The Future of Software Development and Its Impact on APL

Paul A. Hanke, Hilary L. Hershey, and Pamela A. Smith

On a cold morning in January 2015, Kendall McNeill, an APL Service Ecosystem Software Developer, arrives at her office and her hand-sized, personal/business computer/phone is automatically connected into the desktop peripheral support station with high-speed, secure network connections, additional storage, several large touch screen displays, etc. A verbal command validates her identity and connects her to the peripherals.

One of Kendall's large displays is dedicated to the online team interaction management space where team members working at home and located physically throughout APL and its field offices interact via managed collaboration processes in which development artifacts are tracked and shared. On a second screen, she brings up the graphical palette through which she does all "programming" and begins work on an interface ontology that will permit the semantic middleware to expose an aging system as a collection of user-specified services within the user's next-generation service ecosystem. She drags a "web service adapter" component onto the workspace and sets the underlying communications as SOAP 3.1. The semantic agent in the adapter displays its best guess at an interface ontology to map the aging system's interface to the user's service interface specifications; Kendall starts correcting and fine-tuning the agent's initial guesswork and takes note of the semantic gaps that will have to be filled by external sources. Meanwhile, a Requirements Change process workspace appears on her team display, and the voice of the systems engineer on the project, Liam Franks, speaks: "I just reviewed the latest User Process Execution Language software submitted by our client and it introduces a new service abstraction to their ecosystem—I think we can implement this new service via the system you are currently abstracting. As you can see from the user's simulation results, the nonfunctional constraints for this new service are. . . ." Kendall views the new service specification in real time and tells Liam she needs to pull in Jacob. She drags Jacob's picture into the Requirements Change process workspace, where they discuss the viability of the change as well as potential impacts and rework. Their discussion session is recorded within the context of the Requirements Change process instance and will be stored in the project archive for future reference. The group estimates that the change is realizable within project constraints, and Liam accepts the change. The Requirements Change process concludes by merging the new service specification and Liam's design decisions into the project requirements and design artifacts, respectively, as well as deploying the user's software to the development environment.

INTRODUCTION

This scenario is not that farfetched. Traditional approaches to the development of traditional monolithic software systems have emphasized requirements discovery, design, coding, testing, and deployment iterations in various time cycles, with each stage having distinct methods, tools, and terminology. Unfortunately, such traditional approaches are geared toward the development of software systems that address only isolated parts of the mission of an organization and, with delivery cycles that can be measured in years, are typically obsolete with respect to the organization's evolving mission by the time they are deployed. Thus, the discipline is moving away from the development of monolithic software systems toward the development of integrated systems of systems and on to globally distributed ecosystems of services where a "system" may indeed exist only ethereally, assembled on the fly in response to user input. As such, the role of "programmer" is permeating all the way out to the user. We are beginning to see the seeds of new software development paradigms where the activities of requirements discovery, design, coding, and testing are continuous processes carried out at multiple levels of abstraction, where the software developed by domain experts at higher levels of abstraction forms the requirements specifications for the next level down, and where the software developed at lower levels of abstraction implements the nontechnical, domain-specific "programming primitives" for the next level up. Add to this the ongoing advancements in computer support for knowledge work and team collaboration, and a vision for the software development of tomorrow begins to coalesce—a vision that supports the full breadth of an organization's mission and at the same time is responsive to the (sometimes rapid) evolution of that mission.

This article peers into the fog and attempts to locate factors that will lead us toward a next generation of software development that can fulfill this vision. We limit ourselves to four broadly defined areas in which we expect to see the largest divergence from current software development practices: the art and science of software development, programming languages, ecosystems of services, and computer support for knowledge work and team interaction. The goal of this article is to forecast a list of nontraditional competencies that APL will need in order to continue to contribute in both a science and technology domain and as a trusted partner with our sponsors.

ART AND SCIENCE

We can define "development" in the abstract as the systematic application of art (e.g., project management) and science (e.g., predictive models) to otherwise unpredictable creative and constructive endeavors with the goal of making them more predictable. Many of the more mature development disciplines enjoy relatively high degrees of predictability as a result of a highly stable substrate (e.g., physics) upon which to anchor their science.

