

UNIT – V

CASE STUDIES

APPENDIX –D

D.1 CCPDS-R:

1. Command Center Processing and Display System-Replacement (CCPDS-R)
2. TRW (Thompson Ramon Wooldridge) Space and Defense in Redondo Beach, California
3. Customer : U.S. Air Force
4. Focus : Common Subsystems
5. Mission Critical Software

D.2 Common Subsystem:

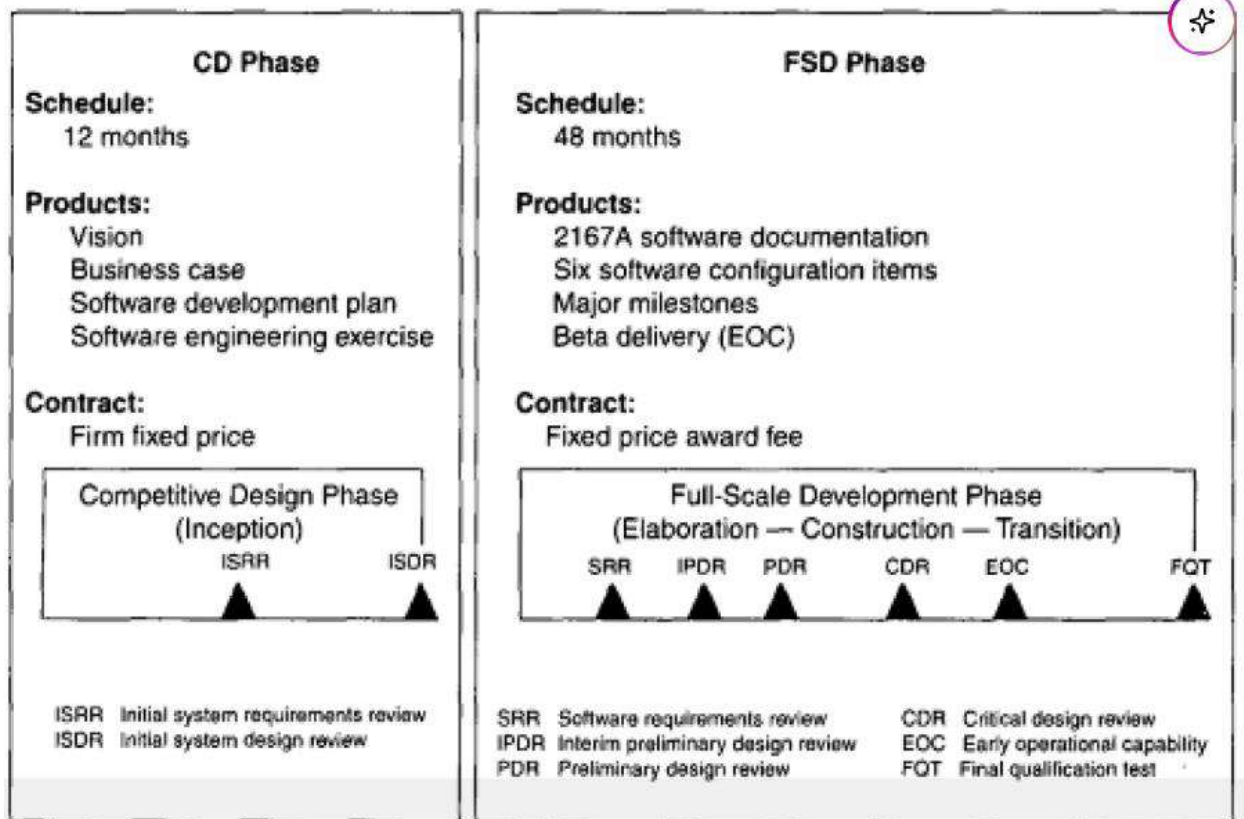
1. Primary missile warning system
2. Main installation: Cheyenne Mountain
3. Backup system : Offutt Air Force Base, Nebraska
4. 48-month software development schedule
5. 355,000 Source Lines of Code (SLOC)
6. Ada : Design and Implementation
7. 6 builds were required
8. Completed Successfully, on time within budget to customer satisfaction

CCPDS-R Acquisition Process:

1. Concept Definition (CD), 12-month schedule
2. 5 major bidders, 2 contracts awarded
3. Full-Scale Development(FSD)
4. 1 contract awarded to TRW

5. Contract award based on performance in Software Engineering Exercise
6. The winning contractors also invested their own discretionary resources to discriminate themselves with the best-value FSD phase proposal.

CCPDS-R life-cycle overview



CONCEPT DEFINITION (CD):

1. Vision
2. Analyze and Specify project requirements
3. Define and develop the top-level architecture
4. Plan FSD phase software development activities
5. Configure the process and development environment

6. Establish trust and win-win relationships among the stake holders
7. Software engineering exercise

Process Overview for FSD:

1. Standard DOD life cycle after contract award
2. Software Requirements Review (SRR)
3. Interim Preliminary Design Review (IPDR)
4. Preliminary Design Review (PDR)
5. Critical Design Review (CDR)
6. Final Qualification Test (FQT)

D.3 Project Organization:

CD Phase Team Responsibilities:

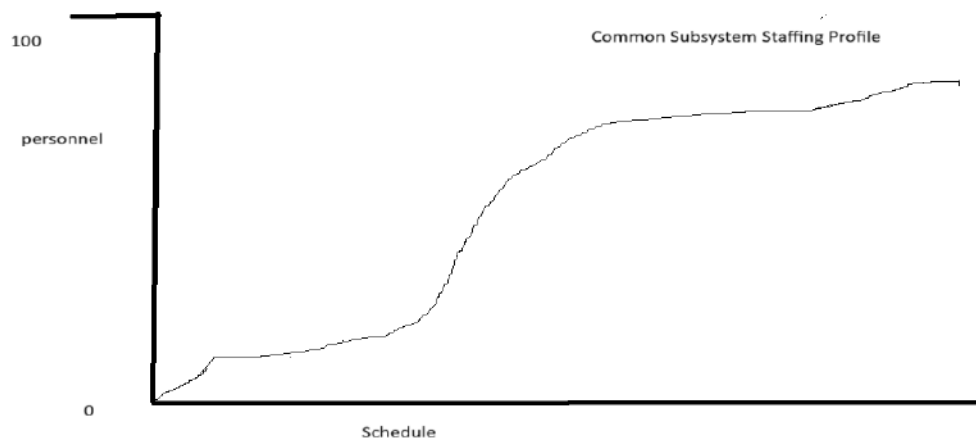
1. Analyze and specify the project requirements
2. Define and develop the top-level architecture
3. Plan the FSD phase software development activities
4. Configure the process and development environment
5. Establish trust and win-win relationships among the stake holders

D.4 Common Subsystem Product Overview:

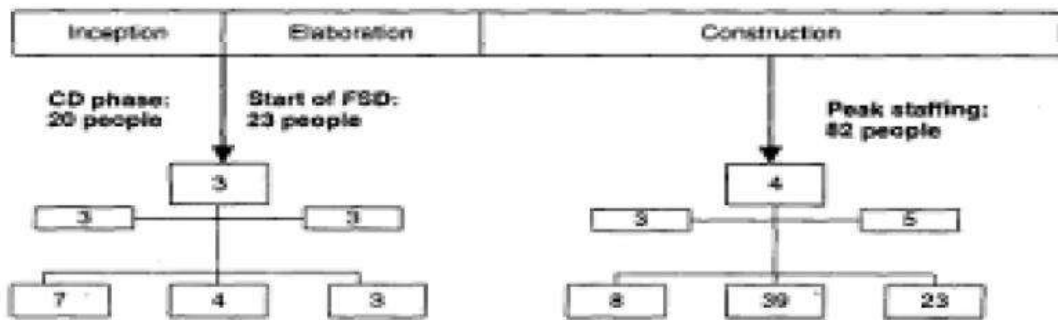
It has six Computer Software Configuration Items (CSCIs):

1. **Network Architecture Services (NAS):** It provides reusable components for network management, interprocess communications, initialization, reconfiguration, anomaly management, and instrumentation of software health, performance, and state.
2. **System Services (SSV):** This CSCI comprised the software architecture skeleton, real-time data distribution, global data types, and the computer system operator interface.
3. **Display Coordination (DCO):** This CSCI comprised user interface control, display formats, and display population.

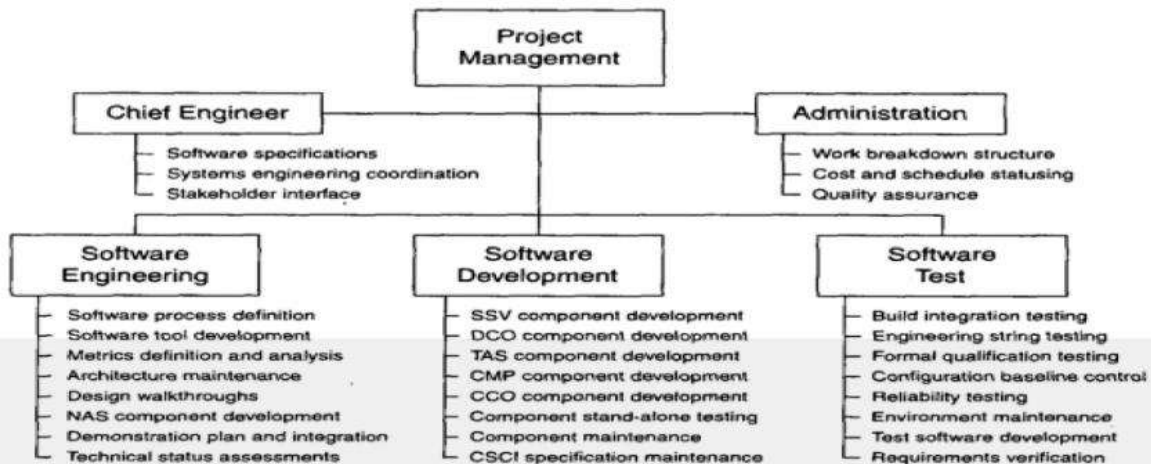
4. **Test and Simulation (TAS):** It comprised test scenario generation, test message injection, data recording, and scenario playback.
5. **Common Mission Processing (CMP):** It comprised the missile warning algorithms for radar, nuclear detonation, and satellite early warning messages.
6. **Common Communication (CCO):** It comprised external interfaces with other systems and message input, output, and protocol management.



Full-scale development phase project organization



FSD Project Organization and Responsibilities



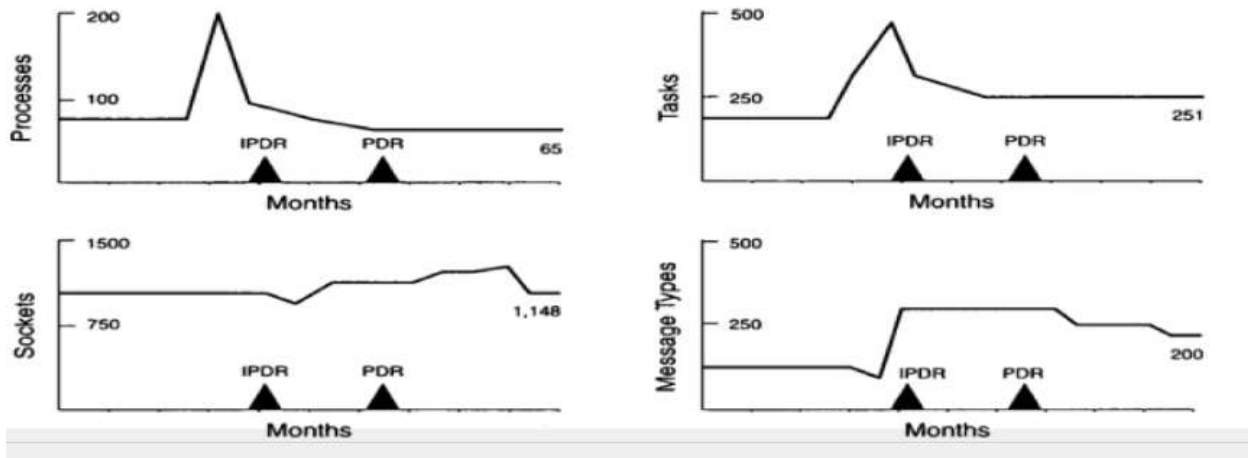
Distinguishing characteristics of each CSCI

CSCI	SIZE (SLOC)	COMPLEXITY	ASSETS (+) AND CHALLENGES (-)
NAS	20,000	Very high	+ Experienced, superior team; second-generation product - Reusable across subsystems; high performance, reliability
SSV	160,000	High	+ Some tool-produced code; stable NAS primitives - Numerous global interfaces, types, components
DCO	70,000	Moderate	+ Flexible display design; tool-produced formats - Tough performance requirement; continuous change
TAS	10,000	Low	+ Simple application; some off-line processing - Offsite team; limited environment resources
CMP	15,000	Moderate	+ Domain experience; straightforward processing - Offsite team; stringent performance requirements; numerous stakeholders involved in approving algorithm design
CCO	80,000	High	+ Skilled people; precedent domain experience - Stringent performance requirements; unstable external interface
Total	355,000	High	Large scale, high performance, high reliability

The Software Architecture Skeleton:

- All Ada main programs
- All Ada tasks and task attributes
- All sockets (asynchronous task-to-task communications), socket attributes, and connections to other sockets.
- Data types for objects passed across sockets
- NAS components for initialization, state management of processes and tasks, interprocess communications, fault handling, health and performance monitoring, instrumentation, network management, logging, and network control.

