TRW Systems Engineering &
Development Division

TRW-TS-90-01

TRW

1

TRW's Ada Process Model for Incremental Development of Large Software Systems

Walker Royce

January 1990

TRW Technology Series

TRW Technology Series

TRW Technology Series

TRW Technology Series

TRW Technology Series

TRW Technology Series

TRW Technology Series

TRW Technology Series

chnology Series

TRW's Ada Process Model for Incremental Development of Large Software Systems

Walker Royce

TRW Systems Integration Group
Space & Defense Sector
Redondo Beach, CA 90278

ABSTRACT

TRW's Ada Process Model has proven to be key to the Command Center Processing and Display System-Replacement (CCPDS-R) project's success to date in developing over 300,000 lines of Ada source code executing in a distributed VAX VMS environment.

The Ada Process Model is, in simplest terms, a uniform application of incremental development coupled with a demonstration-based approach to design review for continuous and insightful thread testing and risk management. The use of Ada as the life-cycle language for design evolution provides the vehicle for uniformity and a basis for consistent software progress metrics. This paper provides an overview of the techniques and benefits of the Ada Process Model and describes some of the experience and lessons learned to date.

Project Background

The Command Center Processing and Display System-Replacement (CCPDS-R) project will provide display information used during emergency conferences by the National Command Authorities; Chairman, Joint Chiefs of Staff; Commander in Chief, North American Aerospace Command; Commander in Chief, United States Space Command; Commander in Chief, Strategic Air Command, and other nuclear-capable commanders in chief. It is the missile warning element of the new Integrated Tactical Warning/Attack Assessment system architecture developed by North American Aerospace Defense Command/Air Force Space Command.

The CCPDS-R project is being procured by Air Force Systems Command Headquarters, Electronic Systems Division (ESD) at Hanscom AFB and was awarded to TRW Defense Systems Group in June 1987. TRW will build three subsystems. The first, identified as the Common Subsystem, is 27 months into development. The Common Subsystem consists of over 300,000 source lines of Ada with a development schedule of 38 months. It will be a highly reliable, real-time distributed system with a sophisticated user interface and stringent performance requirements implemented entirely in Ada. CCPDS-R Ada risks were originally a very serious concern. At the time of contract definition, Ada host and target environments and Ada-trained personnel availability were questionable.

The genesis of the Ada Process Model was a TRW Independent Research and Development project which pioneered the technology from 1984-1987 and provided the key software personnel for CCPDS-R startup.

The CCPDS-R architecture consists of approximately 300 tasks executing in a network of *10 VAX family* processors with over 1,000 task-to-task software interfaces. This large distributed network was developed primarily on the VAX Ada environment augmented with a Rational R1000 host. Currently, CCPDS-R is immersed in formal testing of most of the software builds with the last build still in development. To date, the software effort (75 people) has flowed smoothly on schedule and on budget.

Conventional Software Process Shortfalls

All large software products require multiple people to *converge* individual ideas into a single solution for a *vaguely* stated problem. The real difficulties in this process are twofold:

1. *Convergence of individual solutions* into an integrated product implies that multiple people must communicate capably.
2. *Vaguely stated problems* stem from the use of ambiguous human communication techniques (e.g., English) as well as numerous unknown criteria at the time of requirements definition.

Conventional methods of software engineering focused on explicit separation of the problem (requirements), the abstract solution (design), and the final product (code and documentation). No single representation format was suitable for requirements and design, or design and code, or all three. The tools supporting these phases were very different, and the cost of change increased exponentially from one phase to the next. This lack of evolving flexibility forces the industry into a mode of "perfecting" one phase prior to the next; hence, no design was permitted until requirements were baselined and no coding was permitted until the design was baselined, etc.

The inherent problem with different representation formats for the products of each phase is that translation, interpretation, and communication in transitioning to the next phase is very error prone. "Representers" were frequently not the "translators" throughout the life cycle and basic intentions were often corrupted. Furthermore, the evaluation of each intermediate phase was typically based on paper review, simulation, and for the most part, engineering judgment and conjecture. Given a complex software system, there are far too many subtle interactions, miscommunications, and complex relationships to predictably achieve quality design verification without actually building subsets of the product and getting *factual* feedback. The "real" evaluation of goodness occurred very late in conventional programs when components were integrated and executed in the target environment *together* for the first time. This usually resulted in excessive rework and caused late "shoehorning" of less than desirable solutions into the final product. These late, reactive changes resulted in added fragility and reduced product quality.

Figure 1 identifies the result of a typical conventional project when integration of components is delayed until late in the life cycle: substantial rework. The figure plots "Development Progress" against schedule. Development progress here is defined to be the percentage of the software product coded, compiled, and informally tested in its target language (i.e., demonstrable). Although conventional projects operated under the guise "no coding prior to CDR," we have displayed the conventional project's development progress assuming some standalone prototypes are done prior to CDR and that much of conventional Program Design Language (PDL) is directly translatable into the target language (i.e., demonstrable). In the figure, the conventional project is characterized by: an early PDR supported by small prototypes and a foreign PDL, no substantial coding until after CDR, risk management by conjecture, paper design reviews, protracted integration, and unlikely adherence to the planned completion schedule.

CCPDS-R is characterized in Figure 1 by a later PDR with a more tangible definition of what constitutes a preliminary design, demonstrations of incremental capabilities coupled with each design review, continuous integration during the design phase rather than the test phase, systematic risk management based on factual early feedback, and a higher probability of meeting an end-item schedule with a higher quality product.