Unfortunately, the discipline of software development today does not have a mature science (and quite possibly no stable substrate upon which to anchor that science). At the lowest levels, computer science provides a set of theoretic models—information theory, automata theory, computability theory, etc.—that yield good predictions for what can and cannot be computed (either in the absolute or within a tractable amount of time). However, at the levels where the science meets development (e.g., automata theory, a mathematical model of computation and the basis of many imperative programming languages), things are less well defined. Automata theory provides a good model for monolithic computational structures, but it is not as well suited as a model of today's integrated systems of systems nor tomorrow's distributed ecosystems of services. Alternative theoretical constructs (e.g., the pi calculus, a calculus of communicating processes) are emerging to fill the void. Whether any of these alternatives will simply complement or actually replace automata theory in a Kuhnian paradigm shift remains to be seen. And as far as the stability of the substrate goes, new advancements in computing (e.g., quantum computing) threaten to upset the pillar of mainstream computer science down to its very foundations.

Without a mature science upon which to base the predictability of the endeavor, software development today is instead heavily focused on the art component of development. In fact, the art of development has been so extended by software development in its short history that it is informing the other development disciplines with ways and means to enhance predictability in those endeavors as well. Some of the biggest advancements in the art relate to dealing with extreme complexity; today's software systems are some of the most complex systems ever devised by humankind. There have been monumental failures, such as the Federal Aviation Administration's Advanced Automation System, and monumental successes, such as the World Wide Web. These two exemplars demonstrate the range of organization of software systems themselves (highly structured versus self-organized) as well as the range of organization of the groups that build them (bureaucratic versus social).

The art of software development is currently focused on highly structured software systems built by bureaucratic organizations through the application of systematic approaches such as industry standards (e.g., the unified modeling language [UML], web services), model-based approaches to development (e.g., model-driven architecture [MDA]), and knowledge-based approaches to development (e.g., design patterns). However, in the future (and with the current momentum steering toward loosely coupled ecosystems of services that are evolved in a social context), the software development discipline may in time become a driving force in the ongoing development of the nascent sciences of complexity (nonlinear dynamics, chaos theory, etc.) as a means toward understanding (at least qualitatively at first) the evolution and behavior of these self-organizing systems.

APL has the opportunity to contribute on numerous fronts here—from the advancement of computing (e.g., quantum computing) and the associated models of computing (computer science) via pure research through advancing the art and science of software development (e.g., project oversight and the sciences of complexity)—via applied research and development. And with the experience gained through such endeavors, APL will be able to assist and guide our sponsors in the application of these advancements.

A LITTLE SOFTWARE DEVELOPMENT JARGON

Design patterns: Design patterns are standard representations of innovative solutions to common problems that have evolved over time. As such, they capture and disseminate industry knowledge and expertise and form a common solutions vocabulary that greatly enhances communication among software developers.

Engineering: Some people define "engineering" as the science end of the development spectrum (the art component of development is then referred to as "engineering management"), whereas other people consider "engineering" to be synonymous with development. There is an ongoing debate on this subject, and the former definition is often cited to question whether there is such a thing as "software engineering." Here we avoid this political fog by refraining from the use of the word.

Industry standards: Industry standards pave the way toward seamless integration at all levels of software development, be it the communication of modeling artifacts via the unified modeling language (UML) to the run-time interoperation of software components via the web services standards. Widespread acceptance of industry standards is a necessary step in the maturation of an industry, as evidenced, for example, by the electronics component industry. The software development industry is finally beginning to appreciate and embrace this reality.

Model-driven architecture: The goal of the MDA initiative is to use standards-based modeling languages as formal development languages. Fully expressed MDA models can be directly executed and "compiled" into 3GL languages and other lower-level development artifacts.

Ontology: Ontology refers to the full spectrum of knowledge representation. Examples range from simple taxonomies to thesauri to conceptual models and all the way to logical theories. The more interesting ontologies, such as conceptual models and logical theories, enable automated reasoning. Automated reasoning facilitates the automation of decision support and knowledge management, and thus ontology forms a cornerstone of the semantic web.