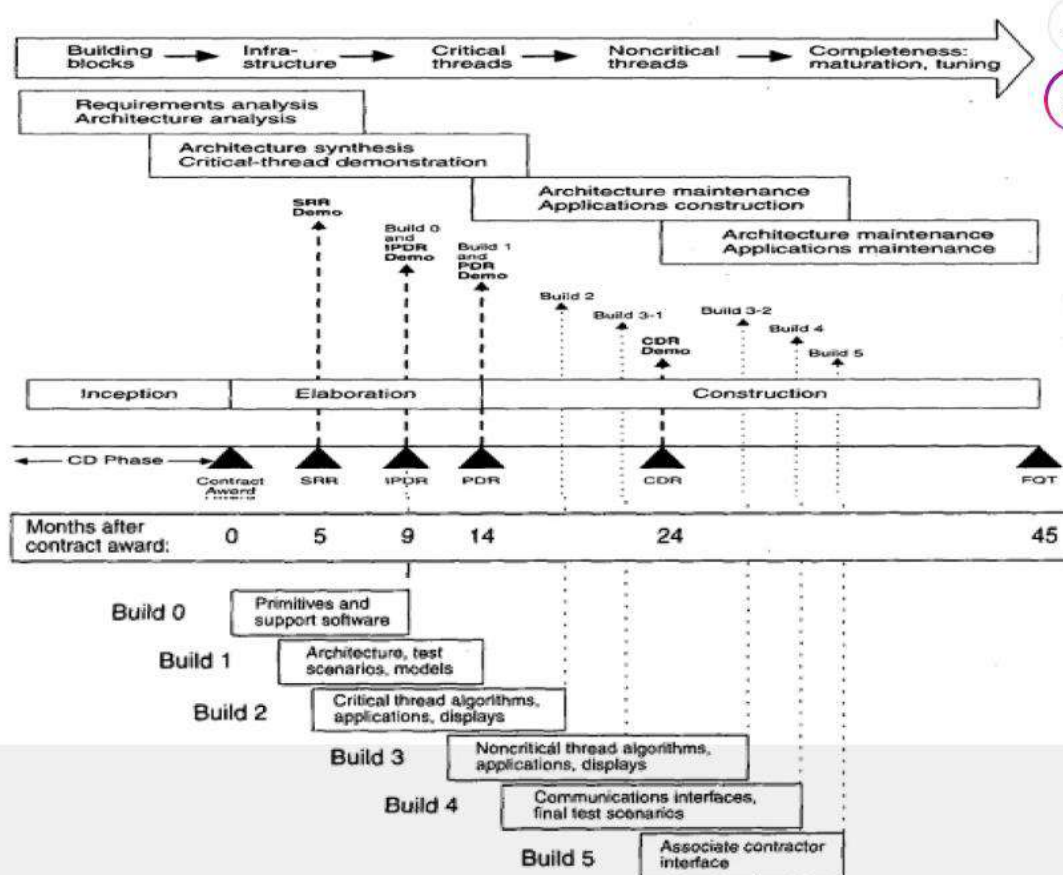
Figure shows perspectives of Software Architecture Stability (SAS)



D.5 Process Overview:

- **The SRR demonstration:** Initial feasibility of the foundation components, and basic use cases of initialization and interprocess communications.
- **The IPDR demonstration:** The feasibility of the architectural infrastructure under the riskiest use cases, including the following.
 1. A peak data load missile warning scenario of a mass raid from the Soviet Union.
 2. A peak control load scenario of a system failover and recovery from the primary thread of processing to a backup thread of processing with no loss of data.
- **The PDR demonstration:** Adequate achievement of the peak load scenarios and the other primary use cases within a full-scale architectural infrastructure, including the other critical- thread components.
- The overall CCPDS-R software process had a well-defined macro process similar to the life-cycle phases.

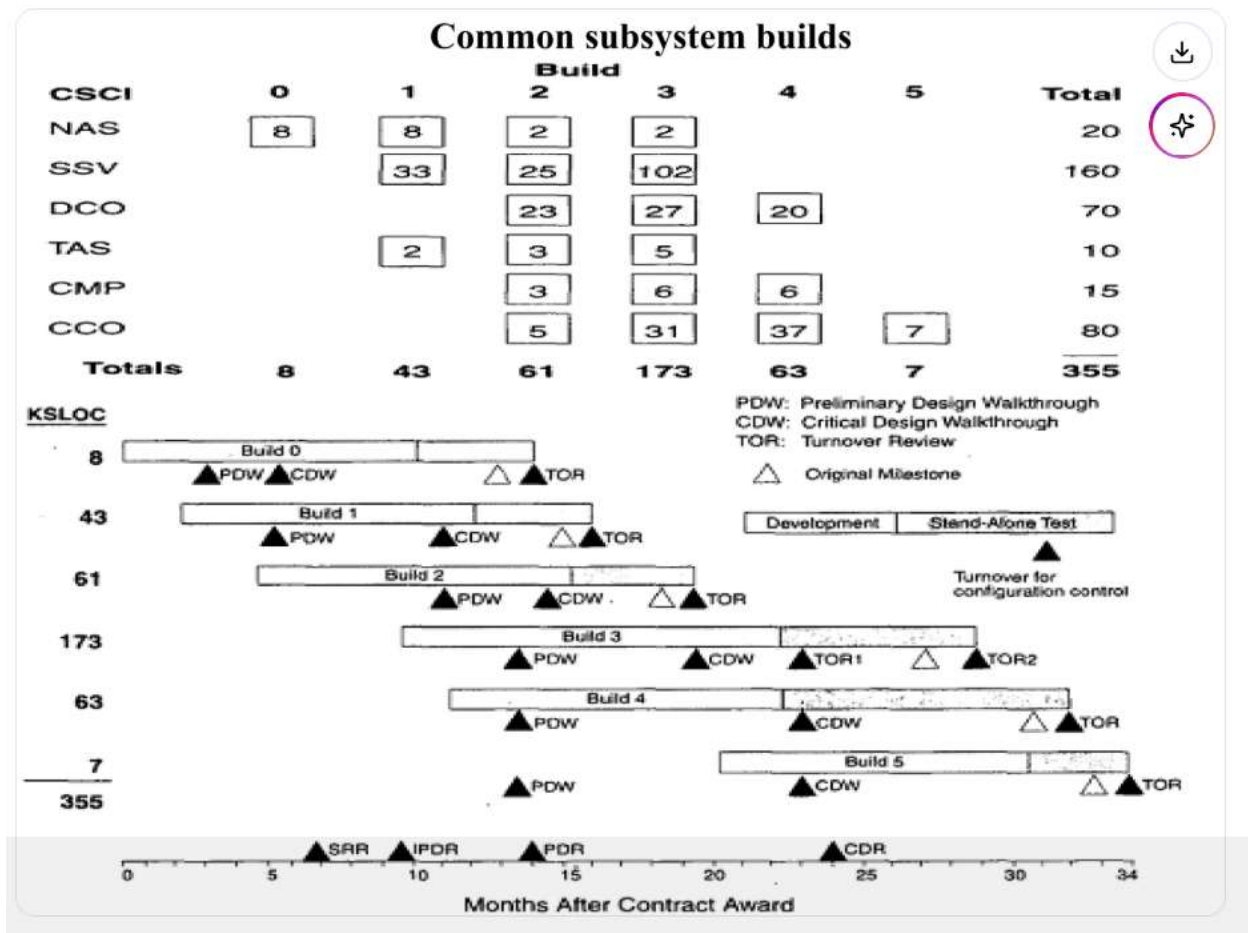
Figure shows CCPDS-R macroprocess, milestones, and schedule



D.5.1 Risk Management: Build Content

1. **Build 0:** It comprised the foundation components necessary to build a software architecture skeleton. The intertask, interprocess communications, generic task and process executives, and common error reporting components were included.
 1. It is the conclusion of the research and development project executed in parallel with the CD (inception) phase. The NAS components were the cornerstone of the architectural framework and were built to be reusable across all the three CCPDS-R subsystems. They represented very complex, high-risk components with stringent performance, reliability and reusability demands.

2. **Build 1:** This build was essentially the “architecture”. It included a complete set of instantiated tasks (300), Processes (70), interconnections (1,000), states, and state transitions for the structural solution of the CCPDS-R software architecture. To achieve a cycling architecture, this build also added all NAS components for initialization, state management (reconfiguration), and instrumentation. A trivial user interface and the capability to inject test scenarios into the architecture were added to support the initial demonstration.
3. **Build 2:** This was the first build of mission-critical components and achieved the initial capability to execute real mission scenarios. The three primary risks inherent in the mission scenarios were the timeliness of the display database distribution, the performance (resource consumption and accuracy) of the missile warning radar algorithms, and the performance of the user interface for several complex displays.



4. **Build 3:** This build contained the largest volume of code, including display format definitions, global type definitions, and representation specifications needed for validation of external interface transactions. Although the code was voluminous, much of it was produced automatically in a cookbook manner by constructing code generation tools.
1. The remaining components allocated to build 3 included the external communications interface protocol handling , the completed user- system interface for the mission operators, the user-system interface for the computer support positions , the system services for the mission reconfigurations, database

resets, off-line data reduction, and the nuclear detonation algorithms.

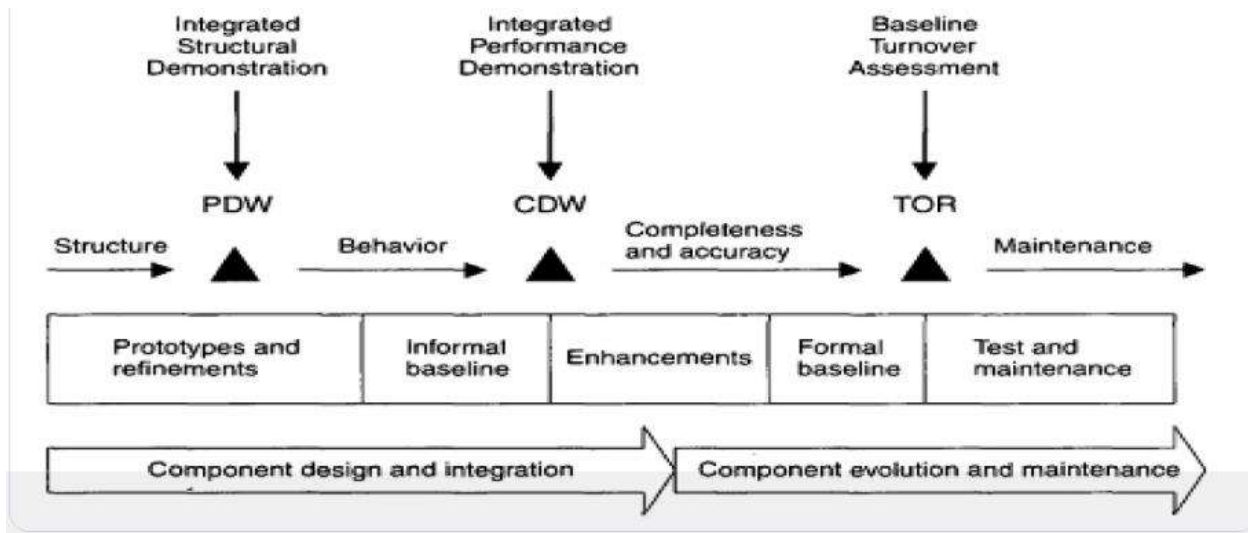
1. **Build 4:** This build provided the final installment of missile warning algorithms for satellite early warning systems, the final installment of mission management and mission status displays, and the final installment of external communications interface processing.
2. **Build 5:** In the middle of the Common Subsystem schedule, build 5 was added to coincide with a particular external interface (being built on a separate contract), the schedule for which had slipped so that it was not going to be available during its originally planned build (build 4). Consequently, the external interface was scheduled into an entirely new build.

