tailoring of appropriate methodologies.

It is important to remember that not all projects are extreme projects. However, when a project is determined to be extreme, a sound extreme methodology will allow the project to accomplish it's objective while not exposing the organization to undue risk.◆

References

1. Byer, Dr. Sam and Highsmith, Jim, RADical Software Development, *American Prog rammer Magazine*, June 1994.
2. Cockburn, Alistair and Williams, Laurie, The Costs and Benefits of Pair Programming, Humans and Technology Technical Report 2000.01, January 2000.
3. Waters, John K., Extreme Method Simplifies Development Puzzle, *Application Development Trends*, July 2000.
4. Software Engineering Institute, *The Capability Maturity Model Guidelines for Improving the Software Process*, Boston, Addison Wesley, 1994.

Additional Reading

1. Beck, Kent, *Extreme Programming Explained: Embrace Change,* Boston, Addison Wesley, 2000.
2. Experience With XP, c2.com/cgi/wiki?ExperienceWithXP
3. Highsmith, Jim, Retiring Lifecycle Dinosaurs, *Software Testing & Quality Engineering*, July/August 2000.
4. Koerner, Brendan I., Extreme, *U.S. News*, June 30, 1997.
5. Rational Software Corporation, Rational Unified Process: Best Practices for Software Development Teams, White Paper TP-026A Rev 11/98.

About the Author

Theron R. Leishman is a consultant currently under contract in the Software Technology Support Center at Hill AFB, Utah, which provides consulting services to Air Force and other Department of Defense (DoD) and government agencies. He began his career as an Electronic Data Processing auditor assessing software development, software project management, computer security and compliance to government standards and regulations. Leishman has a wide range of experience in various aspects of software design, development, project management, and process improvement. He has successfully managed projects of various size and complexity for the DoD, aerospace, manufacturing, health care, gas and oil, higher education, and various other industries. Leishman is currently employed by TRW.

Software Technology Support Center
7278 4th Street
Bldg. 100 G19
Hill AFB, Utah 84056
Voice:801-775-5738
Fax:801-777-8069
E-mail:Theron.Leishman@hill.af.mil

"Computers are to computing as instruments are to music. Software is the score whose interpretations amplifies our reach and lifts our spirits. Leonardo da Vinci called music the shaping of the invisible, and his phrase is even more apt as a description of software"

Alan C. Kay

The Quality of Requirements in Extreme Programming

Richard Duncan
Mississippi State University

Extreme Programming (XP) is a software process methodology that nominates writing code as the key activity throughout the development process. While at first glance this sounds chaotic, a disciplined group utilizing XP performs sufficient requirements engineering. This paper describes and evaluates the quality of requirements generated by an ideal group using XP and discusses how the XP process can assist or hinder proper requirements engineering.

Extreme Programming (XP) is a hot, new software process methodology for medium to small sized organizations. It is designed with requirements drift as a fundamental occurrence to be embraced, rather than dealing with it as a necessary evil. XP nominates coding as the key activity throughout the development process, yet the methodology is based on economics [1].

Barry Boehm presented that the cost of change grows exponentially as the project progresses through its lifecycle [2], Stuart Faulk reiterates this by stating that the relative repair cost is 200 times greater in the maintenance phase than if it is caught in the requirements phase [3]. XP challenges that this is no longer the case. While it is more expensive to modify code than to modify a prose description, with modern languages and development techniques it is not an exponential increase.

Instead, Beck asserts that the cost for change levels out. Rather than spend extra effort in the requirements analysis phase to nail down all requirements (some of which will become obsolete through requirements drift anyway), accept that changes due to incomplete requirements will be dealt with later. XP assumes that lost resources in rework will be less than the lost resources in analyzing or developing to incomplete requirements [1].

The primary vehicle for requirements elicitation in XP is adding a member of the customer's organization to the team. This customer representative works full time with the team, writing stories – similar to Universal Markup Language (UML) Use Cases – developing system acceptance tests, and prioritizing requirements [4]. The specification is not a single monolithic document; instead, it is a collection of user stories, the acceptance tests written by the customer, and the unit tests written for each module. Since the customer is present throughout the development, that customer can be considered part of the specification since he or she is available to answer questions and clear up ambiguity.

The XP life cycle is evolutionary in nature, but the increments are made as small as possible. This allows the customer (and management) to see concrete progress throughout the development cycle and to respond to requirements changes faster. There is less work involved in each release, therefore the time-consuming stages of stabilization before releases take less time. With a longer iteration time it may take a year to incorporate a new idea: with XP this can happen in less than a week [1].

A fundamental of XP is testing. The customer specifies system tests; the developers write unit tests. This test code serves as part of the requirements definition – a coded test case is an unambiguous medium in which to record a requirement. XP calls for the test cases to be written first, and then the simplest amount of code to be written to specify the test case. This means that the test cases will exercise all relevant functionality of the system, and irrelevant functionality should not make it into the system [1].

This paper describes and evaluates the requirements engineering processes associated with the XP paradigm.

The XP Requirements Engineering Process

Harwell et al. break requirements into two types – product parameters and program parameters. A product parameter applies to the product under development, while a program parameter deals with the managerial efforts that enable development to take place [5]. The customer who becomes a member of the XP team defines both product and program parameters. The product parameters are defined through stories and acceptance tests, while the program parameters are dealt with in release and iteration planning.

The product parameters are chiefly communicated through *stories*. These stories are similar to Use Cases defined in UML, but are much simpler in scope [4]. Developing a comprehensive written specification is a very costly process, so XP uses a less formal approach. The requirements need not be written to answer every possible question, since the customer will always be there to answer questions as they come up. This technique would quickly spiral out of control for a large development effort, but for small- to medium-sized teams (teams of fewer than 20 people are most often reported) it can offer a substantial cost savings. It should be noted, however, that an inexperienced customer representative would jeopardize this property.

The programmers then take each story and estimate how long they think it will take to implement it. Scope is controlled at this point – if a programmer thinks that the story, in isolation, will take more than two weeks to implement, the customer is asked to split the story. If the programmers do not understand the story they can always interact directly with the customer. Once the stories are estimated, the customer selects which stories will be implemented for the upcoming release, thereby driving development from business interests. At each release, the customer can evaluate if the next release will bring business value to the organization [1].

Each story to be implemented is broken up into tasks. A pair of programmers will work to solve one task at a time. The first step in solving a task (after understanding, of course) is to write a test case for it. The test cases will define exactly what needs to be coded for this task. Once the test cases pass, the coding is complete [1]. Thus the unit tests may be considered a form of requirements as well. Every test (across the entire system) must pass before new code may be integrated, so these unit-test requirements are persistent. This is not to say that simple unit testing counts as an executable specification – but XP's test-driven software development does record the specific requirements of each task into test cases.

The final specification medium for product requirements is the customer acceptance tests. The customer selects scenarios to test when a user story has been correctly implemented. These are black-box system tests, and it is the customer's responsibility to ensure that the scenarios are complete and that they sufficiently exercise the system [6]. These acceptance tests serve as an unambiguous determiner as to when the code meets the customer's expectations.

How XP Rates

The XP requirements engineering process can be analyzed by considering the 24 quality attributes for software requirements specification (SRS) proposed by [7]. Davis et al. propose that a quality SRS is one that exhibits the 24 attributes listed in Table 1. Rather than applying these metrics to a given document, they are used here to measure the requirements that theoretically come out of the XP process. Of course, a quality SRS is mostly dependent on the discipline used by the people associated with the project, but specific features of XP can influence the quality of a SRS.