Semantic web: The semantic web is a vision of the Internet populated with machine-processable data. This vision is not an academic stretch for a new artificial intelligence, but rather an application of the pragmatic fallout of existing artificial intelligence research. The business case for the semantic web is the facilitation of automated decision support and knowledge management.

PROGRAMMING LANGUAGES

Programming languages are the way we express precisely (in the most literal sense) what it is we want our computers to do. The first programming languages were those spoken by computers natively: machine codes. These programming languages forced the programmer to think like the machine all the way down to the bit level. Not only was this incredibly tedious work (humans do not naturally think in machine codes), but it required huge amounts of manual translation to get from an end-user's requirements down to the working software. As requirements were typically expressed in imprecise human language, interpretational errors frequently arose, and the frequency increased with the size of the project as there were more humans (e.g., designers) interjected between the end user and the programmer (allowing for multiple layers of interpretational error). And when the cost of computing hardware began a dramatic nosedive, the expense of developing software in this labor-intensive and error-prone way began to stand out like a sore thumb. (See the box, "Five Generations.")

Assembly languages were the first improvement over programming in machine codes. Essentially, assembly languages represented bit-level machine codes with mnemonic symbols, thus allowing the programmer to think in terms of machine instructions. This device improved the speed at which a human could program a computer by reducing the amount of memory the human had to devote to rote memorization (mnemonic symbols instead of bit strings) as well as by reducing the number of keystrokes involved. However, assembly languages were just the computer's native language coated in a symbolic sugar. Thus, assembly language did nothing to reduce human interpretational error.

The next evolutionary step in programming languages was the first of the so-called third-generation languages (3GL). Many of the 3GLs are heavily influenced by automata theory and in addition provide structural elements (e.g., "for" loops) that represent more natural ways for humans to think (when compared to machine instructions). 3GLs mark the first step up the ladder toward executable end-user requirements; 3GLs allow the function-level structural elements of software to be expressed as abstract requirements, which can then

FIVE GENERATIONS OF PROGRAMMING LANGUAGES

Generation	Term	Began in	Description
First	Machine code	The beginning	The actual 0s and 1s that computers still actually load and execute. Early programmers developed programs using this numeric representation of instructions and data. 0100010111100011 might represent the "Move value to registry" function.
Second	Assembly	1950s	A symbolic representation of machine code, where 0100010111100011 would be written as "MOVR" or "move value to registry." Assemblers would translate this symbolic representation to machine code. Both machine code and assembly are highly dependent on the underlying processor. When the processor changes, the code must be rewritten.
Third	High-level language	1960s	These human-readable languages support constructs such as "IF-THEN-ELSE" and named variables. Adding three variables together may take eight lines of code in assembly language but could be represented as a single statement "A=B+C+D" in a 3GL. 3GLs are the last in the evolutionary line of programming languages to be completely general-purpose.
Fourth	Very-high-level language	1980s	These languages include support for the end use of the software without becoming domain-specific. Examples of 4GLs include form generators, report generators, and database query languages. Often, a 4GL will have pragmatic (as opposed to theoretic) origins. 4GLs are "niche languages" in that they cannot be used to implement entire software systems; rather, 4GLs are applied as a time-saving device within specific areas of the system.
Fifth		1990s	These languages allow the expression of nonalgorithmic rules, constraints, etc., which the computer then uses to solve a problem. They are sometimes called "declarative" to distinguish them from the imperative languages and frequently originate in artificial intelligence. An example of a 5GL is the OPS5 expert systems language. The use of 5GLs is increasing with the rise of the business rule engine market.

be mechanically translated ("compiled") into machine codes that implement requirements so expressed. 3GLs allow humans to think more naturally. But the 3GLs accomplish something else of even greater import: they eliminate an entire level of human interpretational error; an algorithm can be specified in the abstract in a 3GL, a button pushed, and the whole of the algorithm can be translated automatically into machine codes without further human intervention.