D.5.2 The Incremental design process:

The individual milestones within a build included a

1. Preliminary design walkthrough (PDW)
2. Critical design walkthrough (CDW)
3. Turnover review (TOR)

The schedules for these milestones were flowed down from, and integrated with, the higher level project milestones (SRR, IPDR, and CDR) provides an overview of a build's life cycle and the focus of activities.



1. Well-defined sequences of walkthroughs were informal, detailed, technical peer reviews of intermediate design products.
2. They were attended by interested reviewers, including other designers, testers, and even stakeholders outside of the software organization (customer, user, project systems engineering personnel).
3. people, typically 10 to 20.
4. The explicit focus of these reviews was the important components, the architectural interfaces, and the driving issues namely, the 20% of the stuff that deserved scrutiny.
5. PDWs and CDWs usually lasted one or two days, with each of the participating CSCI managers responsible for presenting appropriate material.

Preliminary Design Walkthroughs:

1. The walkthrough focused on the structural attributes of the components within the build.
2. **Overview :** CSCI overview, interfaces, components, and metrics
3. **Components:** walkthrough of each major component, showing its source code interface, allocated system requirements specification (SRS) requirements, current metrics, operational concept for key

usage scenarios, stand-alone test plan, and erroneous conditions and responses.

4. **Demonstration:** focused on exercising the control interfaces across the components within the integrated architecture.

Critical Design Walkthroughs:

1. While the PDW focused on the declarative view of the design, the CDW focused on the completeness of the components and the behavioral perspective of operating within the allocated performance requirements.
2. **CSCI overview:** interfaces, components, and metrics; summary of changes since PDW; disposition of all PDW action items; build integration test scenarios.
3. **Components:** walkthrough of each major component, showing its source code interface, allocated SRS requirements, current metrics, operational concept for key usage scenarios, stand-alone test plan, and erroneous conditions and responses.
4. **Demonstration:** focused on exercising the critical performance threads.

Code walkthroughs:

- Better dissemination of self-documenting source code style
- Identification of coding issues not easily caught by compilers and source code analysis tools
 - **Object naming, coding style, and commenting style:** Does it promote readability?
 - **Unnecessarily complex objects or methods:** Are there simpler approaches?
 - **Potential performance issues:** Are there potentially inefficient implementations?
- Reduction of the amount of source code needed for review in the larger design walkthroughs.

- Exposure of inexperienced personnel to the products of experts and vice versa.

Turnover Reviews:

1. Turnover reviews were not really reviews; they were typically a one-month activity during which components were completed with stand-alone testing and turned over for configuration control, build integration testing, and engineering string testing.
2. The checkpoints used on CCPDS-R for the incremental design process are good examples of the major milestones.
3. The mixture of walkthroughs and inspections was focused on the 20% of components that had a potential high return on human resource investment.
4. In general, the real value of design walkthroughs and inspections was communications among project sub teams and methodical coordination of processes.
5. Very few serious quality flaws were uncovered in these meetings (as opposed to the demonstration activities), but the technical interchange was well mechanized by these checkpoints.

D.5.3 Component Evolution:

1. TBD (to be defined) package included predefined TBD types, TBD constants, TBD values, and a TBD procedure for depicting source lines of codes associated with comments that together would act as placeholders for as-yet undefined code segments.
2. Procedure declaration:

TBD_statements (Number_of_statements: In Integer);

 1. The basic component evolution would look like this:
 2. At creation, only the interface (the specification part) would be defined with Ada source lines and corresponding comments. The Estimated SLOC count for the component would typically be specified by a single TBD_Statements line.

1. At PDW, the substructure of the component would be fleshed out along with most component declarations and estimates of the subordinate program units using multiple calls to TBD_Statements line.
2. By CDW, most of the program unit interfaces and declarations would be fully fleshed out in Ada, with some detailed processing still using TBD_Statements as placeholders. In general, CDW-level components would be about 70% Ada and 30% ADL. A guideline also stated that by CDW, there would be no calls to TBD_Statements with values greater than 2.5.
3. By turnover, the string TBD would not appear anywhere in the source files. This would correspond to a complete implementation.

VIEW	PROGRAM UNIT	TYPE	Ada	ADL	TOTAL	%
Creation view	Total package	Package	6	122	128	5
	Inm_Erm_Procedures		2	122	124	2
PDWview	Total package		47	101	148	32
	Inm_Erm_Procedures	Package	24	19	43	56
	All_Node_Connections	Procedure	3	19	22	14
	Create_Inm_Ern_Circuits	Procedure	4	8	12	33
	On_Node_Connections	Procedure	3	7	10	30
	Perform_Reconfiguration	Procedure	6	2	8	75
	Perform_Shutdown	Procedure	4	3	7	57
	Process_Error_Messages	Procedure	3	43	46	7
CDW View	Total Package		87	48	135	65
	Inm_Erm_Procedures	Package	30	11	41	73
	All_Node_Connections	Procedure	16	0	16	100
	Create_Inm_Erm_Circuits	Procedure	8	4	12	67
	On_Node_Connections	Procedure	9	0	9	100
	Perform_Reconfiguration	Procedure	6	2	8	75
	Perform_Shutdown	Procedure	6	1	7	86
	Process_Error_Messages	Procedure	12	30	42	2
Turnover view	Total Package		137	0	137	100
	Inm_Erm_Procedures	Package	42	0	42	100
	All_Node_Connections	Procedure	16	0	16	100
	Create_Inm_Erm_Circuits	Procedure	12	0	12	100
	On_Node_Connections	Procedure	9	0	9	100
	Perform_Reconfiguration	Procedure	8	0	8	100
	Perform_Shutdown	Procedure	7	0	7	100
	Process_Error_Messages	Procedure	43	0	43	100

A typical component evolution from creation through turnover

ELEMENTS	TOTAL	DESIGNED	CODED
Top-level components	40	39	33
Lower-level components	13	13	10
Total Program units	494	484	459
	Source lines:18,494	ADL: 1,858 10% TBD	Ada : 16,636 90% complete

NAS CSCI metrics summary at month 10

D.5.4 The Incremental Test Process:

Test sequence was devised with five different test activities: stand-alone test, build integration test, reliability test, engineering string test, and final qualification test.

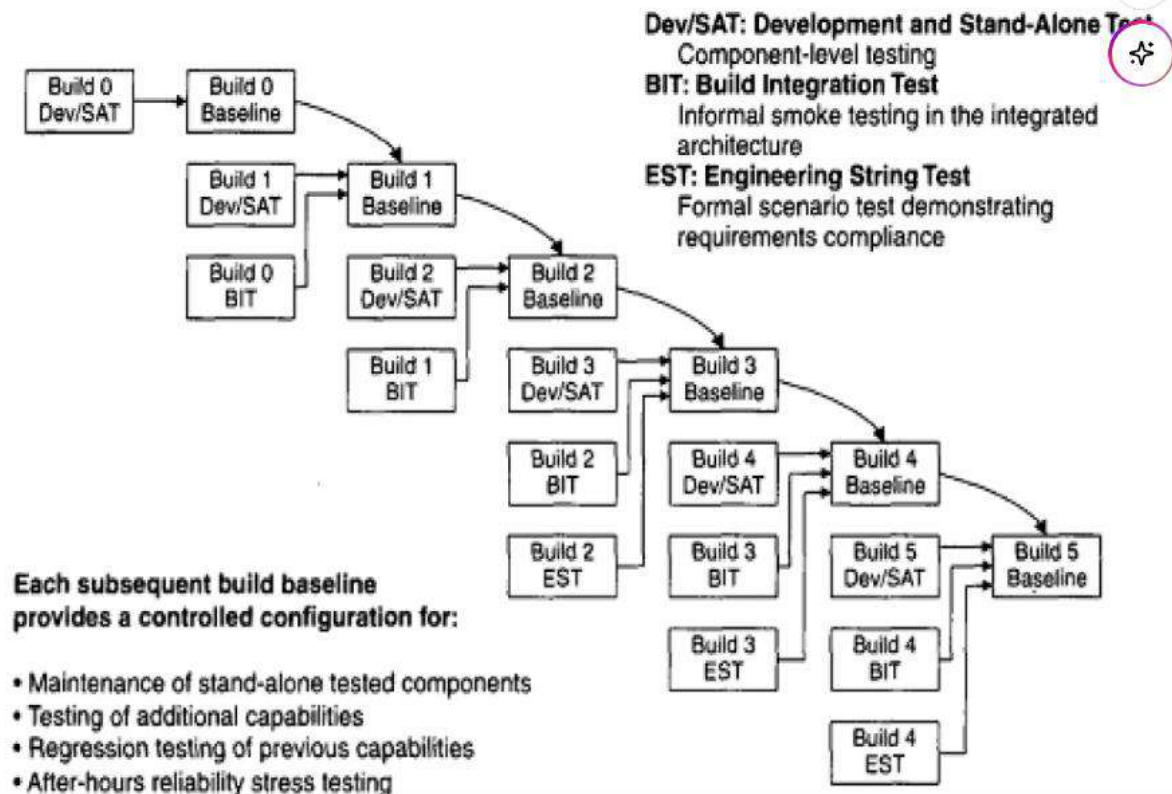
1. **Stand-Alone Test (SAT):** The development teams were responsible for stand-alone testing of components before delivery into a formal, configuration-controlled test baseline used for all other test activities. SAT typically tested a single component (which may comprise several lower level components) in a stand-alone environment. This level of testing corresponds to completeness and boundary condition tests to the extent possible in a stand-alone context.
2. **Build Integration Test (BIT):** This was mostly a smoke test to ensure that previously demonstrated capabilities still operated as expected. A BIT sequence is the primary quality assessment vehicle for closing out a turn-over review. A given build turnover may take days or weeks, depending on its size or the percentage of new componentry.

The purpose of BIT is not to verify requirements but to establish a stable, reliable baseline. It is very informal, dynamic, and focused on exposing errors and inconsistencies. BITs validate the following:

1. Previously demonstrated threads can be repeated successfully
2. Previously defined deficiencies have been resolved
3. Interfaces across components are completely tested
4. The baseline is stable enough for efficient requirements verification testing
5. **Reliability Test:** One of the outputs of BIT process and a turnover review was stable test baseline that was subjected to extensive after-hours stress testing for long periods of time under randomized but realistic test scenarios.

This was designed to help uncover potentially insipid, transient errors of major design consequence.
Reliability testing logged as much test time as possible while resources were otherwise mostly idle (on nights and weekends).
6. **Engineering String Test(EST):** These tests focused on verifying specific subsets of requirements across multiple CSCIs through demonstration and test of use case realizations (called capability threads).
7. **Final Qualification Test (FQT):** These tests were equivalent to ESTs except that they represented the set of requirements that could not be verified unless the whole system was present. For example, a 50% reserve capacity requirement could not be verified until FQT, when the whole system was operational.