Ada Process Model Goals

TRW's Ada Process Model recognizes that all large, complex software systems will suffer from design breakage due to early unknowns. It strives to accelerate the resolution of unknowns and correction of design flaws in a systematic fashion which permits prioritized management of risks. *The dominant mechanism for achieving this goal is a disciplined approach to incremental development.* The key strategies inherent in this approach are directly aimed at the three main contributors to software diseconomy of scale: minimizing the overhead and inaccuracy of interpersonal communications, eliminating rework and converging requirements stability as quickly as possible in the life cycle. These objectives are achieved by:

1. Requiring continuous and early convergence of individual solutions in a homogeneous life-cycle language (Ada).

2. Eliminating ambiguities and unknowns in the problem statement and the evolving solution as rapidly as practical through prioritized development of tangible increments of capability.

Although many of the disciplines and techniques presented herein can be applied to non-Ada projects, the expressiveness of Ada as a design and implementation language and support for partial implementation (abstraction) provide a strong platform for implementing an effective, uniform approach.

Many of the Ada Process Model strategies (summarized in Figure 2) have been attempted, in part, on other software development efforts; however, there are fundamental differences in this approach with respect to conventional software development models.

Uniform Ada Life-Cycle Representation

The primary innovation in the Ada Process Model is the use of a single language for the entire software life cycle, including, to some degree, the requirements phase. All of the remaining techniques rely on the ability to equate design with code so that the only variable during development is the level of abstraction. This provides two essential benefits:

1. *The ability to quantify units of software (design/development/test) work in* **one dimension,** *Source Lines of Code (SLOC).* While it is certainly true that SLOC is not a perfect

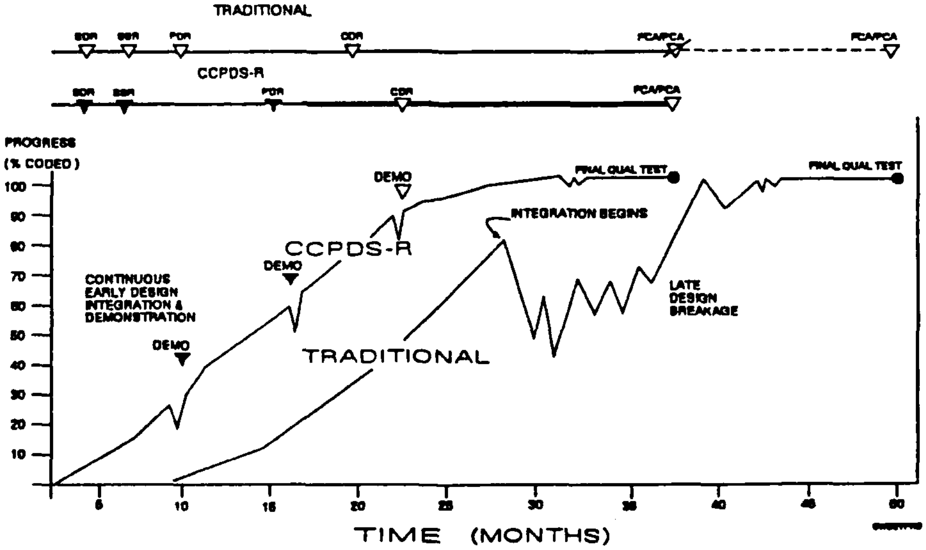

Figure 1. Software Development Progress

Process Model Strategy		Conventional Counterpart
Uniform Ada Lifecycle Representation	⟹	PDL/HOL
Incremental Development	⟹	Monolithic Development
Design Integration	⟹	Integration and Test
Demonstration Based Design Review	⟹	Documentation Based Design Review
Total Quality Management	⟹	Quality by Inspection

Figure 2. New Techniques vs. Conventional Techniques

absolute measure of software, with consistent counting rules [3], it has proven to be the best normalized measure and provides an objective, consistent basis for assessing relative trends across the project life cycle.

2. *A formal syntax and semantics for life-cycle representation with automated verification by an Ada compiler.* Ada compilation does not provide complete verification of a component. It does go a long way, however, in verifying configuration consistency and ensuring a standard, unambiguous representation.

Incremental Development

Although risk management through incremental development is emphasized as a key strategy of the Ada Process Model, it was (or always should have been) a key part of most conventional models. Without a uniform life-cycle language as a vehicle for incremental design/code/test, conventional implementations of incremental development were difficult to manage. This management is highly simplified by the integrated techniques of the Ada Process Model [8].

Design Integration

In this discussion, we will take a simpleminded view of "design" as the partitioning of software components (in terms of function and performance) and definition of their interfaces. At the highest level of design we could be talking about conventional requirements definition; at the lowest level, we are talking about conventional coding. Implementation is then the development of these components to meet their interface while providing the necessary functional performance. *Regardless of level, the activity being performed is Ada coding.* Top-level design means coding the top-level components (Ada main programs, task executives, global types, global objects, top-level library units, etc.). Lower-level design means coding the lower-level program unit specifications and bodies.

The postponement of all coding until after CDR in conventional software development approaches also postponed the primary indicator of design quality: integrability of the interfaces. The Ada Process Model requires the early development of a Software Architecture Skeleton (SAS) as a vehicle for early interface definition. The SAS essentially corresponds to coding the top-level components and their interfaces, compiling them, and providing adequate drivers/stubs so that they can be executed. This early development forces early baselining of the software interfaces to best effect smooth development, evaluate design quality early, and avoid/control downstream breakage. In this process, we have made integration a design activity rather than a test activity. To a large degree, the Ada language forces integration through its library rules and consistency of compiled components. It also supports the concept of separating structural design (specifications) from runtime function (bodies). The Ada Process Model expands this concept by requiring structural design (SAS) prior to runtime function (executable threads). Demonstrations provide a forcing function for broader runtime integration to augment the compile time integration enforced by the Ada language.