3GLs continue to evolve today, the driving factors of which hinge on reducing the costs associated with this still labor-intensive activity:

Modularity and the separation of concerns. 3GLs continue to improve support for modularity and the separation of concerns. Modularity results from the decomposition of a larger system function into smaller and smaller functional components, each separately designed, developed, and tested the way electrical engineers test individual components of a circuit board before adding them together. Separation of concerns enhances modularity by requiring that each module "do one thing and do it well" (a phrase that is relative to the level of decomposition). Perhaps the more important feature of modularity and the separation of concerns from a software development standpoint is that it permits multiple people to work relatively independently but simultaneously on multiple portions of the larger system. In modern 3GLs, modularity is supported through object orientation, and the most extreme form of separation of concerns is aspect-oriented programming.

Labor reduction. 3GLs continue to offer improved opportunity for reducing the amount of code that has to be developed (designed, written, and tested). Assembly languages began this trend by offering macros as a means of repeating the same set of instructions on a different set of data. The first 3GLs offered related constructs such as subroutines and functions. Modern 3GLs offer object- and aspect-oriented features, including classes and inheritance, which permit developers to reduce the amount of code that needs to be developed while providing improved support to enhance the modular quality of the code by reducing coupling, increasing cohesion, and supporting generative programming.

Reuse. Long considered the holy grail of software development, the ability to reuse previously written software is facilitated with modern 3GLs and industry standards. In the computing hardware industry, reuse is facilitated by industry-standard hardware interfaces between hardware components and software drivers that resolve any remaining incompatibilities. Unfortunately, the software industry is only recently seeing the value in industry-standard interfaces between software components, and there is no analogous technology that does for software what software drivers do for hardware (thus, software itself has to deal with any remaining incompatibilities that exist beyond what is covered by industry standards). Widespread reuse, therefore, is still limited to a few key areas: COTS utility libraries, including such features as user interface services, communications services, and operating system services; class- and component-based frameworks; and large-scale software components for which industry-standard interfaces exist (e.g., relational databases).

Support tools. From interactive line editors in the late 1970s replacing punch cards, to source code analyzers in the 1980s, to the integrated development environments (IDEs) of the 1990s, the emergence of supporting tools to facilitate the authorship of 3GL software has helped improve the productivity and quality of developers who are being asked to develop increasingly complex systems, systems of systems, and ecosystems of services. IDEs today improve developer productivity (decreasing costs) by facilitating reuse of COTS components through drag-and-drop interfaces, supporting debugging through online code-generated documentation and call trees, and integration with configuration management and project management tools.

Today's IDEs offer an early window into some of the key capabilities that will be used by future APL software developers in the development of end-to-end software systems. And although 4GLs and 5GLs can only be applied to selected facets of a software system, 3GLs are not the end of the software systems development story.

The use of models, or simplified representations of complex systems, is a proven tool for aiding a person or a team to effectively understand and discuss, and preserve for posterity, a description of a system. In the early years, modeling became necessary for the domain expert to communicate with the software developer or for multiple software developers to communicate with each other. Early on, the value of graphical modeling, and in particular, data flow diagrams, were (and continue to be) clearly demonstrated as effective tools in understanding a system.

Tools to support the design process began with the addition of plastic templates with flowchart symbols to expensive, integrated design tools that could actually generate some semi-usable 3GL statements (hold on to that thought). A key milestone in the modeling process was the acceptance of the UML in the 1990s as the *lingua franca* for software modeling.

As with any other tool, models could be (and were) misused. Design documents containing hundreds of pages of graphical models became maintenance nightmares as the 3GL software changed and the associated design documents were not kept in sync. Reverse engineering (ingesting the 3GL software, producing design documentation) was seen as one solution, but many software developers viewed the large library of models as a management cost, not providing much value added over actually understanding the 3GL software using an IDE's capabilities.

Historical problems with modeling have included inconsistencies resulting from the two representations of the software—the 3GL software itself and the models of the 3GL software are inconsistent, and the modeling languages and tools used by the systems developers who define the requirements of the system and the software developers who build the system are divergent. If only software models could be compiled into 3GL software the way that 3GL software is compiled into machine codes!