Incremental baseline evolution and test activity flow



D.5.5 DOD-STD-2167A ARTIFACTS

1. Use of the evolving Ada source files as the single homogeneous life-cycle design format and evolution of these files in a self-documenting manner. This technique exploited Ada's readability features and avoided the extra effort involved in preparing separate, detailed design descriptions that inevitably diverge from the implementation.
2. Organization of the test sequences and deliverable documents around the build content driven by subsets of use cases (referred to as engineering strings and scenarios) rather than by CSCI. This string-based testing spanned components in multiple CSCIs. It was organized by build and mechanized via a software test plan, software test procedure, and software test report documentation sequence. These document sequences were provided for each BIT

(one for each build), each EST (for builds 2,3, and 4), and FQT (one final all-encompassing test sequence). Each test sequence involved components from several (incomplete) CSCIs because integration was proceeding continuously.

3. Building of separate unit test documentation as self-documented, repeatable software. This was treated like other operational source code so that it was maintained homogeneously and up-to-date for automated regression testing. The same concept was used for the BIT and EST scenario testing: Rather than develop test procedure documents, the CCPDS-R process generated self-documenting test scenarios that were software programs in their own right. Because they were subjected to change management just like other software, they were always maintained up-to-date for automated regression testing.

CCPDS-R software artifacts

NO.	CONTRACT-DELIVERABLE DOCUMENT	ARTIFACT COUNTERPART (CHAPTER 6)
1	System specification	Vision statement
6	Software requirements specification (SRS)	End-item release specifications 1 for each CSCI
1	System design document (SDD)	Architecture description
6	Software top-level design document (STLDD)	UML design models 1 for each CSCI
12	Software development file (SDF)	Implementation set artifacts 1 per component
6	Software product specification (SPS)	Final implementation set artifacts Deployment set artifacts 1 for each CSCI
4	Demonstration plan (not required by 2167A)	Major milestone release specifications 1 for each major demonstration
4	Demonstration report (not required by 2167A)	Major milestone release description 1 for each major demonstration
9	Test data file (TDF)	Release descriptions 1 for each build's BIT, EST, and FQT test sequences
4	Software test plan (STP)	Release specifications Design set artifacts, test models
4	Software test procedure (STPR)	Implementation set artifacts
4	Software test report (STR)	Release descriptions
1	Software development plan (SDP)	Software development plan
1	Software standards and procedures manual (SSPM)	Software development plan
18	Project management review (PMR)	Status assessments
3	Software user manual (SUM)	User manual

Software development file evolution

SECTION	PDW STATUS	CDW STATUS	TOR STATUS
Requirements	Summarized	Allocated	Traced
Component overview	Complete	Complete	Complete
Top-level program unit	Ada	Ada	Ada baselines
Subordinate units	Ada/ADL	Ada/ADL	Ada baselines
SAT plan	Draft	Complete	Complete
SAT test code	Some demonstration	Draft	Ada baselines
SAT test results	Some demonstration	Some demonstration	Ada baselines
SCO log	None	None	Initial
Metrics	Initial metrics	Updated metrics	Updated metrics
Code auditor results	None	Initial	Complete
Notes/waivers	None	None	As needed

D.6 DEMONSTRATION-BASED ASSESSMENT:

4. Design review on an executable demonstration provides a more understandable and concrete review vehicle for diverse set of stakeholders.
5. Demonstrations provide acute insight of the architecture and its subordinate components, the run-time performance risks, and the understanding of the system's operational concepts and key use cases.
6. The sequence of demonstration activities included the development of a plan, definition of a set of evolution criteria, integration of components into an executable capability, and generation of test-drivers, scenarios, and throw-away components.

7. The demonstration of plans are captured the purpose of actual evolution criteria for assessing the results, the scenarios of execution and overall hardware and software configuration that would be demonstrated.

Key lessons were in the CCPDS-R demonstration activities:

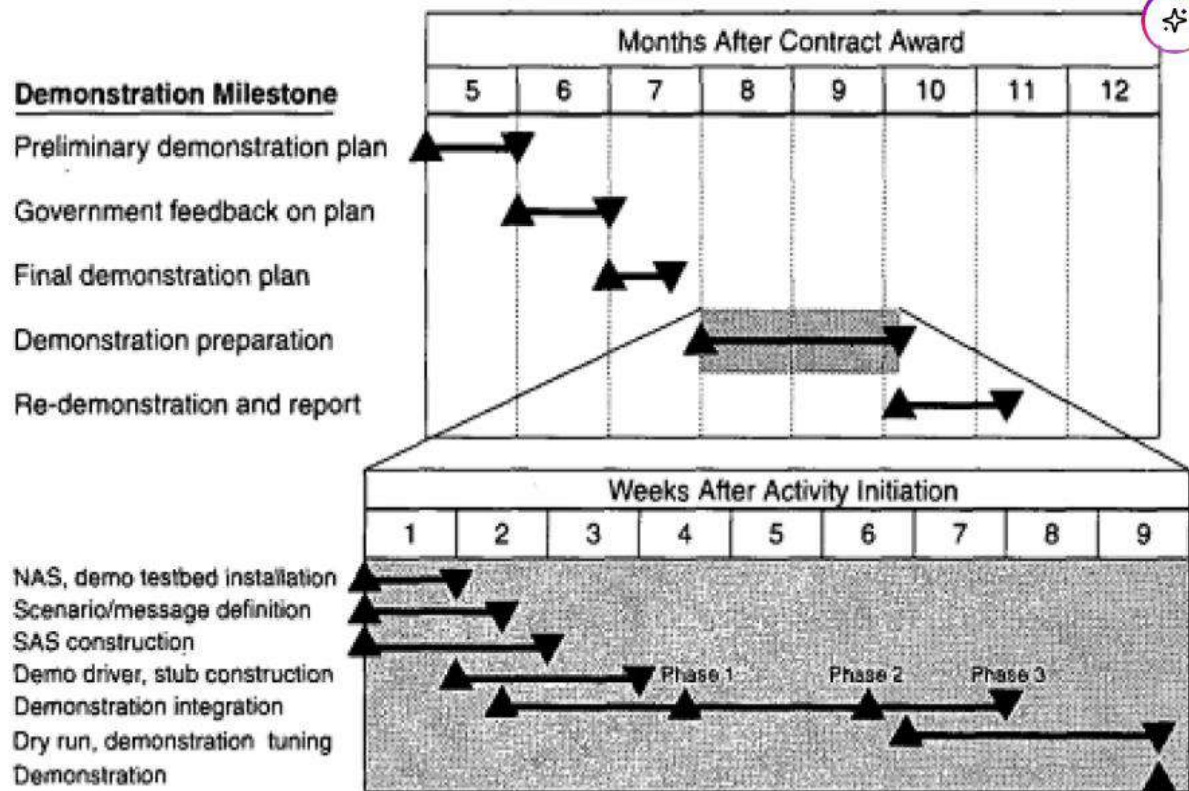
1. Early constructions of test scenarios have a high ROI. The early investment in building some of the critical test scenarios served two invaluable purposes.
2. Demonstration planning and execution expose the important risks. Negotiating the content of each demonstration and associated evolution criteria served to focus all teams in organization.
3. Demonstrate defined capabilities at NORAD.
4. Demonstration infrastructure, instrumentation, and scaffolding have a high ROI.
5. Capabilities well understood by the customer and TRW.
6. Demonstration activities expose the crucial trade-offs, the integration of the demonstration provided timely feedback on the important design attributes and the level of design maturity.
7. Early performance issues drive early architecture improvements.
8. Results were apt to change requirements, plans and design.

The IPDR Demonstration:

1. The interim PDR major milestone demonstration of the Common Subsystem had three critical objectives
 1. Tangible assessment of the software architecture design integrity through construction of a prototype SAS.
 2. Tangible assessment of the critical requirements understanding through construction of the worst-case missile warning scenario.
 3. Exposure of the architectural risks associated with the peak missile warning scenario (the worst-case data processing performance corresponding to a mass missile raid from the Soviet Union) and the fault detection and recovery scenario

(the worst-case control processing associated with a failure in the primary processing thread and a real-time switchover to a hot backup processing thread).

2. Figure summarizes the schedule for the IPDR demonstration; it includes details of the intense integration period in the two months before the demonstration.
3. The first three months of planning, which encompassed a draft plan, government review and comment, and final plan production, could have been achieved in one week with a collected team of all interested stakeholders.
4. The review sequence that occurred was a requirement of the contract. Because this was first time that TRW or the customer had used a demonstration-based approach, both were unsure of the best process and agreed on an overly conservative approach.
5. This demonstration was the first attempt at constructing a full-scale SAS.
6. The subsequent demonstrations tended to have shortened but equally intense, integration activities lasting 4 to 5 weeks.



IPDR Demonstration Scope:

1. The contractor shall demonstrate the following capabilities as the NORAD.
2. Demo 1: System services, system initialization, system failover and recovery, System reconfiguration, test message injection, and data logging.
3. These capabilities were fairly well understood by the customer and TRW
 1. System services were the NAS software components of general utility to be reused across all three subsystems. The components are interprocess communications services, generic application control (generic task and process executives), NAS utilities (list routines, name services), and common error reporting and monitoring services.

2. Data logging (SSV CSCI) was a capability needed to instrument some of the results of the demonstration and was a performance concern.
3. Test message injections (TAS CSCI) components permitted messages to be injected into any object in the system.
4. System initialization illustrates the existence of a consistent software architecture skeleton and error-free operation of a substantial set of the system services.
5. The second scenario was to inject the peak message load into architecture and cause all the internal message traffic to cascade through the system in realistic way.
6. System failover and recovery was one of the riskiest scenarios, because it required a very sophisticated set of management and state transition control interfaces.

IPDR Demonstration Evaluation Criteria

- Derived from the requirements, the assessments, and the evolving design trade-offs:

All phases:

- No critical errors shall occur.

Phase 1:

- The system shall initialize itself in less than 10 minutes.
- The system shall be initialized from a single terminal.
- After initialization is complete, the number of processes, tasks, and sockets shall match exactly the expected numbers in the then-current SAS baseline.

Phase 2:

- Averaged over the worst-case minute of the 20-minute peak scenario, the total processor utilization for each node shall be less than 30%.
- There shall be no error reports of duplicate or lost messages.
- All displayed data shall be received within 1 second from its injection time.
- The message injection process shall maintain an injection rate matching the intended scenario rate.
- The data logs shall show no unexpected state transitions or error reports and shall log all injected messages.

IPDR Demonstration Evaluation Criteria



Phase 3:



- The operator shall be capable of injecting a fault into any object.
- An error report shall be received within 2 seconds of the injection of a fault.
- The switchover from the primary to backup thread shall be completed within 2 seconds of the fault injection with no loss of data.
- The shutdown of the failed primary thread and reinitialization as a new backup thread shall be completed in less than 5 minutes from failure.
- The data logs shall match the expected state transitions with no fatal errors reported other than the injected fault.

IPDR Demonstration Results



- From the 37 evaluation criteria, 31 were considered satisfactory.
- Six criteria were not met, including three of the essential criteria just discussed. These were considered very serious issues that required immediate redesign and re-demonstration.
- Of most concern was excessive processor utilization during the peak load scenario. While the threshold was 30%, actual utilization was 54%.
- This corresponded to the essential overhead of the architectural infrastructure, operating system, and networking software. Because this was always a perceived risk of the CCPDS-R reusable middleware design, it received extensive attention.
- Five distinct action items for performance analysis were created, as well as an action item to demonstrate the performance improvement at the next project management review after the five action items were resolved.

Five action items are

1. Update the scenario.
2. Tune the interprocess communications.
3. Enhance network transactions.
4. Improve performance of the IPe component.
5. Improve reliability in the IPe component.

IPDR Demonstration Results



Lessons were learned from results are:



1. Very effective design review was occurring throughout the period. The demonstration was the result of the engineering team's review, presented to the stakeholders as tangible evidence of progress. Although we ended up with only five open issues, 50 or more design issues had been opened, resolved, and closed during the 8-week integration activity. This early resolution of defects-in the requirements specification, the process, the tools,
2. Through day-to-day participation in this activity, gained detailed insight into where the design was weak, where it was robust, and why.
3. The demonstration served as a strong team-building exercise in which there was a very tangible goal and the engineers were working in the forum they preferred: getting stuff to work.
4. The detailed technical understanding and objective discussions of design trade-offs proved invaluable to developing a trustworthy relationship with all stakeholders, including the customer, the user; and TRW management. We were armed with facts and figures, not subjective speculation.