A specification created with XP would appear to score very well across most of these attributes, but fare poorly on others. Those qualities with a "+" symbol indicate that the subsequent paragraphs argue the XP process can lead to an improvement in the area: a "-" that XP detracts from the quality. The "+/-" annotation indicates that XP partially helps and partially harms a specification in achieving the quality. Many of the qualities are not addressed by XP and are hence annotated with a "?," for these qualities a group's organization, discipline, and specific project needs will decide. It should be noted that to religiously follow XP requires a great deal of discipline: This discipline should be expected to carry over into the other qualities. Following is a look at some of the quality attributes.

Unambiguous, Correct, and Understandable: Since the customer is present, ambiguity and problems understanding the requirements are generally minimal and easily solvable [1] . Requirements are correct if and only if each represents an actual requirement of the system to be built. Since the customer writes the stories from business interests, the requirements should all be correct. With so much responsibility and freedom, clearly the selection of an appropriate customer representative is crucial to the success of the project. Even if the customer does not know exactly what he or she desires at the start of the project, the evolutionary nature of XP development leads to a system more in line with the customer's needs.

Modifiable: The XP lifecycle allows changes to the requirements specification at nearly any point in system development. The specification exists as a collection of user stories, so the customer can switch out one future story for another with little impact on existing work. Since the planning, tests, and integration are all performed incrementally, XP should receive highest marks in modifiability. Of course, work may be lost in this changeover, but with XP the programmers should be able to estimate how much a change will cost.

Unambiguous, Verifiable: Since the customer writes acceptance tests (with the assistance of programmers), it could be argued that the functional specification is recorded in an unambiguous format. Furthermore, the first activity performed by a programming pair to solve a task is to write test cases for it. These test cases become a permanent part of the specification/test suite. Customers (with the help of the XP coach) will also make sure that the specification is verifiable, since they know that they will have to write test cases for it.

Annotated by Relative Importance: The customer defines which user stories they wish implemented in each release. Hence, each requirement is annotated by relative importance at this time – the customer should ask for the highest-priority stories to be implemented first and programmers are never left guessing priorities.

Achievable: Since each release provides some business value, a portion of the system found to be unachievable should not leave the customer with a very expensive yet unusable piece of technology. If the high-risk piece is important, it will be implemented first, in which case the unachievable component should be found quickly and the project aborted relatively inexpensively. If it is less important, then the system may be delivered in useful form without it.

Design Independent: Design independence is a classic goal for requirements, but today's object-oriented development methods recognize that design independent requirements are often impractical. Portions of the requirements (such as the user stories) can be very design independent, but the unit tests that are archived as part of the requirements and used to cross-check new modules may depend heavily on the actual system.

Electronically Stored: XP calls for the stories to be written on index cards, so this portion of the requirements is not electronically stored. While the stories could

Table 1: *The 24 Quality Attributes [7]*

1. Unambiguous	+	13. Electronically Scored	+/-
2. Complete	-	14. Executable/Interpretable	+/-
3. Correct	+	15. Annotated by Relative Importance	+
4. Understandable	+	16. Annotated by Relative Stability	?
5. Verifiable	+	17. Annotated by Version	+
6. Internally Consistent	+/-	18. Not Redundant	-
7. Externally Consistent	+/-	19. At the Right Level of Detail	?
8. Achievable	+	20. Precise	?
9. Concise	+	21. Reusable	?
10. Design Independent	+/-	22. Traced	?
11. Traceable	?	23. Organized	?
12. Modifiable	+	24. Cross-Referenced	?

+/- indicates XP both assists and degrades. + indicates XP may assist in this area.
? indicates XP has little bearing on the area. - indicates XP degrades this area.

be placed in a word processor, Jeffries et al. assert that handwritten index cards produce less feelings of permanence and allow the customer to more freely change the system [4]. The customer is also available as a requirements resource, obviously not electronically stored. However, the requirements are written on individual cards so modifications can often be localized to a single card if rewriting is necessary. Furthermore, the customer codifies the system requirements with acceptance tests, so it could be argued that the most important part of the specification is stored.

Complete, Concise: XP stresses programming as the most important development activity, hence little effort is spent on creating documents, therefore the specification is very concise. The cost may be a lack of completeness, however. Since little up front analysis takes place, there may very well be holes in the system. Yet the customer drives what functionality is implemented and in what order, so true functionality should not be left out. Furthermore, since the XP process accommodates change, it should be possible to compensate for these holes later in the development lifecycle.

Security Assurances

Since the XP development methodology does not progress from a verified requirements document, how might a system developed with XP rate on a security evaluation? The Common Criteria has seven evaluation assurance levels (EAL1-EAL7). For EAL5 and above the Common Evaluation Methodology calls for the system to be semi-formally designed and tested [8]. This leaves two questions to be addressed. First, can a project use formal methods with XP? Second, without formal methods, how trusted can a system developed under XP be?

The XP process screams informality in many respects. The name alone conjures images of snowboarders with laptops, and even the books about XP are written in a conversational tone. Nevertheless, what would happen if the customer writes stories and they are annotated with a formal specification? Clearly, this would entail a large cost in training personnel, writing the specifications, and verifying the specifications. This also reduces the agility of the XP product – since more money is spent on specification, the cost of change will increase. But if each story were rewritten in a formal notation it would be possible to formally verify the specification and design.

Formal methods aside, the way an XP project progresses does offer many assurances of trust. First, all code is written directly from the user stories (the specification). All functionality is tested in the unit tests and all integrated code is required to pass all tests all the time. While testing does not guarantee the absence of errors, many security holes come from poorly tested software. Hence, the test-oriented nature of XP may be a great step forward.

A strong security feature of XP is pair programming. The observer in a pair constantly evaluates the code being written by his or her partner. This programmer can help reduce the probability of coding errors that might later be exploited (e.g., buffer overruns). XP also adds counterbalances to reduce the impact of a single malicious coder (either in a truly malevolent sense or inadvertently opening holes as Easter Eggs[2] side effects) through the pairing process. Rather than just inserting code into the system, one programmer would have to convince the other of a rationale for why the code was being inserted. Due to collective code ownership, it is entirely possible that the next pair in the course of re-factoring would catch malicious code. Pair programming and collective code ownership add further assurance that the code is written exactly to the specification.

Conclusions

XP performs requirements engineering throughout the life cycle in small informal stages. The customer joins the development team full time to write user stories, develop system acceptance tests, set priorities, and answer questions about the requirements. The stories are simpler in scope to use cases because the customer need not answer every conceivable question. The informal stories are then translated into unit and system acceptance tests, which have some properties of an executable specification.

Of the 24 quality attributes of a software specification, the XP process leads to higher points in nine attributes and lowers the score in two. The most noteworthy gains are in ambiguity and understandability, since the customer is always present to answer questions and clear up problems. Furthermore, since the customer is also responsible for developing test scenarios he or she will create more verifiable requirements. The discipline enforced by the XP process should also carry over into other areas of requirements engineering.◆

Notes

1. XP has been used on several projects. Beck mentions a campaign management database, a large-scale payroll system, a cost analysis system, and a shipping and tariff calculation system in [9]. Yet there is little objective data available for analysis at this time. More data should be made available through the upcoming XP Universe Conference, Raleigh, NC, July 2001, www. xpuni verse.com
2. An unsolicited, undocumented piece of code a programmer inserts into software, generally for his or her own amusement.