Steps are being taken to rectify the situation, and not just at the level where software models meet 3GLs. The UML is being fleshed out into a systems-level programming language in its own right (executable UML), and the MDA initiative proposes the compilation of such platform-independent UML models onto real-world 3GL platforms such as the Java 2 Enterprise Edition and the .NET Framework. And moving ever closer to the end-user, industry-standard "service orchestration" languages oriented toward the end user are emerging and being compiled downward toward the level of 3GLs. (For example, the business process modeling notation is a graphical language for expressing the logic that connects ecosystems of services into systems that automate a user's strategic, tactical, and operational processes. Tooling already exists that will compile the business process modeling notation down to the lower-level business process execution language.) Viewing the levels of magnification that separate the end user from machine codes as a sort of fractal geometry, where the qualitative aspects at one level of magnification are the same as at all others, it would appear that we are standing at the cusp of realizing executable end-user requirements.

Certainly, the software written by an end user in a service-orchestration language is not the whole story. Rule languages are another format for end users to express their requirements, and with the rise of the business rule engine market these requirements, too, are now directly executable. Ontology languages for describing the end-user's domain will form another component of executable requirements. So do executable requirements leave the software developer out in the cold? In a word, no—it just changes the nature of the job. Instead of developing monolithic systems and/or integrated systems of systems, the software developer of the future will be charged with developing ecosystems of services that form the programming primitives that the end user will orchestrate into automated processes. This has huge implications for how the business of software development is conducted: no longer will the end-user's requirements flow down through multiple translations and interpretations until they reach a 3GL programmer in a form that is unrecognizable to the end user and virtually untraceable back to end-user requirements; rather, the end-user's process-level software will be compiled down to the ecosystem developer's systems-level language, whereupon the ecosystem developer will flesh out and allocate the logic by applying annotations and/or aspects to the end-user's logic and then the ecosystem developer's systems-level software will be compiled down to the service developer's service-level language, whereupon … and so on. In this broad brushstroke of a vision of the future, human interpretational error as well as the amount of intellectual capital that is typically expended on these manual translations and interpretations, could be radically reduced.

If any of the above initiatives and prognostications should come to fruition, APL has the opportunity to contribute on numerous fronts, a major one being the development of simulation modeling and analysis methodologies and tools at each level of software as a means to discover nonfunctional requirements for lower-level services. At any rate, APL will need to radically alter its software development practices to embrace the new paradigm and could provide leadership and oversight in this respect to our sponsors.

ECOSYSTEMS OF SERVICES

"Ecosystems of services" is a phrase intended to reflect the realities of global services networks such as the web and the Global Information Grid. These systems are of unprecedented size—only the Internet is bigger, and the lesson learned there is that even a simple packet-switching network operating under a few simple local rules can come crashing to its knees when the unforeseen and emergent global consequences of those rules suddenly come into play. The Internet's transmission control protocol (TCP) was years in the tuning, finally achieving a sense of stability that was based largely on empirical findings rather than theoretical predictions; it was only within the last 10 years that mathematical models that define "TCP-friendly" protocols were discovered and documented to quantify whether new rate-controlled protocols under development would starve, coexist with, or be starved by the Internet's TCP.

On the top-down side of the coin were the first service-oriented architectures (SOAs) such as the distributed computing environment (DCE) and the object management architecture (OMA, under which the common object request broker architecture or CORBA standard was developed). These first-generation SOAs focused on fine-grained networked infrastructure services such as directory, security, and transaction services that help to greatly simplify the construction of complex distributed systems of systems. One of the findings here was that these fine-grained services overuse the network, and pay a heavy performance penalty. A common end result was the bundling of such services within component frameworks (commonly known as "application servers") such as the CORBA Component Model, J2EE, and .NET, such that these horizontal services could be collocated with the applications that use them.

In a vertical leap, second-generation SOAs are newly focused on coarse-grained domain-specific services that provide access to the various systems that underpin an organization's strategic, tactical, and operational processes. Second-generation SOAs are also rallying around the technology standards that underpin the Internet, the web, and the next-generation web—the semantic web—to gain universal acceptance (something neither DCE nor CORBA ever achieved). Second-generation SOAs promise vastly improved transformational and adaptive capabilities for an organization's strategic, tactical, and operational processes via the managed orchestration of ecosystems of domain-specific services. These transformational and adaptive capabilities will only improve as the emerging techniques and technologies of the semantic web (e.g., ontology) are brought to bear, where the negotiation and mediation of the interfacing between software entities and their users can become more dynamic and automated.