Government Response to the **IPDR Demonstration**

After careful evaluation of the Government's Preliminary Demo 1 Plan comments, following observations summarize this submittal of the Demo 1 Plan and the modifications that have been made from the previous version:

1. This submittal has eliminated all requirements references to avoid making any allusion to an intent of satisfying, proving, or demonstrating any requirements. These requirements verification activities are performed by the test organization in a very rigorous and traceable fashion. The demonstration activity is intended to be an engineering-intensive activity, stream-lined through minimal documentation, to provide early insight into the design feasibility and progress. TRW intends to maximize the usefulness of the demonstration as an engineering activity and to avoid turning it into a less useful documentation-intensive effort.
2. Several government comments requested further details on requirements, designs, etc. This information is not necessary in the Demo Plan. It is redundant with other documents (SRS, SDD, design walkthrough packages) or it is provided in the informal test procedures delivered 2 weeks prior to the demonstration. Providing more information in a single document (and in every document) may make the reviewer's job easier but it would also be excessive, more time-consuming, and counterproductive to produce, thereby reducing the technical content of the engineering product being reviewed.

Government Response to the **IPDR Demonstration**



3. In light of the government's concern over the relationship of the demonstration to requirements, the evaluation criteria provided in this plan should be carefully scrutinized. We feel that the evaluation criteria are explicit, observable, and insightful with respect to determining design feasibility, especially at such an early point in the life cycle. Although we are open to constructive modification of these evaluation criteria, we feel that modifying them to relate more closely to the System Specification or SRS requirements would be inappropriate. The requirements perspective and our demonstration perspective are different and difficult to relate.
4. The source code for the components being demonstrated has not been delivered with the plan as required in the statement of work. The total volume for the demonstrated components is roughly 1 to 2 feet thick, and it is still changing at a rapid rate. Instead of delivering all the source code, interested reviewers may request specific components for review. All source code will be browseable at the contractor facility during the demonstration.

CORE METRICS



TRW formulated a metrics program with four objectives:

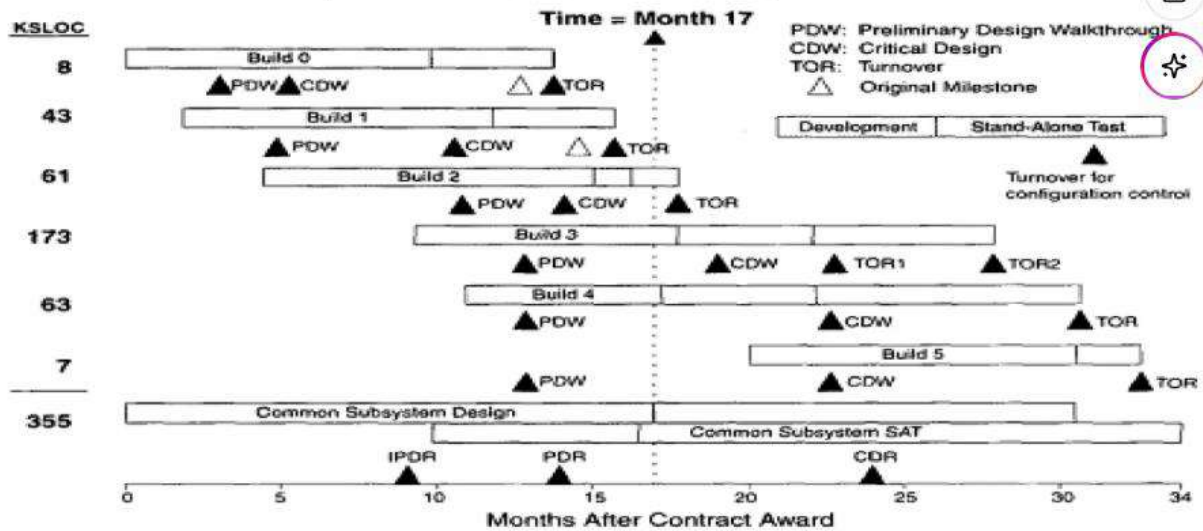


- Provide data for assessing current project trends and identifying the need for management attention
- Provide data for planning future builds and subsystems
- Provide data for assessing the relative complexity of meeting the software end-item quality requirements
- Provide data for identifying where process improvements are needed and substantiating the need

1. DEVELOPMENT PROGRESS

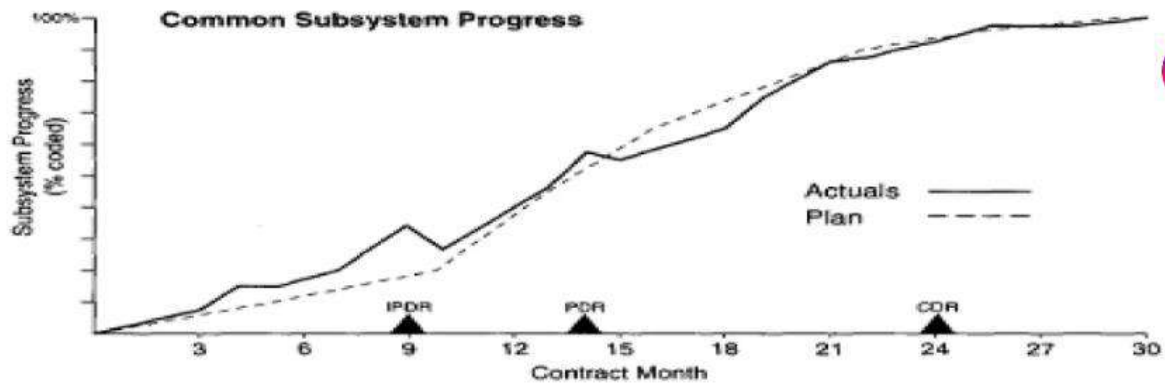
- **The Ada/ADL metrics.** These data provided good insight into the direct indicators of technical progress. By themselves, these metrics were fairly accurate at depicting the true progress in design and implementation. They were generally weak at depicting the completed contract deliverables and financial status.
- **Earned value metrics.** These data provided good insight into the financial status and contract deliverables. They were generally weak indicators of true technical progress.

Development progress summary

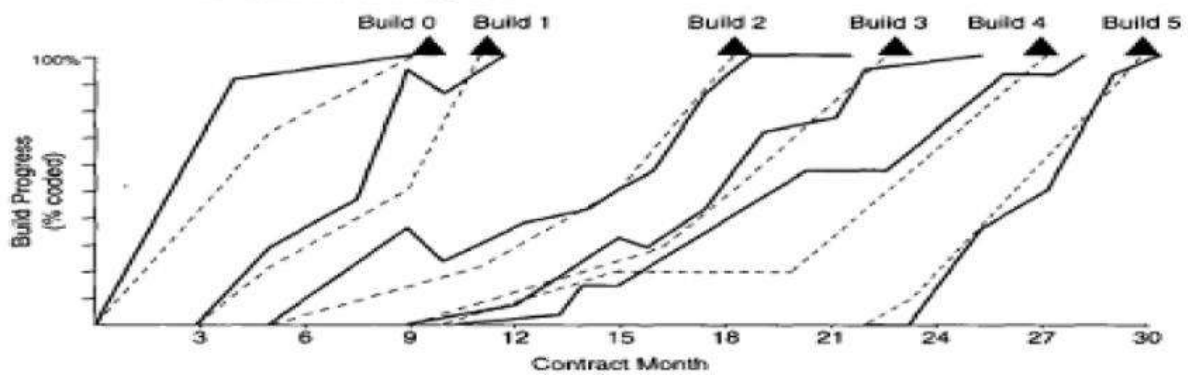


- Figure displays month 17 status: Build 2 SAT testing is one month behind schedule, build 3 design work is one month ahead of schedule, the Common Subsystem design effort is on schedule, and Common Subsystem SAT testing is one month behind schedule.
- The planned evolution was based roughly on weight-averaging the SLOC counts for each build with the guidelines described in Section 0.5.3: 30% done by PDW and 70% done by CDW. Overall, the Common Subsystem performed very close to its plan, with one exception.

Common Subsystem development progress



Individual Build Progress



CORE METRICS



2. TEST PROGRESS



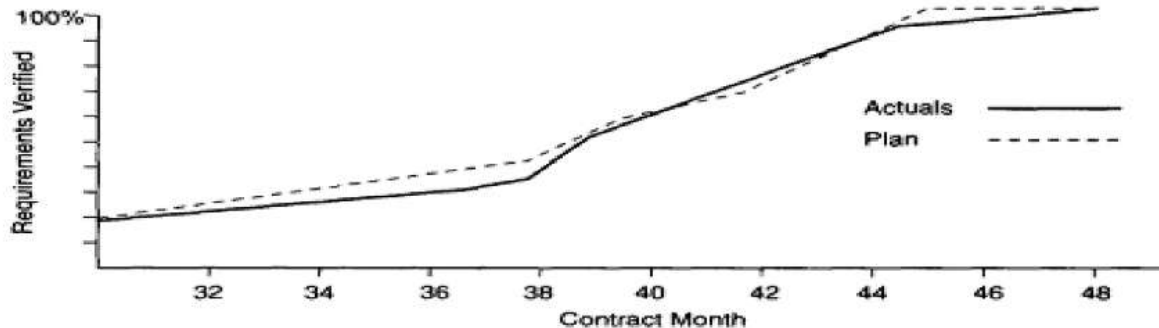
- Build integration tests and requirements verification testing (some SATs, ESTs, and FQT). Build integration testing proved to be less effective than expected for uncovering problems.
- Table and Figure provide perspectives on the progress metrics used to plan and track the CCPOS-R test program. The figure plots the progress against the plan for requirements verification tests.
- SATs, ESTs, and FQTs were sources of test cases used by the software organization.

PROBLEM SOURCE	MINOR (<1 HOUR)	MODERATE (<1 DAY)	MAJOR (>1 DAY)	TOTAL
Requirement interpretation	5			5
Inadequate stand-alone test	3	4	2	9
Interface problem	9	2	1	12
Inadequate performance	1			1
Desired enhancement (not a problem)	3			3
Inconsistent configuration	3	2		5
Totals	24	8	3	35

- SATs were the responsibility of the development teams but had to be executed in the formal configuration management environment and witnessed (peer-reviewed) by the development personnel.
- ESTs consisted of functionally related groups of scenarios that demonstrated requirements spanning multiple components. FQTs were tests for requirements compliance that could not be demonstrated until a complete system existed.
- Quantitative performance requirements (QPRs) spanned all CSCIs. The SAT procedures were rarely available 30 to 60 days before turnover, as required by the contract for any requirements verification test. The formal SAT process was one of the main reasons that the turnover reviews were consistently completed later than planned.