References

1.Beck, Kent, *Extreme Programming Explained: Embrace Change*, Boston, Addison Wesley, 2000.

2.Boehm, Dr. Barry, *Software Engineering Economics*, Prentice Hall, 1981.

3.Faulk, Stuart, Software Requirements: A Tutorial, *Software Engineering*, IEEE 1996, pp. 82-103.

4.Jeffries, Ron; Anderson, Ann; and Hendrickson, Chet, *Extreme Programming Installed*, Boston, Addison Wesley, 2001.

5.Harwell, Richard; Aslaksen, Erik; Hooks, Ivy; Mengot, Roy; and Ptack, Ken, What is a Requirement? Proceedings, Third Annual International Symposium National Council Systems Engineering, 1993, pp. 17-24.

6.Wells, J. Donovan, Extreme Programming: A Gentle Introduction, www.ExtremeProgramming.org 2.75

Continued on page 31

The Software Engineer: Skills for Change

Dr. Stephen E. Cross and Dr. Caroline P. Graettinger
Software Engineering Institute (SEI)[1]

The rapid pace of change in software engineering means that software-intensive organizations must develop a core competency for proactive change management. While most software engineers are concerned with doing their jobs and getting a product out, others need to take on the role of technology change-agents for their organizations. We suggest that a career path in change management will be a critical need and opportunity for the 21st century software engineer.

During the past 15 years, the Defense Science Board (DSB, www.acq.osd.mil/dsb) has made more than 130 recommendations intended to improve the ability of software engineering organizations to produce high quality software on time and at cost, yet these organizations continue to find it difficult to make these changes. If improvement were simply a matter of purchasing a new software tool, change would not be so difficult. But to realize the improvements called out in the DSB reports and others like them requires changes in the day-to-day practices of software engineers and their managers. Achieving these kinds of changes requires more skills than the ability to purchase a new tool – skills for change management become important.

Unfortunately, too few of the software-intensive organizations are proactive about managing change. A recent article by Boehm and Basili shows that poor software engineering practices are still a major contributor to software defects [1]. While some organizations adopt new and better software engineering practices and technologies because they recognize their strategic value, most organizations are more reactive than proactive.

Reactive organizations are at the mercy of change led by others, and generally attempt radical improvements in their software engineering disciplines only after customer pressures or disaster. By then, this catch-up game is costly and is often implemented by pick-up teams of individuals who may be well intentioned, but lack the change management skills to be successful. The most common result is money spent with little change to show for it.

Change: What Past Studies Say

Let us look in more detail at some of the past recommendations for change in the software engineering community. The DSB conducted task forces on defense software in 1987, 1994, and 2000 [2, 3, 4]. Each report advocated management and technical practices aimed at helping organizations improve the quality of their software as well as their productivity, cycle time, and cost effectiveness. For example, each report offered recommendations in the areas of acquisition policy, contracting, work force, and technology to encourage the Department of Defense (DoD) and its industry base to adopt spiral development and evolutionary acquisition techniques, and to exploit evolving commercial technology. Each report observed with concern the shortage of software engineers in the government and defense industry base and the need for better training and professional development.

However, each report proposed slightly different approaches to realize the recommendations, depending on the technology, organizational trends, and opportunities prevalent at that time. For example, the 1987 report stressed the role of high-order languages, most notably Ada and computer-aided software engineering tools in support of Ada implementations[2].

The 1994 report encouraged use of commercial products and processes and a greater focus on the design phase, most notably on software architecture and product line practice. Both were considered fundamental to a successful strategy for migrating software engineering from a line-by-line endeavor to one in which systems were assembled from commercial components.

The 2000 report provided six recommendations:

1. Stress past performance and process maturity.
2. Restructure contract incentives.
3. Collect, disseminate, and employ best practices.
4. Initiate independent expert reviews.
5. Improve the software skills of acquisition and program management.
6. Strengthen and stabilize the technology base.

In essence, the 2000 DSB study summarized the important recommendations from past studies. These recommendations say that change is required just to gain the advantages of what is known to be effective software engineering practices.

However, there are other sources of change as well. DoD software requirements are expanding, including demand for increased functionality and flexibility of software within any one system. Human resources shortages are continuing, as observed in studies during the past 15 years, with severe shortages in the government and the defense industry base[3]; productivity improvements are essential. Lastly, security issues are now a major concern. These trends all suggest additional changes that organizations and their software engineers will need to deal with in their day-to-day practices.

Skills for Change

What are the implications for the 21st century software engineer? While most software engineers are concerned with doing their job and getting a product out, others need to take on the role of software technology change-agents. Such engineers need to develop the following skills:

- Identifying and evaluating new software engineering methods and tools.
- Understanding how individuals react to and commit to new innovations/technology.
- Understanding how organizational culture influences the adoption of new innovations/technology.
- Utilizing effective mechanisms for institutionalizing and sustaining an adoption.

To apply this knowledge successfully requires engineering competency coupled with competencies in management and psychology. Unsuccessful attempts at change can often be traced to failure in one or more of these competencies. It is important then for organizations to develop these skills.

One approach we have seen recently is the creation of a career path in software engineering change management. Engineers who elect this path are provided with training, mentors, and a certification process designed to develop their change management skills. There is a promotion path with a top position at the executive level. Change management is treated as a key success factor in these organizations, with a career path as part of the infrastructure to support that competency.

As two practitioners in the field of change management have stated [5], "No technology will remain in first place forever. The trick is to pick technologies likely to remain useful long enough for the firm to recover its investment as well as to place itself in a position to leverage the next big technological advancement. This is a difficult task."

Even with skills for change management, technologies can still come along that are so revolutionary, it is not possible or practical to leverage the existing technologies [6]. Who will be more able to recognize and respond to this change: those with skills for change or those without? Obviously, those with change management skills will be the winners.

Change: A Medical Analogy

Let us conclude with a quick look at the field of medicine as an analogy to demonstrate the challenges of adopting new practices and why knowledgeable, experienced change-agents are so vital. While this, like most analogies, is an oversimplification, and there is always a danger of carrying it too far, we nevertheless offer it as a way of relating to some of the skills needed in change management.

The medical field offers some useful analogies to software engineering. While we think of DoD weapon systems as being highly complex, we think it safe to say that the system that physicians work on (the human body) is even more complex. In both fields however, the practices and tools of the practitioners are changing rapidly. A software engineering change-agent (the equivalent of the physician in our medical analogy) should be aware of the changes in the field and have the competencies to evaluate and select the appropriate changes for their organizations (where the organization is equivalent to the patient in our medical analogy).

We can carry this analogy further by relating the various medical specialists and diagnosticians to the various change-agent roles and skills. For example, sometimes a patient is willing to make whatever changes are necessary to achieve improvement and possibly a complete cure. They follow their doctor's advice and even take it upon themselves to learn more about their condition and treatment options. Such patients are willing adopters of the change.

On the other hand, when the changes require a complete lifestyle overhaul, this can be a much bigger challenge to the patient. When that happens, the patient may require other support in the form of psychologists, therapists, etc. that will address the various dimensions of the treatment.