How will this affect software development at APL? It will do so by changing the environments into which APL will be expected to deploy software as well as the artifacts that compose that software. Rather than simply developing monolithic 3GL applications that meet end-user requirements and deploy on specified hardware, APL will in addition need to study the end-user's existing ecosystem of services in order to shorten the development cycle through the reuse of existing services and to ensure graceful integration within the service ecosystem (e.g., design new services not only for functional completeness but also for minimized detrimental perturbation of the existing ecosystem dynamics). And the software that APL deploys will not only consist of 3GL statements but will also include domain descriptions (ontology) and service descriptions (e.g., the semantic web service language) to facilitate the automation of interface negotiation and mediation in an ecosystem of services.

COMPUTER SUPPORT FOR KNOWLEDGE WORK AND TEAM INTERACTION

Today's computer support for knowledge work and team interaction is often akin to massive toolboxes—the tools are on display for the expert craftsman to grasp and use with ingrained skill. Unfortunately, there is a steep learning curve to becoming such an expert, and this presents an intellectual challenge that robs from the very resource that needs to be applied in such work. For computers to provide true support for knowledge work and team interaction, they must do so "invisibly," that is, with the minimum amount of intellectual challenge in their use.

Process management technology is a step in the right direction, as the description of a process encodes a level of expertise in the use of these massive toolboxes. Benefits of process management technology include the ability for end users to encode process descriptions in a language that is familiar to them as well as the automation, tracking, and collection of performance metrics of process instances. Unfortunately, the current direction in process description is derived from predecessor programming languages, which were designed to encode precisely (in the most literal sense) what we want our *computers* to do.

Instead of process description languages that encode precisely what we want our knowledge workers and team members to do on the one hand and massive toolboxes that require extreme expertise on the other, what is needed is a balance between the two; i.e., process description languages that are prescriptive of what needs to be done and when, yet only suggestive of how to go about doing it. Such languages allow knowledge workers and team members to backtrack, loop, and even modify the suggestive elements of a process instance so long as the prescriptive constraints are not violated. In this way management controls are in place (the prescriptive elements), hard-won expertise is brought to bear (the suggestive elements), and the management of the process can be automated (via the formal process description), yet there is still enough flexibility for the knowledge workers and team members to "own" the work. The technology based on this concept of process description languages that support human and social processes has been termed the Human Interaction Management System (HIMS).[1]

Note that the HIMS, in presenting suggestive direction to knowledge workers and team members, is a decision support system. As such, it might be enhanced as needed with more powerful decision aids such as fuzzy expert systems and Bayesian belief networks. But equally important to the human-centric element of the HIMS concept is that these processes and decisions are being automatically managed by the HIMS, that is, *processes and decisions will form the basis for new metrics of the software development of the future*. No longer will the metrics that are used to characterize the software development activity be solely "black box" (i.e., measure the outputs of the activity such as the number of 3GL statements); instead, the metrics will be more reflective of the intellectual and social inner workings of the activity itself.

Today, the intellectual and social inner workings of the software development activity include standard project management practices such as expectation (requirements) management, scheduling, quality reviews, as well as practices that, although not unique to software development, often provide the only visible insight into the status of a development effort: metrics definition/collection/analysis, problem tracking, configuration management, and demonstrable milestone definition.

The Capability Maturity Model (CMM) for Software and later-generation, related CMMs offer a means for understanding and communicating standard policies, processes, and practices related to the software development effort. The CMMs also offer methods for numerically assessing the maturity of the software development organization. The premise is that mature organizations more effectively manage the risk associated with software development and produce more consistent results. Currently, the CMM emphasis on process over product results in unwieldy, inflexible, and costly practices, but application of the HIMS to automate and manage the CMM processes could change all of that. The CMM assessment process has been most useful to the DoD as a method to reduce the risk associated with the DoD's unique, one-of-a-kind software development—HIMS-managed CMM processes could vastly reduce the costs associated with this risk reduction activity as well as facilitate a greater uptake of these processes by a cost-conscious commercial industry.