Demonstration-Based Design Review

Many conventional projects built demonstrations or benchmarks of standalone design issues (e.g., user system interface, critical algorithms, etc.) to support design feasibility. However, the design baseline was represented on paper (PDL, simulations, flowcharts, vugraphs). These representations were vague, ambiguous and not amenable to configuration control. The degree of freedom in the design representations made it very difficult to uncover design flaws of substance, especially for complex systems with concurrent processing. Given the typical design review attitude that a design is "innocent until proven guilty," it was quite easy to assert that the design was adequate. This was primarily due to the lack of a tangible design representation from which true design flaws were unambiguously obvious. Under the Ada Process Model, design review demonstrations provide some proof of innocence and are far more *efficient* at identifying and resolving design flaws. The subject of the design review is not only a briefing which describes the design in human understandable terms but also a demonstration of important aspects of the design *baseline* which verify design quality (or lack of quality).

Total Quality Management

In the Ada Process Model there are two key advantages for applying TQM. The first is the common Ada format throughout the life cycle, which permits consistent software metrics across the software development work force. Although these metrics do not all pertain to quality (many pertain to progress), they do permit a uniform communications vehicle for achieving the desired quality in an efficient manner.

Secondly, the demonstrations serve to provide a common goal for the software developers. This "integrated product" is a reflection of the complete design at various phases in the life cycle for which all personnel have ownership. Rather than individually evaluating components which are owned by individuals, the demonstrations provide a mechanism for reviewing the team's product. This team ownership of the demonstrations is an important motivation for instilling a TQM attitude.

Incremental Development

Incremental development is a well-known software engineering technique which has been used for many years [1],[2]. The Ada

Process Model simply extends the discipline of incremental development into three dimensions:

Subsystem Increments, called *builds*, are selected subsets of software capability which implement a specific risk management plan. These increments represent a cross section of components which provide a demonstrable thread of capability. Integration across builds is mechanized by constructing major milestone (SSR, PDR, CDR) demonstrations of capabilities which span multiple builds.

Build Increments, called *design walkthroughs*, are sets of partially implemented components within a build which permit evolutionary insight into the allocated build components' structure, operation, and performance as an integrated set. Integration of components within builds is mechanized by constructing small scale design walkthrough (PDW and CDW) demonstrations composed of capabilities which span multiple components.

Component Increments, called *Ada Design Language*, or ADL, are partial implementations of Ada program units maintained in a compilable Ada format with placeholders for pending design detail. Integration of Ada program units is enforced to a large degree through compilation.

The paramount advantage of Ada in supporting incremental development is its support for partial implementations. Separation of specifications and bodies, packages, powerful data typing, and Ada's expressiveness and readability are features which can be exploited to provide an effective development approach and insightful development progress metrics for continuous assessment of project health from multiple perspectives. These development progress metrics are described later in this paper and in [8].

Figure 3 is an overview of a generic definition of project "builds" for insightful risk management. This definition has been abstracted from CCPDS-R experience where a similar build content/schedule has been extremely successful. The key features of the proposed build definition are:

Risk Management

Planning the content and schedule for each of the builds is perhaps the first and foremost risk management task. This activity essentially will define the risk management plan for the project. The importance of a good build content and schedule plan cannot be overemphasized, the efficiency of the software development depends on its initial quality *and the ability for the*

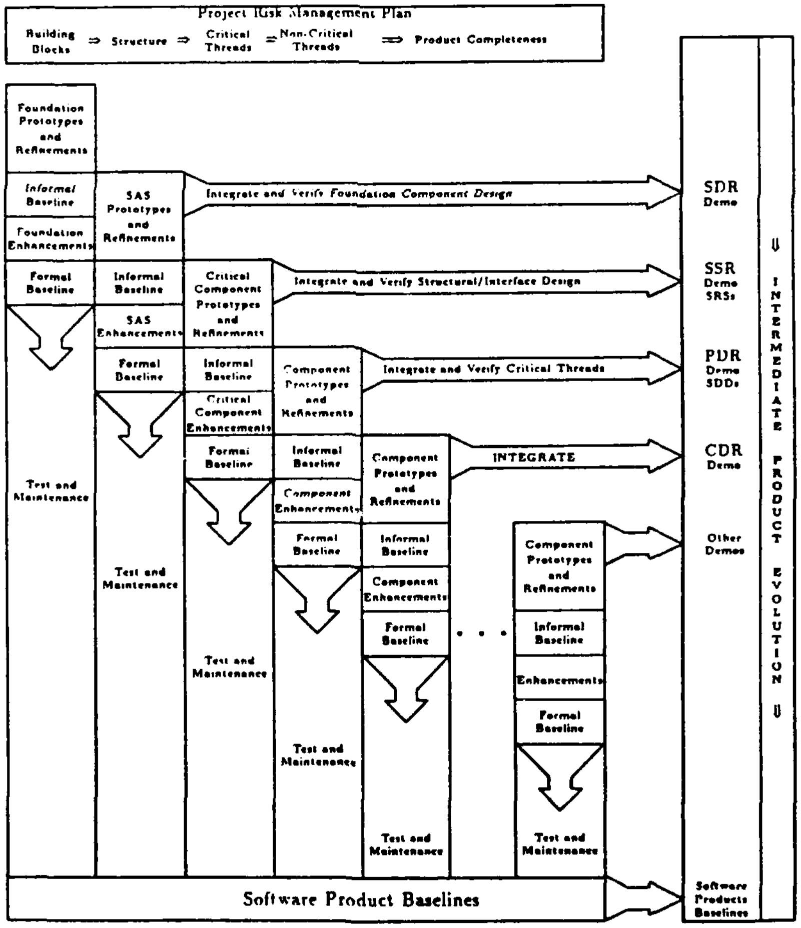
Project Risk Management Plan
Building Blocks ⇒ Structure ⇒ Critical Threads ⇒ Non-Critical Threads ⇒ Product Completeness
Foundation Prototypes and Refinements
Informal Baseline
Foundation Enhancements
Formal Baseline
Test and Maintenance
SAS Prototypes and Refinements
Informal Baseline
SAS Enhancements
Formal Baseline
Test and Maintenance
Critical Component Prototypes and Refinements
Informal Baseline
Critical Component Enhancements
Formal Baseline
Test and Maintenance
Component Prototypes and Refinements
Informal Baseline
Component Enhancements
Formal Baseline
Test and Maintenance
Component Prototypes and Refinements
Informal Baseline
Component Enhancements
Formal Baseline
Test and Maintenance
Component Prototypes and Refinements
Informal Baseline
Enhancements
Formal Baseline
Test and Maintenance
Integrate and Verify Foundation Component Design
Integrate and Verify Structural/Interface Design
Integrate and Verify Critical Threads
INTEGRATE
SDR Demo
SSR Demo SRSs
PDR Demo SDDs
CDR Demo
Other Demos
Software Product Baselines
Software Products Baselines
⇓ INTERMEDIATE PRODUCT EVOLUTION ⇓