Likewise, the software engineering change-agent must be prepared to address the multiple dimensions of the change – business, technical, cultural, and political. Whether these skills are embodied in one or more change-agent roles will be determined as more organizations explore the realm of software engineering change management. It is probably safe to say that organizations will have a spectrum of adopters who respond and commit differently to changes in their day-to-day software engineering practices. Hence, we believe there is a need for skilled practitioners with the know-how to address these issues.

Conclusion

Change in the field of software engineering provides organizations and software engineers with both opportunities and challenges. The opportunities are in the adoption of new practices and supporting technologies that will improve the quality of their products while at the same time enabling more predictability in their development costs and schedules.

The challenges are in recognizing that no technology or practice will remain in place forever, and the organization must be prepared to continually improve their software engineering discipline and technologies.

We propose that organizations with core competencies in change management have these advantages:

- Better able to continually identify, evaluate, and implement real improvements.
- Less likely to waste resources and time on inappropriate choices and poor implementation. Organizations need the skills to identify, select, and institutionalize new practices and technologies that will provide an acceptable return on investment and put them in a position to leverage the next big advancement. We believe these are the skills that the software engineering change-agent needs now and in years to come.◆

Acknowledgements

The authors would like to thank Dr. John Goodenough and William Pollak, both of the Software Engineering Institute, for their valuable contributions to this article.

References

1. Boehm, B. and Basili, V., Software Defect Reduction Top 10 List, *Computer*, January 2001, pp. 135-137.
2. Report of the Defense Science Board Task Force on Military Software, 1987, DTIC #ADA 188560.
3. Report of the Defense Science Board on Acquiring Defense Software Commercially, 1994, DTIC #ADA 286411.
4. Report on the Defense Science Board Task force on Defense Software, November 2000, www.osd.acq.mil/dsb
5. Carter, L. R. and Dufaud, Lt. Col. Scott, 21st Century Engineer, *Crosstalk*, December 1999, pp. 14-20.
6. C.M. Christensen, *The Innovator's Dilemma*, 1997, Harvard Business School Press, Boston, Mass.

Notes

1. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.
2. A 1997 National Research Council report, Ada and Beyond: Software Policies for the

Department of Defense, www.nap.edu/catalog/5463.html, influenced a DoD acquisition policy change to refrain from mandating ADA in DoD software-intensive systems.

3. In a series of interviews conducted last year by the SEI with several DoD industry organizations, the continued migration of software engineers to the commercial sector was attributed to the perception of more rewarding positions in the commercial sector.

About the Authors

Dr. Stephen E. Cross, Ph.D., is director and CEO of the Software Engineering Institute at Carnegie Mellon University. He is also principal research scientist in the Robotics Institute and the Institute for Software Research International in the university's School of Computer Science. Currently he is chairman of the Information Science and Technology panel. Cross has a doctorate from the University of Illinois, a master's in electrical engineering from the Air Force Institute of Technology, and a bachelor's in electrical engineering from the University of Cincinnati. He is a graduate of the U.S. Air Force Test Pilot School, the Air War College, and the National Defense University. He retired from the Air Force in 1994.

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213
Phone: 412-268-7740
Fax: 412-268-5758
E-mail: sc@sei.cmu.edu

Dr. Caroline P. Graettinger, Ph.D., leads the Accelerating Software Technology Adoption initiative at the Software Engineering Institute. She manages R&D activities to identify, develop, mature, and codify recommended practices for technology transition and change management. She also helps organizations develop capabilities for technology transition and change management. Graettinger has a doctorate, master's, and bachelor's degrees in electrical and computer engineering from Carnegie Mellon University.

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213
Phone: 412-268-6109
E-mail: cpg@sei.cmu.edu

DOES YOUR SOFTWARE PROJECT RANK AS ONE OF THE GOVERNMENT'S TOP FIVE?

The Department of Defense and CrossTalk are currently accepting nominations for the top five software projects in the government.

Outstanding performance of software teams will be recognized and best practices promoted.

Nominations will be accepted through July 20, 2001 at: www.stsc.hill.af.mil/CrossTalk

Entrants will be selected based on adherence to quality and to the contracted cost, schedule, and requirements.

To be eligible, projects must have preformed under a government contract (internal government contracts also eligible) and provided a software deliverable during the period of January 2000 through June 2001. The deliverable may be a completed project or an incremental deliverable.

Get Your CrossTalk *Free Subscription*

Fill out and send us this form.

OO-ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056
Fax: 801-777-8069 DSN: 777-8069
Voice: 801-775-5555 DSN: 775-5555

Or request online at www.stsc.hill.af.mil

Name:____________________

Rank/Grade:____________________

Position/Title:____________________

Organization:____________________

Address:____________________

Base/City:____________________

State:_____ Zip:__________

Voice: ____________________

Fax:____________________

E-mail: __________@__________

Check box(es) to request Back issues:

Apr 1999___ Aim for Quality

May 1999___ Climb to the Top

Dec 1999___ Evolution of Software

Jan 2000___ Lessons learned

Feb 2000___ Risk Management

Apr 2000___ Cost Estimation

May 2000___ Into the Future

June 2000___ Keeping Time

Nov 2000___ Software Acquisition

Dec 2000___ Project Management

Jan 2001___ Modeling/Simulation

Feb 2001___ Measurement

Mar 2001___ Process Improvement

Apr 2001___ Web-Based Apps

May 2001___ Software Odyssey

Lessons Learned From Using COTS Software on Space Systems

Richard J. Adams and Suellen Eslinger
The Aerospace Corporation

The incorporation of commercial off-the-shelf (COTS) software into software-intensive systems brings promises of reduced cost and schedule and higher reliability and maintainability by using "proven" software. However, the reality of using COTS software can be very different! This article presents the results of a survey of U.S. Air Force Space and Missile Systems Center and National Reconnaissance Office programs on their experiences with incorporating COTS software into their systems. Six major lessons learned derived from the survey are described, along with recommendations for improving the acquisition of COTS-based systems.

The new Department of Defense (DoD) 5000 series of acquisition regulations[1] requires the exploitation of commercial off-the-shelf (COTS) products in DoD acquisitions. The incorporation of COTS software into software-intensive systems brings promises of reduced cost and schedule, and improved reliability and maintainability by using *proven* software. However, the reality is often very different! The use of COTS software, which can provide significant benefits, also poses major new risks in the acquisition, development, and sustainment of software-intensive systems that must be acknowledged and managed.

In support of the U.S. Air Force Space and Missile Systems Center's (SMC's) Directorate of Systems Acquisition, the authors performed an in-depth study of actual COTS-based system (CBS)[2] development and sustainment experiences on SMC and National Reconnaissance Office (NRO) programs. This study was motivated by numerous reports of COTS software-related problems. The purposes of this study were: (1) to synthesize and share lessons learned from actual CBS development and sustainment experiences, and (2) to provide recommendations for mitigating the risks inherent in CBS development and sustainment.

COTS Software Study Process

The study process comprised three steps: gathering information, synthesizing lessons learned, and developing acquisition recommendations. The techniques selected for the first step were focused interviews and documentation reviews. A COTS software experience questionnaire was developed to provide a framework for the interviews. This questionnaire was sent to the interviewees in advance of the interview to enable them to prepare for the type of information being requested. The interview itself was then allowed to proceed as a free exchange of information, with the questionnaire being used to return conversation to the topic, if necessary. The questionnaire was also consulted toward the end of the interview to identify areas that had not been addressed. In addition, interviewees were asked to provide any written documentation previously prepared by their program on COTS software experiences or lessons learned.