Test progress at month 41, for example:

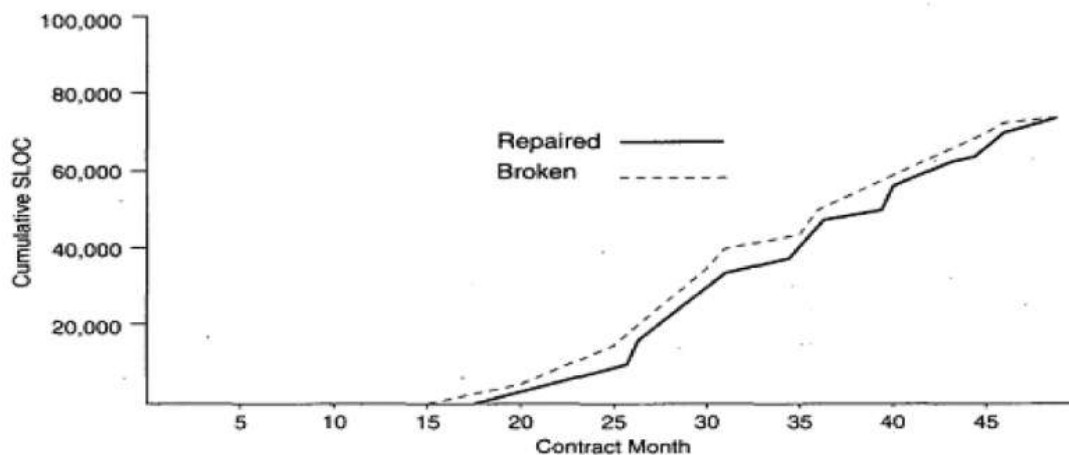
523	of	589	SAT verifications
1,003	of	1,003	EST verifications
0	of	543	EST5/FQT verifications
1,526	of	2,135	requirements verified (72% complete)



CORE METRICS

3. STABILITY

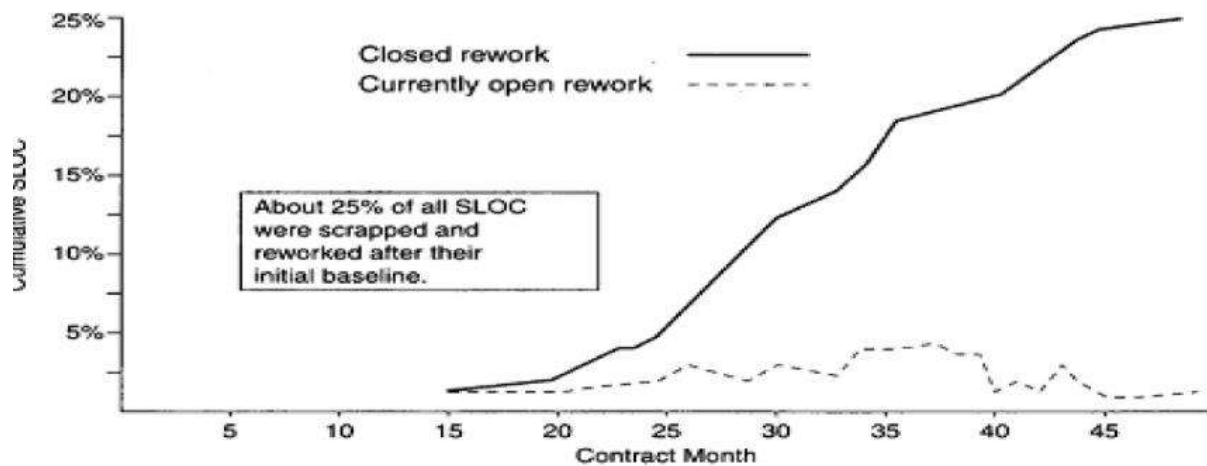
- Figure illustrates the overall rate of configuration baseline changes. It shows the cumulative number of SLOC that were **broken** and the number of **SLOC repaired** (checked back into the baseline with fixes, enhancements, or other changes).
- Breakage rates that diverged from repair rates resulted in **management attention**, **reprioritization of resources**, and **corrective actions** taken to ensure that the test organization (driving the breakage) and development organization (driving the repair) remained in relative equilibrium. Overall, the situation shown depicts a very healthy project.



CORE METRICS

4. MODULARITY

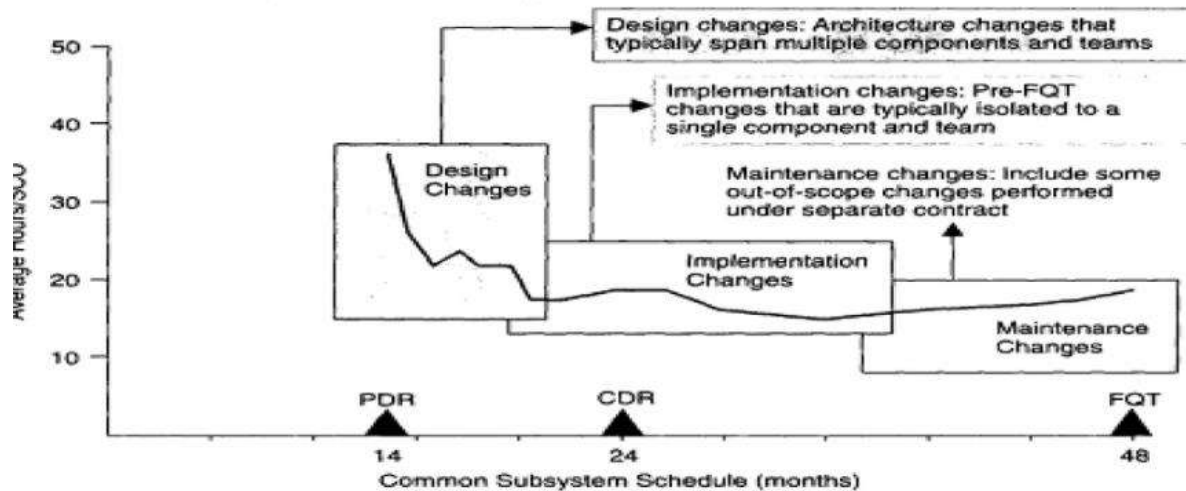
- Figure identifies the **total breakage as a ratio** of the entire software subsystem. This number identifies the total scrap generated by the Common Subsystem software development process as about 25% of the whole product. Industry averages for software scrap run in the 40% to 60% range. The initial configuration management baseline was established around the time of PDR, at month 14. There were 1,600 discrete changes processed against configuration baselines.



CORE METRICS

5. ADAPTABILITY

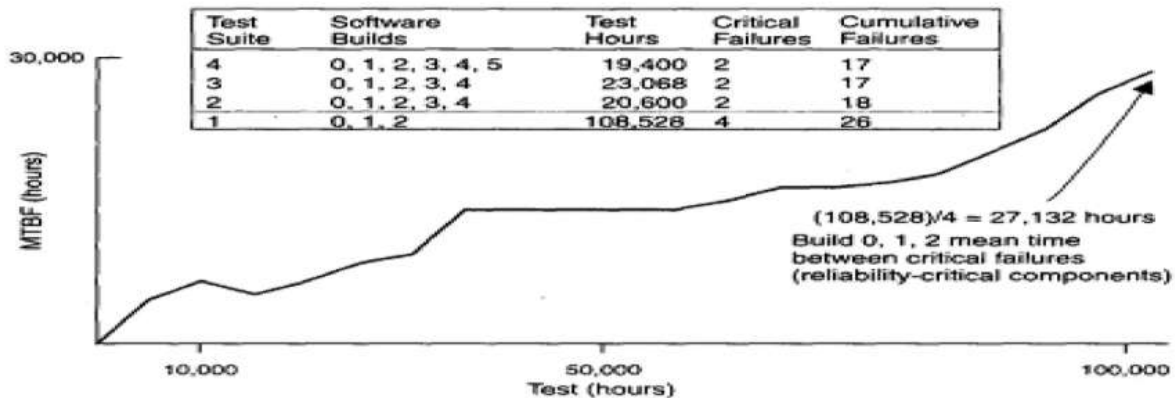
- Over the entire Common Subsystem, about **5% of the total effort** was expended on **rework** activities against software baselines. The average cost of change was about 24 hours per SCO. These values provide some insight into the ease with which the software baselines could be changed.
- The level of **adaptability achieved** by CCPDS-R was roughly **four times better** than the typical project, in which **rework costs** over the development life cycle usually **exceed 20% of the total cost**. Figure plots the average cost of change across the Common Subsystem schedule. The 1,600+ SCOs processed against the evolving configuration baseline by FQT resulted in a fairly stable cost of change.



6. MATURITY

CORE METRICS

- CCPDS-R had a specific reliability requirement, for which the software had a specific allocation. The independent test team constructed an after-hours, **automated test suite** that exercised **evolving software** baselines with randomized message scenarios.
- This strategy resulted in extensive test time being logged under realistic conditions from which a credible software MTBF could be substantiated. The **reliability-critical components**, forced by the iteration plan into the earliest baselines, were subjected the most reliability stress testing. Figure illustrates the results with **modern distributed architectures uncovering significant issues** of races, deadlocks, resource overruns, memory leakage, and other Heisen-bugs. **Executing** randomized and accelerated scenarios for **long periods of time** (running all night or over a long weekend) enables early **insight into overall system resource** integrity.



CORE METRICS

7. COST EFFORT EXPENDITURES BY ACTIVITY

- Table provides the overall cost breakdown for the CCPDS-R Common Subsystem. 7 data were extracted from the final WBS cost collection runs and were structured.

WBS ELEMENT	COST (%)	ACTIVITIES AND ARTIFACTS
Management and administration	9	Deliverable plans, administrative support, financial administration, customer interface, contracts, overall control and leadership
Process/product specification	7	Technical requirements, demonstration plans and evaluation criteria, iteration plans, software process, metrics analysis
Software engineering	11	Architecture engineering, design walkthrough coordination, NAS CSCI development, metrics definition and assessment, demonstration planning and integration
Development	38	Development, testing, documentation, and maintenance of application components
Testing, assessment, and deployment	24	Release management; formal test preparation, conduct, and reporting; test scenario development; change management; deployment
Infrastructure	11	System administration, hardware and software resources, toolsmithing, tool integration
Total software activities	100	Cost expenditures, including hardware and software tools (in the infrastructure element), travel, and other direct costs

- The overall test team effort is relatively low compared with the effort typically expended on projects that used the conventional process.
- CCPDS-R used an efficient environment that represented 11 % of the total cost of the effort.
- Overall maintenance (total rework effort expended) was only 5% of the total cost. Although this is not shown explicitly in the tables, it was tracked in the individual CSCI WBS elements.

8. OTHER METRICS



Few other metrics of CCPDS-R project performance:

1. software size evolution,
2. subsystem process improvements,
3. SCO resolution profile, and
4. CSCI productivities and quality factors.

1. Software size evolution

- Large amount of code growth from the original contract bid (150,000 SLOC) to the delivered product (355,000 SLOC), with no substantial increase in the software development budget.

There were two reasons for this level of code growth:

- The method for **counting source lines** was changed around month 8 to provide a better balance in estimating the engineering effort and to be consistent with the counting method embraced by Ada COCOMO.
- **Several automatic code generation tools** were developed that output verbose source code with fewer human-generated input lines. These tools were used for the straightforward generation of display formats, message validation processing, and socket/circuit bookkeeping functions. They represented about 14,000 SLOC of tools, which required another 20,000 lines of input data files. The output of these tools was about 200,000 SLOC of operational software. **In gross terms, these code generation tools resulted** in about a fivefold return on investment.

Four coding standards allowed the SLOC counting to be consistent:

1. **Each parameter of a subprogram declaration** is listed on a separate line. The effort associated with design of a subprogram interface is generally proportional to the number of parameters.
2. For **custom enumeration types** (such as socket names and system states) and record types, each enumeration or field is listed on a separate line. Custom types usually involve custom design and engineering, resulting in an increased number of SLOC.
3. For **predefined enumeration types** (such as keyboard keys and compass directions), enumerations are listed on the fewest number of lines possible without loss of readability. These types generally require no custom engineering.
4. **Initialization of composite objects** (such as records and arrays) is listed with one component per line. Each of these assignments represents a custom statement; an "others" clause is typically used for non custom assignments.

2. SUBSYSTEM PROCESS IMPROVEMENTS

- Overall, the Common Subsystem subsidized much of the groundwork for the PDS and STRATCOM subsystems-namely, the process definition, the tools, and the reusable architecture primitives.
- With each successive subsystem, productivity and quality improved significantly.
- This is the expectation for a mature software process such as the one developed and evolved on CCPDS-R. The PDS Subsystem was delivered at 40% of the cost per SLOC of the Common Subsystem, and the STRATCOM Subsystem at 33 %.

3. sea RESOLUTION PROFILE

- The average change costs evolved over time into a fairly constant value of 16 hours per change. This effort included analysis time, redesign, recode, and retest of the resolution.

4. CSCI PRODUCTIVITIES AND QUALITY FACTORS

- Productivities for the CSCIs are not absolute. The subsystem productivity is based on a total effort of approximately 1,800 staff-months. This includes all management, development, and test resources. The individual productivities of each CSCI were normalized. Productivities are described from two perspectives: **SLOC per staff-month** and **ESLOC per staff-month**.

These data leads to the following conclusions:

- NAS was an extremely complex software engineering problem, requiring and achieving both high performance and reusability. It had an exceptional team, was based on an existing prototype, and had adequate schedule.
- SSV had very high absolute productivity because the automatically generated code, from custom CASE tools, was contained mostly within this CSCL. The above-average team on SSV also contributed to the high productivity.
- DCO was fairly average on all counts but accommodated substantial requirements volatility in the display interface without a contract amendment. The design of this CSCI and the performance of the team were far better than these numbers would indicate.