Figure 3. Incremental Development under the Ada Process Model

project to react to changes as the development progresses. The last point is important (as we have learned on CCPDS-R) because of the need to adjust build content and schedule as more accurate assessments of complexity, risk, personnel, and value engineering are achieved. Applying the underlying theory of this process model, the incremental development of the risk management plan must also provide some flexibility during the early builds as conjecture starts evolving into fact.

Pioneering an Early Build

The first build is identified in the figure as the foundation components. This build is critical to the Ada Process Model for two reasons: it must provide the prerequisite components for smooth development as well as a vehicle for early prototyping of the development process itself. There are always components which can be deemed "mission requirements independent" that fall into the category of foundation components.

The foundation software build represents components which are likely to be depended on by large numbers of development personnel. Their early availability, usage, feedback, and stabilization represent a key asset in avoiding downstream breakage and rework.

The second purpose of this build is to provide a guinea pig for exercising the process to be used for the subsequent mainstream builds. The existence of this early pioneer build on CCPDS-R was key to uncovering inefficiencies in the standards, procedures, tools, and techniques in a small scale where breakage was controllable and incorporating improvements and lessons learned in time to support later builds more efficiently. This "incremental development" of the process itself is important until the process matures into a truly reusable state itself.

Requirements/Design Concurrence

An interesting aspect of the proposed build sequence is the concurrency of requirements definition and the Software Architecture Skeleton (SAS) build. This concurrency ensures design/requirements consistency at the interface level described in the Software Requirements Specifications (SRS). In general, we are prescribing that this level be the set of interfaces which are inherent in the Software Architecture Skeleton. The purpose of this concurrency is twofold: to ensure that the requirements are validated (to some degree) by a design representation and to eliminate ambiguities and identify holes in the requirements through experience in constructing the candidate solution's structure. This approach explicitly admits that the difference between requirements and design is a subtle one. The extent to which a project defines its software requirements will depend on both contractor and customer desires. The purpose of this approach is not to define that extent, but rather to provide enough early information in both the design and requirements representations to converge on an SRS baseline which is suitable to both parties. This subject is treated further in [5].

Software Architecture Skeleton (SAS)

The concept of a software architecture skeleton (SAS) is fundamental to effective evolutionary development. Although different applications domains may define the SAS differently, it should encompass *the declarative view of the solution which identifies all top-level executable components (Ada Main Programs and Tasks), all control interfaces between these components, and all type definitions for data interfaces between these components.* Although a SAS should compile, it will not necessarily execute without software which provides data stimuli and responses. The purpose of the SAS is to provide the structure/interface baseline environment for integrating evolving components into demonstrations. The definition of the SAS (Figure 4) represents the forum for interface evolution between components. In essence, a SAS provides only software potential energy: a framework to execute and a definition of the stimulus/response communications network. Software work is only performed when stimuli are provided along with applications components which transform stimuli into responses. If an explicit subset of stimuli and applications components are provided, a system thread can be made visible. The incremental selection of stimuli and applications components constitutes the basis of the build a little, test a little approach of the Ada Process Model. It is important to construct a candidate SAS early, evolve it into a stable baseline, and continue to enhance, augment, and maintain the SAS as the remaining design evolves.