The COTS software experience questionnaire requested information on both good and bad experiences with COTS software. For each experience, information was requested on the nature of the experience, where in the life cycle it occurred, the COTS software involved, the functions provided by the COTS software, and their criticality to the system. For good experiences, the interviewees were asked to identify any actions taken that contributed to the good experience. Regarding bad experiences, the interviewees were asked to do the following:

- Identify any actions taken to solve the problem or mitigate further risk.
- Provide information as to what they would have done differently to find the problem earlier and to solve the problem or mitigate the risk, given the benefit of perfect hindsight.

The authors interviewed more than 50 representatives from 18 SMC and NRO program organizations, including personnel from the government, development contractors, and The Aerospace Corporation. In addition, domain experts from The Aerospace Corporation were interviewed concerning their experiences with COTS software in their domains. The domains chosen were considered to be critical to space systems, such as computer security and telemetry processing.

Following the interviews, the authors reviewed the documentation on COTS software experiences and lessons learned that were obtained from the interviewees. Note that the information gathered was from experiences on the ground segments of space systems, due to the still rare use of COTS software in onboard software for satellites and launch vehicles.

During the second step of the study, the authors first identified and documented more than 150 distinct findings from their interview notes and document reviews. The authors then performed an iterative series of analysis and syntheses to derive lessons learned from the findings. Six significant lessons learned were identified that encompassed the collection of findings. The final step of the study was to develop specific acquisition recommendations to help mitigate the risks in CBS development and sustainment. The lessons learned and recommendations are described below.

Lesson Learned 1

Critical aspects of CBS development and sustainment are out of the control of the customer, developer, and user.

This lesson concerns the realities of the commercial marketplace. COTS software vendors are driven by today's fast-paced market, characterized by highly volatile business strategies and market positions. Vendors may go out of business, merge with, or be acquired by other companies. Vendors may also drop or de-emphasize products or hardware platforms, usually without warning.

Of particular note is the fact that the more numerous commercial customers, not the defense community, drive the market for COTS software. In some instances, the market is diverging from defense needs. One example of this is the emphasis by some vendors on Windows NT platforms for commercial users (and the corresponding dropping or de-emphasis of the high performance UNIX workstations extensively used in ground systems for defense space applications). Another example is the targeting of COTS satellite control software to the operational paradigm of commercial communications satellite customers where there is minimal human intervention rather than the person-in-the-loop paradigm of defense satellite operations.

The quality and content of COTS software upgrades are unpredictable. Vendors are market-driven in their upgrades, focusing upon additional features to attract new customers and fixes to problems encountered by their principal customer base. Vendors may not be willing to fix problems experienced by only a few customers, even if the customer is willing to pay the vendor to fix the product. This can be especially applicable to defense space applications, which may use different features or place different stresses upon the COTS software than commercial users of the same software.

Because of time-to-market pressure, vendors perform limited testing on COTS software upgrades, especially regression testing of supposedly unchanged features. In addition to introducing bugs in previously working capabilities, upgrades may decrease performance, increase computer resource utilization, and introduce incompatibilities with other COTS software products. Also, upgrades may eliminate backward compatibility with previous versions, possibly necessitating design and data structure rework. Numerous interviewees stressed the need for developers and sustainers to fully test each upgrade before incorporating it into the system.

The schedule of upgrades, both frequency and release dates, is time-to-market driven. However, the pressure to bring new features to market quickly may cause vendors to drop or slip other promised features, fixes, or upgrades for particular platforms. Sometimes needed upgrades are delayed because of dependencies between COTS software products (e.g., vendors waiting for the next operating system upgrade before issuing their next major upgrade).

The costs of COTS software products and associated services are also market driven. Fees and fee structures of licenses and services may change without warning potentially resulting in a large cost impact if changes occur after developer commitment to a particular COTS software product. One particularly damaging example of this is when a vendor eliminates site licenses and requires a separate copy of the COTS software to be purchased for each operator seat in the ground system. Vendors may also change the type and quality of the customer support they provide. In particular, new vendors trying to gain a market position may initially be very responsive but may lose their responsiveness after establishing a larger customer base.

No program organization interviewed had experienced all of the problems cited above, nor did any organization have problems with all of their selected COTS software. However, every program organization interviewed had some problems with one or more COTS software products. Encountering these realities of the commercial marketplace should be considered the norm, not the exception, in the development and sustainment of CBS. Contingencies (e.g., alternative product choices, and cost and schedule margin) to handle these occurrences need to be built into development and sustainment plans from the beginning of the life cycle.

Lesson Learned 2

Full application of system and software engineering is required throughout the CBS life cycle.

This lesson reflects the fact that using COTS software does not eliminate portions of the system life cycle or the necessity for performing system and software engineering. Using COTS software reduces the scope of software design and implementation activities for that part of the software whose functionality is provided by the COTS software. Software requirements analysis, architectural design, integration and testing, and qualification testing must still be performed along with certain detailed design and implementation tasks. Moreover, system requirements analysis, design, integration and testing, and qualification testing must still be performed for all system functionality, independent of how the functionality is implemented.

Every new COTS software release requires a full application of the system and software engineering life cycle to properly incorporate the new release into the CBS. For example, incorporating each new release can require the following:

- Regression testing and prototyping to determine its behavioral characteristics as compared to the previous release and its compatibility with other COTS soft ware in the CBS.
- Testing of new features and bug fixes.
- Modifications to glue code and user interfaces.
- Modifications to the COTS software data base/file structure and content.
- Full software and system integration testing and requirements verification.
- Training for both the software developers and the operators.

Thorough requirements analysis is especially important with CBS development since it is necessary to understand which requirements can be traded against existing COTS software capabilities versus which requirements are essential to the mission and not in the trade space. This is true for all requirements levels from the highest level system requirements through the software level requirements.

To understand existing COTS software capabilities, hands-on prototyping of the COTS software is necessary since it is not generally possible to determine the true COTS software capabilities from the vendor's marketing demonstrations and literature. Furthermore, due to the possibility of incompatibilities or adverse interactions (e.g., performance degradation) between COTS software products, this prototyping must be performed in a system context where unexpected impacts of integrating multiple COTS software products can be discovered.

Numerous interviewees emphasized the importance of designing CBS architectures to support the evolution or replacement of COTS software. Since true *plug and play* among COTS software products does not yet exist in the commercial marketplace, architectural features that help minimize the impact of upgrading to new releases of COTS software or replacing one COTS software product with another of similar functionality are essential. The CBS architecture must also have a sufficient computer resource margin and growth path to accommodate increases in resource utilization by COTS software upgrades.

Security, safety, and supportability must be designed into the CBS at the system level. COTS software capabilities in these areas are aimed at commercial applications having different, and frequently less stringent, security, safety, and supportability requirements than defense applications. Furthermore, each COTS software product is designed independently, as a stand-alone package, not as part of an integrated system. The CBS design needs to provide for integrated security, safety, and supportability features across COTS software products and newly developed or reused code. This is especially important for security since each COTS software product has its own vulnerabilities. Many of these vulnerabilities are well known in the industry, and new vulnerabilities are continually being identified. Without integrated security features being designed into the CBS, the system's vulnerabilities can be determined simply by knowing which COTS software products are in use.