- TAS had a very low productivity despite being the simplest and most well understood software. The main reason was far less ambitious than the plans for other teams and team was located off-site, with highly constrained development environment resources.
- CMP had a very high cost of change and low productivity for no obvious technical reason. To ensure technical integrity, the inherent missile warning algorithm changes were closely scrutinized by many stakeholders. The coordination of this process resulted in very high overhead in CMP productivity and changes.
- CCO had the worst quality metrics, This was due primarily to a design that did not foresee a major message set change and therefore resulted in fairly broad and hard-to-resolve breakage. The CCO team was also perhaps the most difficult to transition (culturally) to the process, metrics, and demonstration approach used on CCPDS-R.

9. People factors



CCPDS-R used **two unique** approaches to managing its people.



- The first was the core team concept, which focused on **leveraging the skills** of a few experts across the entire team.
- The second was targeted at actively **avoiding attrition**.
- CCPDS-R was TRW's first large Ada project, and management was concerned that personnel trained by the project would become attractive targets for opportunities elsewhere inside and outside the company.
- As a result of the overall management approach to CCPDS-R, there was very little attrition of people across the Common Subsystem, with most of the engineering team transitioning to new assignments at planned points in the life cycle.
- Contrary to initial expectations, the PDS and STRATCOM subsystems were overlapped enough with the Common Subsystem that most of the people employed were new to the project.
- **Core team concept**
- **Leverage skills of a few experts across the entire team**
- **Avoid attrition**
- **Profit sharing of award fees**

9.1 Core Team



1. Developing the highest leverage components (mostly within the NAS CSCI). These components **resolved** many of the difficult computer science **issues** such as real-time scheduling, interpretation, communications, run-time configuration management, error processing, and distributed systems programming. As a result of encapsulating these complex issues in a small number of high-leverage components, the mainstream components were simpler and far less dependent on expert personnel.
2. **Setting the standards and procedures for design walkthroughs and software artifacts.** In general, the core team represented the frontline pioneers for most of the software activities. This team was generally the first team to conduct any given project workflow and built the first version of most artifacts. Consequently, the core team was intimately involved with setting precedent, whether it was the standards for a given activity or the format/ content of a given artifact.
3. **Disseminating the culture throughout the software organization.** The core team was truly a single, tight-knit team during the inception phase and for most of the elaboration phase. As the process and architecture stabilized, the team started to migrate, with several of its members taking on technical leadership roles on the various development and assessment teams. During construction and transition, a few members of the core team still maintained the architecture integrity across the entire project. However, there was also a set of globally minded individuals with strong relationships to the architecture team who became immersed in other areas of development and assessment. These team and personnel transitions proved to be an invaluable mechanism for maintaining a common culture.

9.2 AWARD FEE FLOWDOWN PLAN



- Award fee flowdown plan was that employees would share in the profitability of the project. TRW management agreed to allocate a substantial portion of the award fee pool at each milestone to be given directly to project employees. This additional compensation was to be distributed to the individuals based on their relative contribution and their longevity on the project.

The implementation of the award fee flowdown plan was intended to achieve the following objectives:

- Reward the entire team for excellent project performance
- Reward different peer groups relative to their overall contribution
- Substantially reward the top performers in every peer group
- Minimize attrition of good people

This was the basic operational concept of the plan:

- Management defined the various peer groups (systems engineering, software engineering, business administration, and administration).
- Every 6 months, the people within **each peer group ranked** one another with respect to their contribution to the project. The manager of each peer group also ranked the entire team. The manager compiled the results into a global performance ranking of the peer group.
- Each award fee was determined by the customer at certain major milestones. Half of each award fee pool was distributed to project employees.
- The algorithm for distributions to project employees was fairly simple. The general range of additional compensation relative to each employee's salary was about 2% to 10% each year.

10. CONCLUSIONS



- TRW and the Air Force have extensively documented the successes of architecture-first development on CCPDS-R.
- This project achieved twofold increases in **productivity and quality** along with on-budget, on-schedule deliveries of large mission-critical systems.
- The success of CCPDS-R is due, in large part, to the balanced use of **modern technologies, modern tools, and an iterative development** process that is substantially
- One of the primary improvements that was enabled by the CCPDSR software approach was the **teamwork** between the customer, user, and contractor. Continuous bartering, negotiation, and interpretation of the contract deliverables were productive in making real progress and ensuring that each phase of the life cycle resulted in a win-win situation for all stakeholders.
- CCPDS-R maintained **good project performance** and **non adversarial relationships** among stakeholders throughout the life cycle while continuously absorbing a moderate level of requirements volatility.
- Table summarizes the numerous dimensions of improvement incorporated into the CCPDS-R project.

<u>CCPDS-R technology improvements</u>		
PARAMETER	MODERN SOFTWARE PROCESS	CCPDS-R APPROACH
Environment	Integrated tools	DEC/Rational custom tools
	Open systems	VAX/DEC-dependent
	Hardware performance	Several VAX family upgrades
	Automation	Custom-developed change management system, metrics tools, code auditors
Size	Reuse, commercial components	Common architecture primitives, tools, processes across all subsystems
	Object oriented	Message-based, object-oriented architecture
	Higher level languages	100% Ada
	CASE tools	Custom automatic code generators for architecture, message input/output, display format source code
	Distributed middleware	Early investment in NAS development for reuse across multiple subsystems
Process	Iterative development	Demonstration, multiple builds, early delivery
	Process maturity models	Level 3 process before SEI CMM definition
	Architecture first	Executable architecture baseline at PDR
	Acquisition reform	Excellent customer/contractor/user teamwork; highly tailored 2167 A for iterative development
	Training	Mostly on-the-job training and internal mentoring

Future Software Project Management Practices

Rapid technology improvement ensures increased efficiency and higher productivity in every business whether it is small or large. Here we will see how technology is scaling our business and project management in an altered way and what the future plans are.

Basically, project management is an applied science that doesn't have any dimension. But yes, it has a dimension that is carried by factors of computational and managerial skills.

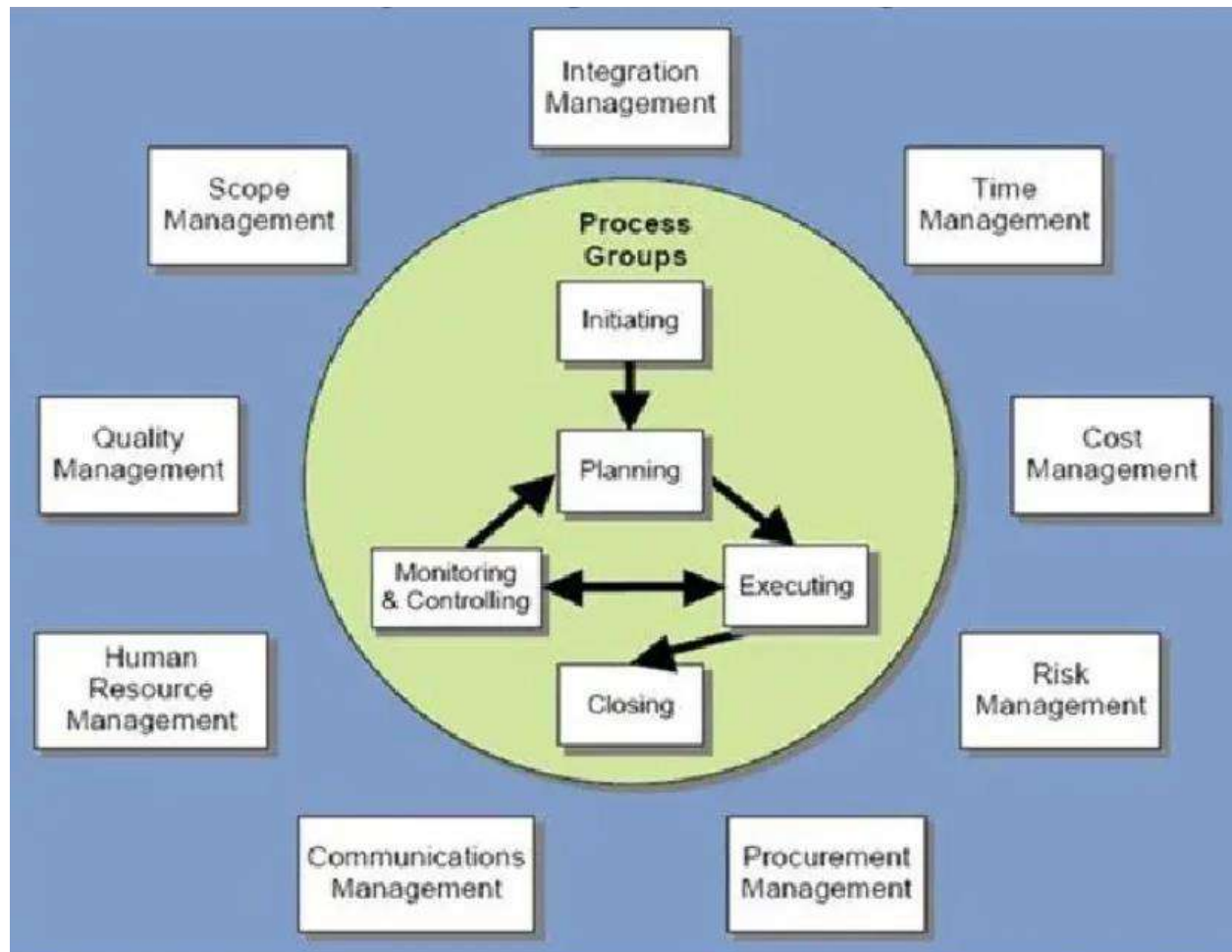
But the question is what the future of project management holds?

Essentially, a project is a process that is meant to complete a service, a goal or product within a fixed budget, time, and resource.

In earlier days, projects had to be managed at any cost to be completed. But there was no certain management process that was discovered to manage the process and projects.

This was the biggest issue for which the managers and the entire organization had to face a lot.

What a project management process exactly looks like? Here's the answer in a pictorial form;



If we consider project management as a science, then it can be identified and applied as a science.

All the time, its application needs to improvise which has led to the quick development of a project. Its impact will definitely increase with time.

Project management is now available for those activities which don't have a physical dimension. You can apply techniques of project management if you are not producing or carrying goods that have a physical existence.

Future of project management look like

Every manager's aim is to complete a project or task within the given time period and available resources that too in his/her control.

Technology is playing and will play a big role in project monitoring and execution. Through project management technology, liability within each team member will increase while executing a project and its delivery.

If you are handling a project with a team, then it is important that every team member is skilled enough according to the requirements of the project.

So that it will go smoothly and you can finish your project in the proper time span without compromising the quality. If the team members are not capable to handle some work, then the manager has to change the members.

Project management is carried out by specialized teams whose skills are owned by their knowledge of certain computational skills.

Today, the software has become more user-friendly. So it has become a mandatory one to be known to the skills and aspects of software so that you can take out the project in an advanced way.

Latest project management tools and methods are emerging. Many tools will need to undergo major changes to fulfill the new demands. Seeing the competition in the market, everyone needs updated software or method so that they can beat the market.

As per the project requirements, responsibilities of every team member will increase. Every team member will need to be the best in their roles and responsibilities in their projects.

Future projects are hoping to be totally paperless with the required information being shared over the internet. PM Software will be advanced that will gather information automatically.

APPENDIX-B

THE COCOMO COST ESTIMATION MODEL:

The **COCOMO (Constructive Cost Model)** is one of the most widely used and well-documented algorithmic models for estimating the **effort, cost, and schedule** for a software development project.