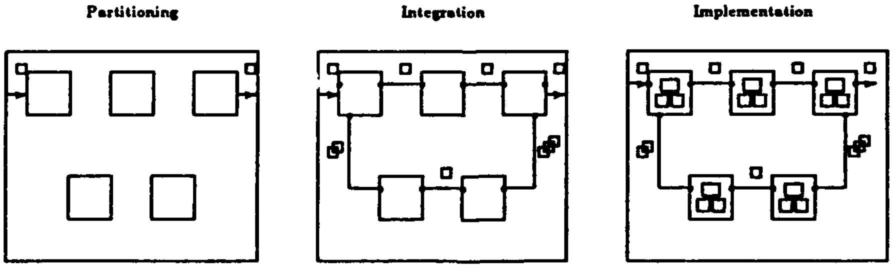

Figure 4. SAS Definition

In Figure 4, the symbols could take on different meanings depending on the object being defined. For example, the larger boxes could represent Ada main programs, tasks, or subprograms. The smaller boxes tagged to interface lines could represent types, packages of types, or other objects. The important message of the figure is that all SAS objects evolve from partitioning (existence of processing objects) to integration (definition of inter-object behavior) to implementation (completion of all object abstractions) in a compilable Ada format. This evolution should occur in a combination of breadth-first and depth-first design activities which best match the risk management plan of the project.

The PDR Milestone

In Figure 3, the Preliminary Design Review (PDR) takes on a different meaning than in conventional process models. In the context of the Ada Process Model, PDR is the review and formal baselining of the SAS. The PDR reviews this candidate baseline with a supporting demonstration of some capability subset exercising the SAS to the extent that the structure can be deemed an acceptable baseline. The definition of the capability subset to be demonstrated at PDR is application dependent, but certainly should include the potential performance drivers (in space, time, throughput, or whatever), risky functional implementations, and design breakage drivers (i.e., components which are "withed" by many other components). With the exception of the SAS, PDR has been intentionally defined generically, to be instantiated appropriately for each project.

Conventional software PDRs define standards for review topics that result in tremendous breadth of review, with only a minimal amount that is really important or understood by the large, diverse audience. Reviewing all requirements in equal detail at a PDR is inefficient and unproductive. Not all requirements are created equal; some are critical to design evolution, some are don't cares. The Ada Process Model attempts to improve the effectiveness of design review by allocating the technical software review to smaller scale design walkthroughs and focusing the major milestone reviews on the globally important design issues. Furthermore, focusing the design review on a demonstration provides a more understandable representation of design perspectives for the diverse PDR audience (procurement, user, technical assistance personnel), most of whom are not familiar with Ada or detailed software engineering tradeoffs.

Demonstrations as Primary Products

Traditional software developments under the current Military Standards focus on documentation as intermediate products. To some extent, this is certainly useful and necessary [6]. However, by itself it is inadequate for large systems. Fundamental in the Ada Process Model is forcing design review to be more tangible via visibly demonstrated capabilities. These demonstrations serve two key objectives:

1. The generation/integration of the demonstration provides tangible feedback on integrability, flexibility, performance, interface semantics, and identification of design and requirements unknowns. It satisfies the software designer/developer by providing first hand knowledge of the impact of individual design decisions and their usage/interpretation by others. *The generation of the demonstration is the real design review.* This activity has proven to provide the highest return on investment in CCPDS-R by uncovering design interface deficiencies early.

2. The finished demonstration provides the monitors of the development activity (users, managers, customers, and other indirectly involved engineering performers) tangible insight into functionality, performance, and development progress. One sees an executing Ada implementation of understandable and relevant capability subsets.

On CCPDS-R, lessons learned from informal design walkthrough demonstrations are tracked via action items. For major milestone demonstrations, a demonstration plan is developed which identifies the capabilities planned to be demonstrated, how the capabilities will be observed, and explicit pass/fail criteria. Pass/fail criteria should be defined as a threshold for generating an action item; they need not be derived directly from requirements. The pass/fail criteria should trigger an action based on exceeding a certain threshold of concern as negotiated by the responsible engineering authorities for both contractor and customer.

With early demonstrations (whether on the target hardware or host environment), significantly more accurate assessments of performance issues can be obtained and resolved *inexpensively*. An important difference in the Ada Process Model, however, is that these demonstrations may tend to start out showing performance issues as immature designs are assessed. Whereas traditional projects started out with optimistic assessments that matured into problems, this approach may start pessimistic and mature into solutions.

Build Chronology

The individual milestones within a build need to be integrated into (or flowed down from) the higher-level project milestones. There is a tradeoff between the number of builds and the integration of concurrent build activities into a higher-level milestone plan. In general, an increased number of builds provides for more detailed development risk control. However, it also increases the management overhead and complexity of scheduling concurrent builds, managing personnel, and supporting an increased number of design walkthroughs. For any given project, these criteria need to be carefully evaluated. As general lessons learned, CCPDS-R defined small but complex early builds where detailed technical control and insight were necessary, and larger, less complex, later builds where the management focus was on production volume rather than on technical risks.

Within an individual build (Figure 5), a well-defined sequence of design walkthroughs takes place, Design walkthroughs are informal, detailed technical peer reviews of intermediate design products with attendance by interested reviewers including other designers, testers, QA, and customer personnel. Typically, the audience is restricted to a small knowledgeable group. However, other attendance is useful for the purposes of training. On CCPDS-R, design walkthrough attendance averaged 20-30 people. Design walkthroughs must be informal and highly interactive with open critique. A contractor or customer who forces too much formality into the walkthrough process will slow down the design evolution, or stifle the openness of raising issues; both are counterproductive (e.g., dry runs should be unnecessary). Likewise, followthrough must be effective and timely to maintain development continuity and capture issue resolution. Design walkthrough standards for format and content are needed early and should be evolved with experience.

Initial prototyping and design work is iterated primarily for the purpose of presenting a Preliminary Design Walkthrough (PDW) and associated capability demonstration. The focus of the PDW should be on reviewing the structure of the components which make up the build., with the demonstration focused on integrated components in an environment which exercises intercomponent interfaces in a visible manner. In simple terms, the PDW is focusing on a review of the declarative view of the components with enough executable capability to evaluate the goodness of the structure and interfaces.