Both initial evaluation of COTS software for product selection, and subsequent periodic re-evaluations of new COTS software releases for product evolution are necessary throughout the development and sustainment life cycle. The system-engineering viewpoint must be applied simultaneously to the selection of the computer hardware and COTS software. Selection of a computer hardware platform without concurrent consideration of the availability of COTS software for that platform can result in more newly developed software being required than expected, and thus can increase the system development and life-cycle costs.

The initial evaluations and periodic re-evaluations of COTS software must be based upon multi-dimensional evaluation criteria, not just upon the functionality provided by the COTS software. Examples of such evaluation criteria include the reliability of the COTS software, its ability to interface with other parts of the CBS and with legacy systems, the COTS software's implied operations concept, the vendor's characteristics, and the cost.

Finally, it is always necessary to have backup strategies and contingency plans for each COTS software product in case unforeseen problems arise that require its replacement.

Lesson Learned 3

CBS development and sustainment require a close, continuous, and active partnership among the customer, developer, and user.

This lesson concerns the need for the customer, developer, and user to be prepared to trade cost, schedule, performance, and operations and maintenance concepts to achieve the maximum benefits from using COTS software. The customer and user must understand their requirements sufficiently well to know which requirements can be relaxed to achieve a COTS-based solution and which are essential to the mission and cannot be traded. To facilitate the trades of requirements versus COTS software capabilities, the customer and user must be willing to prioritize their requirements initially and re-prioritize them as necessary throughout the life cycle. Merging of an intimate understanding of the requirements (as held by the customer and user) and an intimate knowledge of the COTS software capabilities (as held by the developer) is necessary to ensure the adequacy of these trade decisions.

Each COTS software product has its own world view that, when incorporated into the CBS, may force a particular operations concept upon the user. Frequently the COTS software operational paradigm is at odds with the user's existing operational procedures. As such, accommodating a COTS-based solution may require the user to be able and willing to reengineer existing operational procedures. Similar reengineering may be needed for existing on-site maintenance procedures. Ensuring the eventual acceptability of the CBS in the user's operational environment requires close cooperation between the user and developer.

At any time during the CBS life cycle, decisions may need to be made due to issues such as new COTS software limitations or incompatibilities being discovered, COTS software upgrades diverging from needs, or COTS software needing to be replaced due to withdrawal of vendor support. A close, continuous, and active partnership among the customer, developer, and user (e.g., via the application of integrated product and process development) will help to ensure the adequacy of the major COTS software-related decisions and the acceptability of the delivered CBS. Without such a partnership, the customer and user will not gain a full understanding of the evolving CBS capabilities and may experience unpleasant surprises when finally exposed to the capabilities of the delivered CBS in the operational environment.

Lesson Learned 4

Every CBS requires continuous evolution throughout development and sustainment.

This lesson reflects the fact that maintaining currency with COTS software upgrades is essential during both development and sustainment. Because vendors support only a limited number of past releases, delaying implementation of upgrades can result in unsupported versions of COTS software products in the CBS. When this happens, the vendor will not provide fixes to bugs and will not provide consultation services.

Delaying the implementation of upgrades can exacerbate system impacts. Upgrading from one major release to the next consecutive release does require time and effort. However, the upgrade can be considerably more expensive when attempting to skip major releases, which generally occur every 12 to 18 months. Delaying upgrades longer than that time interval can result in significant paradigm changes in the COTS software.

Sometimes upgrading to a later major release requires upgrading through each of the intermediate major releases. This is especially true if the vendor has changed the structure of the COTS software's databases or files. When this occurs, vendors frequently provide automated tools to assist their customers in conversion from one release to the next consecutive release. Such tools are not provided to assist in skipping major releases.

Many factors, both internal and external to the CBS, can drive the need to maintain currency with upgrades to the CBS' COTS software. Organizations or systems external to the CBS can require

COTS software upgrades. Examples of this include upgrades to government off-the-shelf software incorporated into the CBS, to legacy systems to which the CBS must interface, or to the Defense Information Infrastructure Common Operating Environment (if used by the CBS).

Another factor influencing the need to maintain currency with COTS software upgrades is the limited life span of computer hardware platforms (e.g., workstations and servers). Most programs plan on hardware upgrades every four to five years during sustainment. Maintaining currency with COTS software releases is essential for upgrading to new hardware since it is not usually possible or desirable to execute old versions of the operating system and other COTS software on new hardware platforms.

Furthermore, COTS software may need to be replaced or added at any time due to factors such as: elimination of vendor support, divergence from system needs, identification of unacceptable limitations or vulnerabilities, increased costs for licenses or support services, and new or modified user needs requiring changes in functionality or performance. Incorporating new COTS software usually requires the latest version of the operating system and other related COTS software to be in place.

One of the most damaging decisions frequently made in CBS development is to freeze the versions of the COTS software products throughout the development period. Due to the length of the development period for large software-intensive defense systems, this decision can result in the delivery of a system that is obsolete because its COTS software products are no longer supported. A major upgrade effort with associated cost and schedule impacts is then necessary before or shortly after the system becomes operational. Since maintaining currency with COTS software upgrades is necessary throughout development as well as sustainment, upgrading COTS software needs to be built into both development and sustainment plans from the beginning of the life cycle.

Numerous interviewees emphasized the folly of modifying COTS software, which can constrain the CBS evolution path and increase life-cycle costs. Modifying COTS software should always be a solution of last resort in CBS design. Incorporating a modified COTS software product into the CBS requires the developer and government to engage in a long-term relationship with the vendor to ensure that the unique modifications will be made to future releases. Attaining such a relationship is not always possible.

Lesson Learned 5

Current processes must be adapted for CBS acquisition, development, and sustainment.

This lesson concerns the need to modify existing processes to be suitable for the acquisition, development and sustainment of CBS. The developer's software and system engineering processes must be adapted to handle the integration of COTS software into the system. New processes must be added and existing processes updated to handle such activities as performing requirements trades against COTS software capabilities, evaluating COTS software against robust evaluation criteria, accounting for COTS software in safety, security and supportability analysis and design, and incorporating COTS software upgrades during development.

CBS development works best when iterative life-cycle models (e.g., spiral or evolutionary) and extensive prototyping of the COTS software in the system context are used together, and when the understanding gained from COTS software prototyping is integrated with the software architecture and design models. Furthermore, the time and effort distribution for development tasks need to be reallocated. Additional time and effort need to be spent on evaluation, prototyping, and analysis (the front end), and on integration and testing (the back end), and less needs to be spent on software implementation (the middle).

Numerous interviewees stressed the need for enhanced configuration management processes to handle the complexities of COTS software during both development and sustainment. The configuration management system must be able to manage multiple releases and patches to each release for each COTS software product. It must also be able to manage different configurations of COTS software at each development, sustainment, and operations facility (including mobile units), and even different configurations of COTS software on each computer hardware platform within each facility. For COTS software incorporated into firmware, configuration management cannot be performed at the board level but must be performed for the contents of the chips on the board.

Customer and user processes also need to be created or adapted to be suitable for the acquisition and sustainment of CBS. Examples include prioritizing user requirements, providing flexible and efficient responses to unexpected impacts due to problems encountered with COTS software, and handling the schedule variability of COTS software upgrades. Other examples include developing contracts compatible with the acquisition of CBS, and ensuring program milestones are compatible with the reallocation of time needed for CBS development schedules.