Under the sponsorship of the APL Science and Technology Council, APL undertook a Laboratory-wide effort to improve the management practices associated with the unique risks associated with software-intensive projects. These practices and procedures have now been integrated into the APL Quality Council initiative and are based on components of Carnegie Mellon University's Software Engineering Institute's CMM as they are most appropriate for the breadth and depth of APL-developed software and its intended use. However, our forecasts for qualitative change in how software is developed (executable requirements, ecosystems of services, etc.) imply related qualitative changes in how software development is managed, and thus from a quality perspective, "continuous adaptation" may be just as important as "continuous improvement" when it comes to APL's software development practices. In this respect, it could be beneficial to use modern data mining techniques to continuously mine the data collected by software development HIMSs in order to discover the practices that are the best facilitators of success in step with the inevitable changes in how software is developed.

IN SUMMARY

This article paints its vision of the future with extremely broad brushstrokes and narrowly focuses on those areas where change is expected to be most imminent. Thus, the visions for the future discussed here are not necessarily applicable to all of the software development that occurs at APL. As but one example, consider embedded software. In this case, the end user (a systems developer) is orders of magnitude closer to the 3GL programmer than would be the case with enterprise software, and in fact the systems developer may directly express the software requirements to the programmer. In such cases, there is considerably less return in investing in a multilayered "executable requirements" approach, and the more traditional approaches to the development of embedded software may be more appropriate. Therefore, the future of software development at APL can be expected to consist of a mixture of the contemporary and the futuristic, the implication being that the software development roles at APL will also consist of a mixture of the contemporary and the futuristic.

In the 2003 APL Science and Technology Survey of staff skills and capabilities, 28% of APL staff indicated that they are information processing and management professionals. This high percentage includes systems developers, software developers, software programmers, and individual researchers who develop software to conduct analysis in support of their research. These roles will remain part of APL's software future. However, additional roles will appear in response to the new realities of enterprise software development. We can expect to see new roles such as service ecosystem developer, composite services developer, and so on.

The mixture of roles for any given development project will be a function of the type of software that is being developed (see the box, "Software Development Roles"). Thus, the visions for the future that have been painted here will not benefit all APL software projects uniformly. Enterprise software development will benefit the most in the rapidity with which it can be modified by the end user in response to strategic, tactical, and operational process change. And it is anticipated that process changes resulting in the identification of new service development will be the exception rather than the rule. Therefore, it will only be in the exceptional case that a software development activity will cascade down below the programming environment of the end user. In addition, as the end user will be authoring requirements in a precise, executable form (as opposed to imprecise human language), the human costs of verification and validation can potentially be reduced radically; verification and validation of new functionality would amount to executing new software services and components within the context of the end user's executable requirements and then comparing the results against the end user's simulation results (and so on down the line).

As for more traditional development projects, the benefits will be fewer, but still significant. Even though advancements such as executable requirements may not see use in more traditional development projects, other advancements will (e.g., the use of the HIMS to automate the management of development projects). And certainly, the new practices of enterprise software development will inform the older practices with new insights

SOFTWARE DEVELOPMENT ROLES AT APL

Embedded software component

The development of embedded software presumes the existence of an encapsulating hardware system. The following roles can be expected to play a part:
Systems developer: Synthesizes a design for the overall system and allocates elements of the system design to people, hardware, and embedded software components.
Software programmer: Interprets the requirements that have been allocated to an embedded software component and implements that interpretation.

Monolithic software system

Even in the age of SOA, monolithic software systems will still be necessary. The following roles can be expected to play a part:
Systems developer: Elicits and models the as-is system (if such exists) from the end user, synthesizes a design for the to-be system, and allocates elements of the to-be system design to people and software.
Software developer: Interprets the system-level requirements and synthesizes a design for the software. Allocates software design elements to available software modules (COTS, etc.) and identifies new software components that need to be implemented.
Software programmer: Interprets the requirements that have been allocated to a new software component, synthesizes a design for the component, and implements that design.