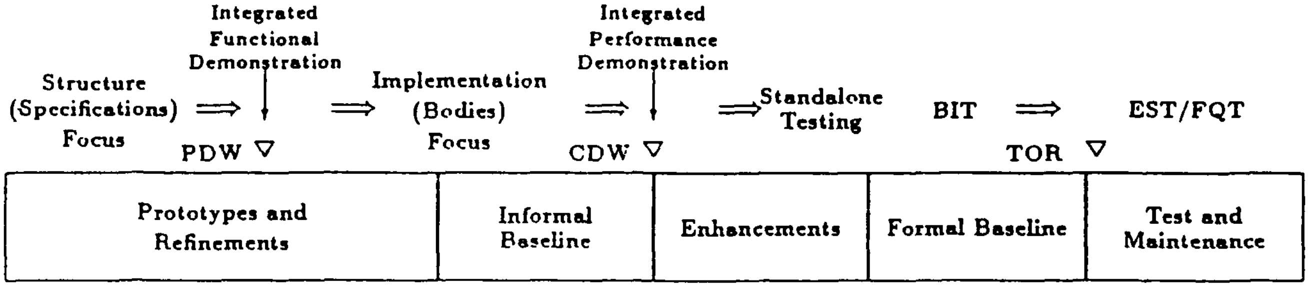
Structure (Specifications) Focus
Integrated Functional Demonstration
PDW
Implementation (Bodies) Focus
Integrated Performance Demonstration
CDW
Standalone Testing
BIT
TOR
EST/FQT
Prototypes and Refinements
Informal Baseline
Enhancements
Formal Baseline
Test and Maintenance

Figure 5. Incremental Development within a Build

Following the PDW, design lessons learned from both the PDW and the demonstration (tracked via informal action items on CCPDS-R) are incorporated into the evolving components, and further refinements of the design are performed en route to a Critical Design Walkthrough (CDW). The focus of the CDW should be on reviewing the operation of the components which make up the build, with the demonstration focused on integrated components in an environment which exercises intercomponent performance in a visible manner. The CDW is focusing on a review of the executable view of the components with enough capability to evaluate the performance of the *integrated* component subset. CDW demonstrations should be defined so that the important aspects of execution performance are visible (e.g., space, response time, accuracy, throughput, etc.), and the subset of partial implementations provides coverage of the components expected to be drivers in any performance criteria of interest.

The next phase is to incorporate the action items from CDW and the CDW demonstration's lessons learned into the build components, and refine and enhance those components into complete implementations. These components are then standalone tested for complete coverage of allocated requirements and boundary corditions in preparation for delivery to an independent test team. These informal tests perform the lowest levels of *integrated test* so that complete library units are turned over for formal requirements verification tests. Note the use of the term "integrated test" which implies that most integration has occurred in the process of providing working demonstrations. One still must account for some future integration effort, however, since incorporation of action items and demonstration lessons learned could result in interface breakage which must be re-integrated.

The final activity is the process of software turnover where the standalone tested units are placed under configuration control. These activities must be carefully planned so that the intercomponent dependencies within a build are accommodated in the turnover sequence. The build integration test phase (described below) is conducted during the turnover activity as an informal verification of configuration consistency, interface integrity, and standalone test completeness. These activities are logically just regression tests of previous demonstrations to ensure that all previously identified issues have been rectified and previously untested interfaces are accounted for prior to formal testing.

Incremental Test

Another benefit of this process model is the partitioning of the software into manageable increments for software test with planned software maintenance time. Although substantial informal testing occurs as a natural by-product of demonstration development, it is by no means complete, nor is it intended to demonstrate requirements satisfaction. Software test under this Process Model includes:

SAT. Standalone Test is an integrated set of Ada program units (typically a complete library unit) tested in a standalone environment. This level of testing corresponds to completeness and boundary condition testing to the extent possible in a standalone environment prior to delivery as a configuration baseline. Most SATs are informal in that they do not verify requirements; however, requirements verification in a standalone test (e.g., algorithm accuracy) can be performed after the components are installed into a controlled configuration baseline.

BIT. Where conventional projects typically suffer extensive design breakage during integration, build integration testing in the Ada Process Model is straightforward and rapid. It is essentially a regression test of previous demonstrations which should have resolved the major interface issues. It is still a necessary step, however, since it corresponds to the "Quality Evaluation" associated with installation of a standalone tested configuration baseline. BIT serves to validate that the previously demonstrated threads can be repeated, that previously defined deficiencies have been fixed, and that the configuration installation does not break any previous configurations. Furthermore, it does provide completeness of interface exercise not provided in the demonstrations.

EST. Engineering String Tests represent test cases which are focused on verifying specific subsets of requirements from possibly multiple CSCIs through demonstration and test of capability threads.

FQT. Formal Qualification Tests require a complete software subsystem for requirements verification. For example, a requirement such as "50% reserve capacity" cannot be verified until all components are present. The subject of this incremental test approach is treated more fully in [7].

Ada as a Design Language

The use of compilable Ada as a Design Language (ADL) is one of the primary facets of the Ada Process Model which provides uniformity of representation format. As will be described in the next section, the uniformity and Ada/ADL standards permit insight via software development progress metrics. Note that the terms Ada and ADL are virtually interchangeable with respect to our usage standards; the standards which apply to our final Ada products are the same as those that apply to earlier design representations. This approach best supports our technique of *evolving* designs into implementations without translating between two sets of standards.