Interviewees also stressed the need for standardization of certain government processes as they relate to COTS software. Areas needing standardization include safety certification and security accreditation so that all parties understand the safety or security requirements that must be fulfilled when the system contains COTS software. In addition, standardized government processes for COTS software licenses need to be implemented to ensure that COTS software license currency is maintained, and that the COTS software licenses agreed to by the government are suitable for defense needs. The license for a COTS software product to be used by operational forces in the field, for example, should prohibit any expiring keys in the COTS software and should not contain any export restrictions.

Lesson Learned 6

Actual cost and schedule savings with CBS development and sustainment are overstated.

This lesson concerns the universal tendency to overestimate the cost and schedule savings due to COTS software usage (i.e., to underestimate the required cost and schedule for CBS development and sustainment). There are two principal components to this underestimation. First is completely overlooking or significantly underestimating tasks that must be performed in CBS development and sustainment, or costs of COTS software license fees and other services. Second is not allowing enough cost and schedule margin to handle unexpected impacts that can occur due to problems with COTS software at any time during the life cycle.

Here are examples of frequently overlooked tasks: hands-on prototyping of COTS software (especially in a system context); acquisition of in-depth knowledge of COTS software (e.g., training mentors and tool-smiths and purchasing vendor support); installation and configuration of the COTS software in the development and operational facilities; and preparing integrated system training and documentation in addition to the vendor-supplied training and documentation.

Also, tasks to incorporate COTS software upgrades are almost always overlooked. Examples of such tasks include performing COTS software and system regression tests for each COTS software upgrade; implementing and testing software changes needed to support the upgrades (e.g., additions or changes to glue code, databases, or configuration files); and training developers and operators for each COTS software upgrade. The time and effort for these overlooked tasks generally cannot be obtained from software cost models, but must be estimated bottom-up and included in the total cost and schedule estimates.

One area where time and effort are significantly underestimated is software engineering. Software development time and effort are generally obtained by estimating the number of source lines of code and applying a software cost model. When the decision is made to use COTS software to obtain certain system functionality, the total number of lines of code is reduced by the number of lines of code that would have been needed to provide that functionality. Using this technique with a software cost model causes the elimination of all software development activities for that functionality from the cost and schedule estimates.

However, use of COTS software to provide functionality reduces only the amount of software design and implementation effort, not all software development activities. Software requirements analysis, architectural design, integration and testing, and qualification testing must still be performed, along with certain detailed design and implementation tasks. Even if the number of lines of glue code for integrating the COTS software is added to the total software size estimate, the resulting cost and schedule estimates are not sufficient to cover all of the necessary software development activities.

While some of the newer software cost models do have features available for estimating costs associated with COTS software, these models are not yet in widespread use, and the accuracy of the resulting estimates has not yet been calibrated in the defense software environment.

Other areas where time and effort are significantly underestimated are system engineering and system integration and testing. The system models and tools currently in use for cost and schedule estimation do not adequately address incorporating COTS software. The effort for system engineering and system integration and testing is commonly estimated as a percentage of the total development cost. When COTS software is used to provide some system functionality, the reduction in the software development effort causes a corresponding reduction in the system engineering and system integration and test effort. This reduction is not warranted since the same system engineering and system integration and testing for that functionality must still be performed, independent of whether COTS software or developed code provides the functionality.

Costs of COTS software license fees are also frequently overlooked or underestimated. The number of different COTS software products required to implement the CBS and the number of individual licenses required to be purchased are difficult to estimate, especially early in the life cycle before the design is known. Additionally, vendors usually charge for services not included in their standard licenses. Examples of such services are on-site vendor assistance during development or operations, and escrowing source code to protect against the possibility of the vendor going out of business.

CBS cost and schedule estimates almost never contain enough margin to handle the COTS software problems encountered in CBS development and sustainment. As described above, unexpected impacts can occur with COTS software at any time during the life cycle. The lack of appropriate margin results in cost and schedule overruns when COTS software-related problems occur. When cost as an independent variable (CAIV) is applied, the cost of handling unexpected COTS software problems can mean that system capabilities must be deleted to balance the cost. Cost and schedule estimates for CBS development and sustainment should always contain a planned margin (i.e., management reserve) for handling the unexpected COTS software problems that are certain to arise.

Acquisition Recommendations

The government needs to be an intelligent CBS buyer. Accomplishing this requires appropriate planning and contracting for CBS acquisition to handle the issues described in the lessons above. In particular, it should be noted that defense CBSs are almost never commercial items in themselves, but are large, complex, software-intensive systems, some of whose components contain COTS software. What is desired is a balanced solution among COTS, reuse, and newly developed software to meet the CBS cost, schedule, and performance objectives. Therefore, commercial item procurements (i.e., FAR 12 acquisitions) are almost never appropriate vehicles for acquiring defense CBSs.

To support defense programs in acquiring CBSs, it is recommended that several cross-program horizontal engineering initiatives be established. First, guidance for CBS life cycle cost and schedule estimation needs to be developed to address the problems described in Lesson Learned 6. Second, a repository for actual development and sustainment experiences with COTS software products (as opposed to vendor marketing information) needs to be developed and made accessible to CBS acquirers, developers, and sustainers. Lastly, specific CBS acquisition guidance that can be tailored to individual programs is needed, such as recommended contract structures, language for incorporation into contracts, and guidance for applying evolutionary acquisition.

Conclusion

The potential benefits of using COTS software in defense systems are extensive. Today's complex defense systems require the leverage provided by COTS software, that is, enhanced system capabilities with reduced cost and schedule. The use of COTS software enables the government and developers to focus on providing the defense-unique needs.

This study demonstrated, however, that only careful acquisition, development and sustainment preparation and execution achieve the potential CBS benefits. CBS success depends upon preparing for a complex development and sustainment effort; preparing for inherent cost, schedule, and performance risks beyond government or developer control; and preparing to make adjustments to current acquisition, development and sustainment processes. While this study was conducted on defense space systems, the authors believe that the lessons learned are not limited to that domain, but are widely applicable to the use of COTS software in any large, software-intensive system.◆

NOTES

1. See, for example, DoD Directive 5000.1, Oct 23, 2000, paragraph 4.2.3.
2. For this paper, a COTS-based system (CBS) is defined to be a system that contains commercial-off-the-shelf *software* products as elements of the system.

About the Authors

Richard J. Adams is a senior engineering specialist at The Aerospace Corporation with more than 30 years experience in software engineering, software project management and software acquisition. Previously he worked for TRW, where he developed software and managed software development projects for DoD software-intensive systems. He has a bachelor's degree in mathematics from California State University at Long Beach.

The Aerospace Corporation
Mail Station M1/112
P.O. Box 92957
Los Angeles, CA 90009-2957
Phone: 310-336-2907
Fax: 310-336-4070
E-mail: Richard.J.Adams@aero.org

Suellen Eslinger is a distinguished engineer at The Aerospace Corporation with more than 30 years experience in software engineering and software acquisition. Previously she worked at Computer Sciences Corporation and General Research Corporation where she developed software and managed software development projects for DoD and NASA software-intensive systems. She has a bachelor's degree from Goucher College and a master's degree from the University of Arizona, both in mathematics.