Enterprise services software

The elements of an ecosystem of services will be developed independently. The following roles can be expected to play a part:
End user: Programs strategic, tactical, and operational processes, the simulations of which may identify gaps in the service ecosystem as well as the performance requirements for services to fill those gaps.
Service ecosystem developer: Compiles the end-user processes and fleshes them out with logic that aligns them with existing services in the ecosystem as well as the technology that underpins the service ecosystem. Allocates service processing to existing services and identifies new functionality that needs to be implemented. Service orchestration simulations provide performance requirements for the new functionality.
Software developer: Compiles the ecosystem-level service orchestration software and fleshes it out with logic that aligns it with existing software modules (COTS, etc.) as well as the technology that underpins the organization's 3GL development. Allocates service processing to available software modules and identifies new functionality that needs to be implemented. Module interconnection simulations provide performance requirements for the new functionality.
Software programmer: Compiles the service-level module interconnection software and fleshes it out with the specified functionality.

and improved management techniques, just as the older practices have similarly informed other development disciplines.

Our sponsors will continue to rely on the Laboratory for software development, whether embedded within compact guidance systems in the heads of missiles or for prototypes of global information management ecosystems for the intelligence community. However, to maintain this role, APL will need to embrace the emerging paradigms of software development anticipated in this article (with a level of accuracy that is debatable with one exception: *change is inevitable)*.

At the end of the day, the development task process workspace in which Kendall has been working prompts her with options for what to do with this day's work. She selects the "deploy to local test environment," and her work is deployed to a local virtual computer to be automatically tested within a simulated environment in the context of the user's most recent software submission. In the morning, she can review the status of her efforts, as can the project leads via the Monitor Project process workspace. As she packs up for the day, Kendall muses about the stories her father told her about actually developing entire software systems from scratch using nothing but a keyboard, e-mail, and a horribly large and imprecise thing called a "Requirements Document"—and she's amazed that any of it actually worked.

REFERENCE

[1]Harrison-Broninski, K., *Human Interactions: The Heart and Soul of Business Process Management: How People Really Work and How They Can Be Helped to Work Better*, Meghan Kiffer Press, Tampa, FL (2005).

THE AUTHORS

Paul A. Hanke graduated from the University of Delaware in 1993 with a B.S. in electrical engineering and received an M.S. in electrical engineering from the Johns Hopkins University in 1998. Having worked at APL previously, Mr. Hanke rejoined APL in 2001 as a senior software systems engineer for the AISD's Network Management Information Systems group. There, Mr. Hanke contributes to the overall definition and development of enterprise architecture for the DoD's evolving Wideband SATCOM Operational Management System and additionally researches and prototypes advanced capability concepts such as integrated decision support for operations management. **Hilary L. Hershey**, Group Supervisor of the Systems Group of the National Security Technology Department. Ms. Hershey, a member of the APL Principal Professional Staff, has twenty years of experience as a software system engineer. She was a key member in establishing standard software engineering practices in NSTD beginning in the mid-1990s. She has previously chaired the Software Engineering Process Group and was the first head of the Software Engineering Standards Subcommittee of the Quality Council. She has a B.S. in computer science and statistics from American University and a M.S. in computer science with a software engineering focus from Johns Hopkins University. A member of the IEEE and ACM, she began employment at APL as a resident subcontractor with Sachs-Freeman Associates in 1987 and converted to APL staff in 1992. Prior to joining APL, she worked as a software engineer at Vector Research Company, Inc. and the Armed Forces Radiobiological Research Institute. **Pamela A. Smith** (not pictured), Branch Head of the Applied Security and Protection Technology Branch in NSTD, had 20 years of experience as a software developer before transitioning to management. She led the first and only APL group that achieved Capability Maturity Model (CMM)-Software Level 2 successful assessment. In 2002, she served as the first head of APL's Software Engineering Process Group. Pam has taught Software Engineering Management at the JHU Whiting School of Engineering. She has a B.S. in Chemical Engineering from the Pennsylvania State University and a M.S. in Computer Science from the Johns Hopkins University. She began employment at APL in 1996. Prior to coming to APL, she worked as a project manager and software engineer at Vector Research Company, Inc. and prior to that she worked as an Associate Research Engineer at the U.S. Steel Research laboratory. For further information contact Paul Hanke at paul.hanke@jhuapl.edu.

Paul A. Hanke

Hilary L. Hershey