The foundation of Ada Design Language is the use of Ada, comments, and predefined TBD Ada objects to evolve a continuously compilable design representation from a high-level abstraction into a complete Ada implementation. *ADL statements* are Ada statements which contain placeholders from the predefined set of TBD Ada objects. A package called **TBD_Types** defines TBD types, TBD constants, TBD values, and a TBD procedure for depicting statement counts associated with comments which together act as placeholders for TBD processing.

The use of objects from package **TBD_Types** constitutes a program unit or statement being identified as "ADL" (i.e., incomplete). Program units or statements which have no references to TBD objects are identified as "Ada" (i.e., complete) for the purposes of metrics collection.

Figure 6 shows a typical transition of ADL to Ada for a specific library unit from CCPDS-R. Although this evolution identifies a fairly uniform transition to Ada across all program units at PDW and CDW, a more typical evolution would have more individual program units being 100 percent complete while others being essentially zero percent complete. In other words, most designers on CCPDS-R tend to focus on completing sets of individual program units rather than portions of all program units. Given the flexibility of Ada packaging and structure, either technique has proven to be equally useful and should be left as designer's choice. However, the general transition trend has still been approximately 30 percent done by PDW, 70 percent done by CDW, and 100 percent done by Turnover.

	Program Unit	Type	Ada	ADL	Cplx	Total	%
Initial View	Inm_Erm_Procedures	TLCSC	6	122	4.0	128	4.7
		Package	2	122	4.0	124	1.6
	Create_Inm_Erm_Circuits	Proc	1	0	3.0	1	100.0
	Perform_Reconfiguration	Proc	1	0	4.0	1	100.0
	Perform_Shutdown	Proc	1	0	3.0	1	100.0
	Process_Error_Messages	Proc	1	0	4.0	1	100.0
PDW View	Inm_Erm_Procedres	TLCSC	47	101	3.9	148	31.8
		Package	24	19	4.0	43	55.8
	All_Node_Connections	Proc	3	19	4.0	22	13.6
	Create_Inm_Erm_Circuits	Proc	4	8	3.0	12	33.3
	On_Node_Connections	Proc	3	7	4.0	10	30.0
	Perform_Reconfiguration	Proc	6	2	4.0	8	75.0
	Perform_Shutdown	Proc	4	3	3.0	7	57.1
	Process_Error_Messages	Proc	3	43	4.0	46	6.5
CDW View	Inm_Erm_Procedures	TLCSC	87	48	3.9	135	64.4
		Package	30	11	4.0	41	73.2
	All_Node_Connections	Proc	16	0	4.0	16	100.0
	Create_Inm_Erm_Circuits	Proc	8	4	3.0	12	66.7
	On_Node_Connections	Proc	9	0	4.0	9	100.0
	Perform_Reconfiguration	Proc	6	2	4.0	8	75.0
	Perform_Shutdown	Proc	6	1	3.0	7	85.7
	Process_Error_Messages	Proc	12	30	4.0	42	28.6
Turnover View	Inm_Erm_Procedures	TLCSC	137	0	3.9	137	100.0
		Package	42	0	4.0	42	100.0
	All_Node_Connections	Proc	16	0	4.0	16	100.0
	Create_Inm_Erm_Circuits	Proc	12	0	3.0	12	100.0
	On_Node_Connections	Proc	9	0	4.0	9	100.0
	Perform_Reconfiguration	Proc	8	0	4.0	8	100.0
	Perform_Shutdown	Proc	7	0	3.0	7	100.0
	Process_Error_Messages	Proc	43	0	4.0	43	100.0

Figure 6. Incremental Development within a Component: ADL

Software Metrics

One of the by-products of our definition of ADL is the uniform representation of the design with a complete estimate of the work accomplished (source lines of Ada) and the work pending (source lines of ADL) embedded in the evolving source files in a compilable format. Although the Ada source lines are not necessarily complete (further design evolution may cause change), they do represent an accurate assessment of work accomplished. Given this life-cycle standard format, the complete set of design files can be processed at any point in time to gain insight into development progress. On CCPDS-R, a metrics tool was developed which scans Ada/ADL source files and compiles statistics automatically. Figure 7 identifies example statistics which are produced by CSCI, by CSCI build, by build, and by subsystem. The metrics collection process is performed monthly for detailed management insight into development progress, code growth, and other indicators of potential problems [8].

The monthly metrics are collected by build, and by CSCI so that high-level trends and individual contributions can be assessed. Individual CSCI managers collect their metrics and assess their situation prior to incorporation into project-level views and are held accountable for monthly explanations of anomalous circumstances.

NAS CSCI Metrics (Month 10)			
	Designed	Coded	Total
TLCSCs	39	33	40
LLCSCs	13	10	13
Units	484	459	494

	ADL	Ada	Total
SLOC	1858	16636	18494
% Coded			90.0%
CPLX			3.8

Figure 7. Summary Metrics for CSCI

This process provides objectivity, consistency, and insight into progress assessment. Although the lowest level assessments of ADL statements are certainly still somewhat subjective, they are determined by the most knowledgeable people, the designers, and are therefore more likely to be accurate. Furthermore, the CSCI manager assesses his own situation based on the combined influence of his design team communicating to him in a uniform format. This consistency is even more valuable to the software management team since all of the responsible managers are communicating progress in the same language and the assessment of CSCI progress, Build progress, and Subsystem progress are also consistent. The software management team can assess trends early, identify potential problems early, and communicate with higher-level management and customer personnel in an objective manner. This definition of development progress makes the development effort more tangible so that decision making and management can be based on fact rather than conjecture.