The Aerospace Corporation
Mail Station M1/112
P.O. Box 92957
Los Angeles, CA 90009-2957
Phone: 310-336-2906
Fax: 310-336-4070
E-mail: Suellen.Eslinger@aero.org

Coming Events

June 11-13
E-Business Quality Applications Conference
qaiusa.com/conferences/june2001/index.html

June 18-22
ACM/IEEE Design Automation Conference
www.dac.com

June 25-27
2001 American Control Conference
acc2001.che.ufl.edu

July 1-5
Eleventh Annual International Symposium of the International Council on Systems Engineering
incose.org/symp2001

July 19-21
2nd Int'l Symposium on Image and Signal Processing and Analysis ISPA'01
ispa.zesoi.fer.h

August 5-10
HCI International 2001: 9th International Conference on Human-Computer Interaction. (UAHCI 2001)
hcii2001.engr.wisc.edu

August 27-31
Fifth IEEE International Symposium on Requirements Engineering
www.re01.org

August 27-30
Software Test Automation Conference
http://www.sqe.com/testautomation

September 10-14
Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)
www.esec.ocg.at

October 29-November 2
Software Testing Analysis and Review
http://www.sqe.com/starwest

February 4-6, 2002
International Conference on COTS-Based Software Systems (ICCBSS)
At the Heart of the Revolution
http://www.iccbss.org

Continued from page 21

7. Davis, Alan; Overmyer, Scott; Jordan, Kathleen; Caruso, Joseph; Dandashi, Fatma; Dinh, Anhtuan; Kincaid, Gary; Ledeboer, Glen; Reynolds, Patricia; Sitaram, Pradhip; Ta, Anh; and Theofanos, Mary, Identifying and Measuring Quality in Software Requirements Specification, Proceedings of the First International Software Metrics Symposium, 1993, pp. 141-152.
8. Common Criteria, International Standard (IS) 15408, csrc.nist.gov/cc/ccv20/ccv2list.htm, September 2000.
9. Beck, Kent, Embracing Change with Extreme Programming, *Computer*, October 1999, pp. 70-79.

About the Author

Richard Duncan is pursuing a master's degree in computer science at Mississippi State University (MSU) with emphasis in software engineering. He has held summer internships at Microsoft, AT&T Labs Research, and NIST. His current research interests involve applying software-engineering process to the development of a public domain speech recognition system at the Institute for Signal and Information Processing at MSU.

Mississippi State University
P.O. Box 9571
Mississippi State, MS 39762
Voice: 662-325-8335
Fax: 662-325-2292
E-mail: Richard.Duncan@ieee.org

Letter to the Editor

Editor:

Just a note to let you know the address for the SPI EGroups web site you reference on page 19 of March's CrossTalk needs to be updated. Yahoo recently took over eGroups so the address is now http://groups.yahoo.com/group/spi

I am the creator, owner, and moderator of this group, and just out of curiosity, was wondering how you came about including it in this issue. I'm very pleased to see it there!

Thanks,
Maj. Andrew D. Boyd
Chief, Software Quality Assurance Section
USAF

Have We Lost Our Focus?

Having recovered from a wonderful case of laryngitis, I just got back from a great Software Engineering Process Group conference in New Orleans. A great time was had by all. Between the SEPG conference and reviewing papers for this issue of CrossTalk, I am totally up-to-date on the latest and greatest in methodologies.

Back in 1969 I started my life as a programmer/developer/computer scientist/software engineer. (As you get promotions, you don't just get pay raises. You get neater and spiffier job titles, too!). I first learned to program a Wang programmable calculator in junior high – a gift to the school from a local company that needed a tax break. At that time the calculator was the size of a desk and filing cabinet with (a whopping) 256 bytes of memory, a paper tape reader, and a single punch card input. Not a *deck* of cards, mind you, but a single card: punched by hand, 80 columns, 12 rows.

There were only 61 possible instructions in the Wang instruction set, so each column in the punched card could encode two six-bit operations. Therefore, each card could contain 160 instructions. To gain admission to the "Computer Science Club," you had to be able to program the quadratic equation on a single card. (Remember ax 2 + bx + c = 0, given a, b, and c, solve for x?) Once you accomplished this Herculean task, why, you were considered a programmer!

I remember how *smart* I felt after accomplishing this task. I was a hacker, a member of the brotherhood/sisterhood! I could code! We didn't use a methodology – design was for wimps! I remember trying to explain to my girlfriend about what I had accomplished. (This might explain why I didn't date much during school).

Fortunately, I matured in my profession. I joined the Air Force and had to learn how to design. As my first design methodology, I flowcharted my code. Of course – as a true hacker – I knew that if I finished the code first, the flowcharting was much easier. I was lucky – I had a wise and tolerant boss who showed me that flowcharting was a requirements and design tool. That when the problem got too big for me to understand, I had to use some tool to help me understand and partition the problem.

I eventually became both a computer scientist and later, a software engineer. Over the years, I went from flowcharting to Top Down Structured Design (TDSP); Hierarchical Input, Process, Output (HIPO); Structured Analysis and Structured Design (SASD); Transform Analysis (TA); Object-Oriented Modeling and Design (OOMD); Unified Modeling Language (UML); and Unified System Development Process (USDP). I tried to become a Certified Data Processor (CDP). I earned degrees from institutions that are Computer Science Accreditation Board-compliant (CSAB). I studied the Software Engineering Body of Knowledge (SWEBOK). I learned lots of acronyms and new methodologies – new methodologies? You know, when I first saw activity diagrams and sequence diagrams in UML and USDP, I felt right at home. I had come full circle. I was flowcharting again!

It's never been about the methodology. Methodologies are simply techniques and tools to help you partition and understand the problem. Here's a kernel of truth – you can't design a system (let alone code it) unless you understand what the system is supposed to do. The methodologies are there to assist me. They are NOT ends in themselves – they are simply a means to the end. You have to focus on the larger picture – getting the system "out the door" on time. It's not enough to be an expert in the latest and greatest methodology, unless you can also use it to help produce a system on time and under budget. The methodology has to make you more productive. If a methodology helps, use it! If it doesn't, get a better methodology! Focus on the ends, not the means.

And – if all else fails – try drawing a flowchart. It might make you feel SMART again!

David A. Cook
Principal Engineering Consultant
Shim Enterprises, Inc.
david.cook@hill.af.mil

A Free Workshop With A Beautiful Backdrop

Configuration Management: Spanning the Software Development Life Cycle

It is FREE to U.S. government employees, and will be set in the spectacular mountains of northern Utah.

Expert consultants from the Air Force s Software Technology Support Center, David Cook Ph.D. and Theron Leishman, will conduct **Configuration Management: Spanning the Software Development Life Cycle** workshop in Utah on **July 17-19, 2001**.

Here is some of the knowledge you will gain:

Understand basic concepts and principles of software configuration management.
Become familiar with the various software configuration management roles and responsibilities.
Identify the role of software configuration management in each phase of the development life cycle.
Understand the relationship between software configuration management and the CMM®/CMMI.
Understand how to establish a change-management process.

Plan to come early and enjoy what Utah has to offer: golfing, fly-fishing, hiking, biking, canoeing, rock climbing, river running, historic locations, and more. For travel ideas visit **www.utah.com.**

SPACE IS LIMITED. Act quickly. To reserve your place at the workshop, contact Debra Ascuena at 801-775-5778 or debra.ascuena@hill.af.mil. For additional information visit our web site at

www.stsc.hill.af.mil

STSC

MISSION CRITICAL SOFTWARE CRSIP

Sponsored by the Computer Resources Support Improvement Program (CRSIP)

CrossTalk / *TISE*
5851 F Ave.
Bldg. 849, Rm B04
Hill AFB, UT 84056-5713

PRSRT STD
U.S. POSTAGE PAID
Kansas City, MO
Permit 34