CCPDS-R Experience

At the time of this writing CCPDS-R is in month 27 of development. Five out of six builds have been developed and are currently maintained in a configuration-controlled baseline. This represents about 97 percent of the total Common Subsystem product being developed, integrated, and standalone tested. We are currently immersed in the formal requirements verification activities. As displayed in Figure 8, the design and development has flowed smoothly *as planned*, and there is a high probability of finishing on schedule, and on budget. The program is currently about 3 months past "System CDR." This is misleading, however, since contractually we were required to hold CDR after our final build's CDW. The success of CCPDS-R to date under the Ada Process Model described in this paper has not been easily attained. There has been high commitment on the part of the CCPDS-R project team, the CCPDS-R customer, and TRW management to make sure that the process was followed and that lessons learned were incorporated along the way.

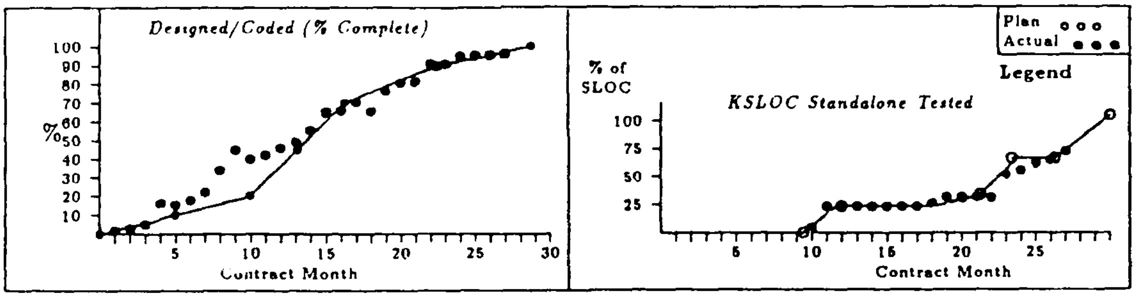

Figure 8. Common Subsystem Development Progress

Summary

Under careful scrutiny by managers, customers, developers, and testers, this Ada Process Model has matured into a powerful technique embraced by all levels. Developers and designers prefer working in the Ada language with demonstrations as products. Managers and customers appreciate the tangible insight into progress and technical risk evaluation and reduction. Testers have benefited from early involvement, better intermediate products, and the ability to focus on testing rather than integration. But most of all, users should ultimately receive a better product on schedule for a reasonable cost.

In the face of early Ada risks (compiler maturity, trained personnel, etc.), the CCPDS-R software development has been exemplary. We have continuously resolved issues as early as possible in the life cycle and the likelihood of confronting any serious problems in the future seems small. All in all, CCPDS-R represents an outstanding first generation Ada project from which many lessons have been learned. Second generation projects should be better.

Acknowledgments

The success of the Ada Process Model on CCPDS-R to date is due to multiple contributions including the TRW Systems Engineering & Development Division's management commitment to follow through on a risky new technology insertion with strong support as well as the entire CCPDS-R Software and Systems Engineering team. Explicit acknowledgments are due to Don Andres, Joan Bebb, Chase Dane, Charles Grauling, Tom Herman, Bruce Kohl, Steve Patay, Patti Shishido, and Mike Springman, whose day to day involvement and commitment have transformed some common sense ideas into tangible successes.

Biography

Walker Royce is the Software Chief Engineer on the CCPDS-R Project. He received his BA in Physics at the University of California, Berkeley, in 1977; an MS in Computer Information and Control Engineering at the University of Michigan in 1978, and has an additional 3 years of post-graduate study in Computer Science at UCLA. Mr. Royce has been at TRW for 10 years, dedicating the last five years to advancing Ada technologies in support of CCPDS-R. He served as the principal investigator of SEDD's Ada Applicability for C^3 Systems Independent Research and Development Project from 1984-1987. This IR&D project resulted in the Ada Process Model and the Network Architecture Services Software, technologies which earned TRW's Chairman's Award for Innovation and have since been transitioned from research into practice on real projects.

REFERENCES

[1] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[2] B.W. Boehm, "The Spiral Model of Software Development and Enhancement," *Proceedings of the International Workshop on the Software Process and Software Environments*, Coto de Caza, CA, March 1985.

[3] B.W. Boehm and W.E. Royce, "TRW IOC Ada COCOMO: Definition and Refinements," *Proceedings of the 4th COCOMO Users Group*, Pittsburgh, PA, November 1988.

[4] W.E. Royce, "Reliable, Reusable Ada Components for Constructing Large, Distributed Multi-Task Networks: Network Architecture Services (NAS)," *TRI-Ada Proceedings*, Pittsburgh, PA, October 1989.

[5] C.G. Grauling, "Requirements Analysis for Large Ada Programs: Lessons Learned on CCPDS-R," *TRI-Ada Proceedings*, Pittsburgh, PA, October 1989.

[6] M.C. Springman, "Software Design Documentation Approach for DoD-STD-2167A Ada Projects," *TRI-Ada Proceedings*, Pittsburgh, PA, October 1989.

[7] M.C. Springman, "Incremental Software Test Approach for a DoD-STD-2167A Ada Project," *TRI-Ada Proceedings*, Pittsburgh, PA, October 1989.

[8] D.H. Andres, "Software Project Management Using Effective Process Metrics: The CCPDS-R Experience," *To be presented at the AFCEA Military/Government Computing Conference*, Washington, DC, January 1990.