It was originally introduced by **Barry Boehm in 1981** and has since been updated to **COCOMO II** to account for modern software development practices. The evolution of COCOMO in software engineering economics over the past 20 years. The COCOMO estimating equations follow this simple form:

$$Effort = C1 EAF (Size)^{P1}$$

$$Time = C2 (Effort)^{P2}$$

Where:

Effort = number of staff – months

C1 = a constant scaling coefficient for effort

EAF = an effort adjustment factor that characterizes the domain, personnel, environment, and tools used to produce the artifacts of the process

Size = size of the end product (in human-generated source code), measured by the number of Delivered Source Instructions (DSI) required to develop the required functionality

P1 = an exponent that characterizes the economies of scale inherent in the process used to produce the end product, in particular the ability of the process to avoid non – value – adding activities (rework, bureaucratic delays, communications overhead)

Time = total number of months

C2 = a constant scaling coefficient for schedule

P2 = an exponent that characterizes the inherent inertia and parallelism in managing a software development effort

B.1 COCOMO

The COCOMO model is a software cost estimation model developed by Barry Boehm based on analysis of over 60 projects. It uses regression formulas to estimate effort, cost, and schedule based on project size (lines of code) and characteristics. The original 1981 COCOMO model (COCOMO I) has been

updated as COCOMO II to account for modern development approaches. COCOMO consists of basic, intermediate, and detailed forms with increasing accuracy. It estimates effort as a function of size and cost drivers related to product, hardware, personnel, and project attributes.

Basic Effort and Schedule Estimating Formulas

These were the original COCOMO cost estimation relationships:

Organic mode

$$Effort = 3.2 EAF (Size)^{1.05}$$

$$Time(in\ months) = 2.5 (Effort)^{0.38}$$

Semidetached mode

$$Effort = 3.0 EAF (Size)^{1.12}$$

$$Time\ (in\ months) = 2.5 (Effort)^{0.35}$$

Embedded mode

$$Effort = 2.8 EAF (Size)^{1.2}$$

$$Time(in\ months) = 2.5 (Effort)^{0.32}$$

Where:

Effort = number of staff – months

EAF = product of 1.5 effort adjustment factors (Table B -1)

Size = number of delivered source instructions (in units of thousands of lines of code)

The effort adjustment factor (EAF) multiplier represents the combined effects of multiple parameters. These parameters allow the project to be characterized and normalized against the projects in the COCOMO database. Each parameter is assessed as very low, low, nominal, high, or

very high. The effect of each parameter setting is a multiplier that typically ranges from 0.5 to 1.5. The product of these 15 effects is used as the coefficient in the cost equation.

COCOMO project characterization parameters

IDENTIFIER	EFFORT ADJUSTMENT FACTOR	PARAMETER RANGE	POTENTIAL IMPACT
RELY	Required reliability	0.75 – 1.40	1.87
DATA	Database size	0.94 – 1.16	1.23
CPLX	Product complexity	0.70 – 1.65	2.36
TIME	Execution time constraint	1.00 -1.66	1.66
STOR	Main storage constraint	1.00 – 1.56	1.56
VIRT	Virtual machine volatility	0.87 – 1.30	1.49
TURN	Computer turnaround time	0.87 – 1.15	1.32
ACAP	Analyst capability	1.46 – 0.71	2.06
AEXP	Applications experience	1.29 – 0.82	1.57
PCAP	Programmer capability	1.42 – 0.70	2.03
VEXP	Virtual machine experience	1.21 – 0.90	1.34
LEXP	Language experience	1.14 – 0.95	1.20
MODP	Use of modern practices	1.24 – 0.82	1.51
TOOL	Use of software tools	1.24 – 0.83	1.49
SCED	Required development schedule	1.23 – 1.10	1.23

Assumptions:

Several assumptions are inherent in the COCOMO formulas. Delivered source instructions all (non comment) lines of computer – processed code. The development life cycle starts at the beginning of product design and ends with acceptance test at the conclusion of the integration and test phase. A staff – month consists of 152 hours. The project will be managed. The project will experience stable requirements.

Life – Cycle Description:

The COCOMO life cycle includes five basic phases: plans and requirements, product design, detailed design, code and unit test, and integration and test. COCOMO provides recommendations for disturbing the effort and schedule across the basic phases of the conventional waterfall model.

Software Work Breakdown Structure:

Work breakdown structures are organized around the product subsystems at the higher levels and around the detailed activities at the lower levels. The standard activities estimated by COCOMO and included in the software WBS are requirement analysis, product design, and programming; test planning, verification and validation, project office functions, configuration management and quality assurance, and manuals. COCOMO also recommends a top – level distribution of efforts across the activities of the WBS.

Effort and schedule partition across conventional life – cycle phases

ACTIVITY	EFFORT (%)	SCHEDULE (%)
-----------------	-------------------	---------------------

Plans and requirements	(+8)	(+36)
Product design	18	36
Detailed design	25	18
Code and unit testing	26	18
Integration and test	31	28

Default effort allocations across COCOMO WBS activities

ACTIVITY	BUDGET
Requirements analysis	4
Product design	12
Programming	44
Test planning	6
Verification and Validation	14
Project office	7
Configuration management and quality assurance	7
Manuals	6

A typical COCOMO Project Profile:

The following discussion focuses on a specific example project in order to illuminate some of the project planning implications. Assume a large, 100,000 source – line (100 - KDSI) mission critical systems built under contract for a government agency. COCOMO would estimate 900 staff – months for development plus 72 staff – months for requirements specification for this project. The schedule required would be 22 months from initiation of design through test, plus 8 months for requirements.

B.2 Ada COCOMO

In the mid – 1980s, TRW confronted the challenge of transitioning several projects to Ada. In some cases, a government mandate was the forcing function. In other cases, the project engineers believed that Ada technology was critical to a competitive bid and successful performance. The first prototype of Ada COCOMO on an internal research and development project was developed in 1985.

Example: 100,000 SLOC projects that requires 972 staff – months of effort and 30 months of schedule

Effort

$$\begin{aligned}
 &= 2.8 \text{ EAF } (KDSI)^{1.2} \\
 &= 2.8 (1.28)(100)^{1.2} \\
 &= 900 \text{ staff-months of} \\
 &\quad \text{development effort} \\
 &+ 72 \text{ staff-months for plans,} \\
 &\quad \text{requirements} \\
 \hline
 &= \mathbf{972 \text{ staff-months of total effort}}
 \end{aligned}$$

The total EAF (1.28 in this example) is the product of all individual cost driver effects.

Cost Driver	Setting	Effect
Language experience	Nominal	1.0
Schedule constraint	Nominal	1.0
Database size	Nominal	1.0
Turnaround time	Nominal	1.0
Virtual machine experience	Nominal	1.0
Virtual machine volatility	Nominal	0.88
Use of software tools	High	1.0
Modern programming practices	Nominal	1.0
Storage constraint	Nominal	1.0
Applications experience	Low	1.10
Timing constraint	Nominal	1.0
Required reliability	High	1.15
Product complexity	High	1.15
Personal/team capability	Nominal	1.0

Effort adjustment factor = 1.2

Time

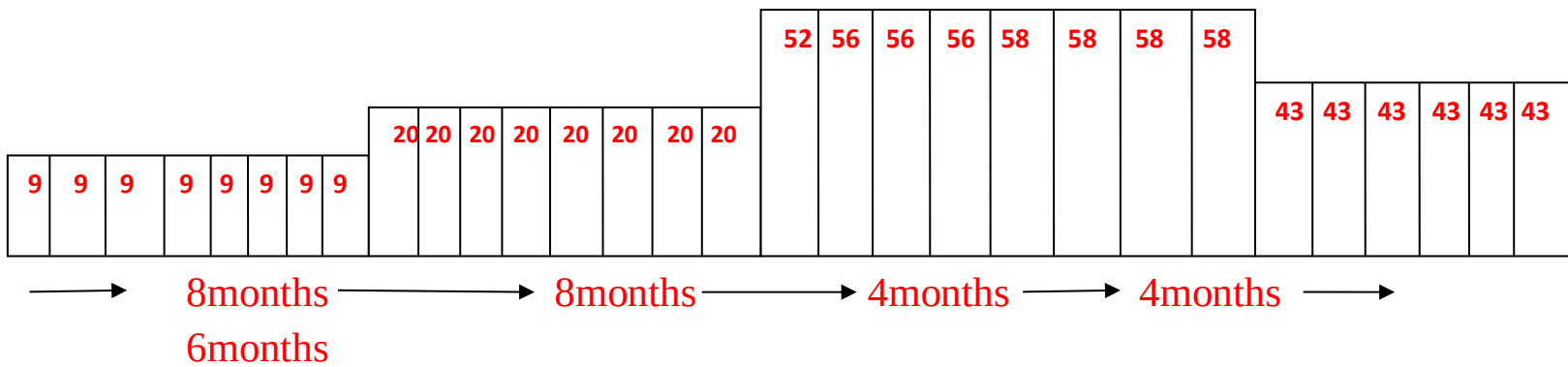
$$= 2.5 (Effort)^{0.32}$$

$$= 2.5 (900)^{0.32}$$

= 22 months of development schedule

= 30 months

Staffing profile and Activity Mix: Profile of conventional project



Plans and requirements	Product design	Detailed design	Code and testing	Integration and test
------------------------	----------------	-----------------	------------------	----------------------

Staff- months

The COCOMO life – cycle schedule distributions and activity distributions vary depending on the scale, domain, and business context. The schedule phases and activity mix illustrated here are typical.

Requirement analysis 36

4%

Product design 108

12%

Programming 398

44%

Test planning 54

6%

Verification and validation 126

14%

Project office

7%

Configuration management

7%

63

6%

Includes detailed design, coding and unit testing

This initial development of Ada COCOMO was just one of the activities in a three- prolonged approach:

4. Develop a set of architectural foundation components in Ada to measure compiler performance and provide a reusable set of foundation components for command and control systems such as CCPDS –R.
5. Develop a next-generation process description to exploit iterative development techniques and architecture- first, demonstration-based approach. This Ada process model [Royce, Walker, 1990b] was a major step toward a modern process for use in defense domain projects.
6. Develop an Ada version of COCOMO to describe the cost and schedule benefits of this new technology and process.

The Ada COCOMO cost estimation relationship was as follows:

$$\begin{aligned}\text{Effort} &= 2.8 \text{ EAF } (\text{Size})^P \\ \text{Time} &= 2.5 \text{ EAF } (\text{Effort})^{0.32}\end{aligned}$$

Where:

Effort = number of staff – months

EAF = product of 19 effort adjustment factors

Size = number of delivered source instructions (in units of
thousands of lines of code)

P = process exponent

Time = total number of months

Ada COCOMO improvements t the effort adjustment factors

Identifier	Effort Adjustment Factor	Ada COCOMO Perturbations
RELY	Required reliability	Changes to underlying effects(positive impact)
DATA	Database size	No change
CPLX	Product complexity	Changes to underlying effects (positive impact)
RUSE	Required level of reuse	New effort for complexity of reusable components
SECU	Security constraints	New effect for classified projects
TIME	Execution time constraint	No change
STOR	Main storage constraint	No change
VIRT	Virtual machine volatility	Deleted(split into two new factors)
VMVH	Host VM volatility	New effect accommodating host aspects of VIRT
VMVT	Target VM volatility	New effect accommodating target aspects of VIRT
TURN	Computer turnaround time	New level of interactive response(positive impact)
ACAP	Analyst capability	Changes to underlying effects (more impact)
AEXP	Applications experience	No change
PCAP	Programmer capability	Changes to underlying effects(less impact)
VEXP	Virtual machine experience	No change
LEXP	Language experience	Changes to underlying effects(more impact)
MODP	Use of modern practices	Changes to underlying effects(more impact)
TOOL	Use of software tools	New levels of automation support
SCED	Required development schedule	Changes to underlying effects(less impact)

Some of the techniques, tools, and technologies of the Ada environment:

7. A small core design team with expertise in software architecture and the applications domain
8. An early focus on executable architecture skeletons for demonstrating critical components and system-level threads and for exposing risk
9. Incremental and separate detailed design walkthroughs for components and builds rather than a monolithic Critical Design Review (CDR) across the whole system
10. Continuous integration via Ada compilation and architecture-first development
11. Test program and requirements verification focused engineering string tests(now called use cases) and component stand-alone tests
12. Self-documenting Ada code and big-picture descriptions instead of massive, detailed design documents that describe the as-built design
13. Automated metrics derived from the evolving code baselines

The Ada COCOMO process exponent ranged from 1.04 to 1.24 and was defined from the combined effects of the following four parameters:

14. Ada process model experience. This process maturity rating ranged from “no familiarity” with the process to “successfully employed on multiple projects”.
15. Design thoroughness at PDR.This parameter characterized the level of design detail inherent in the design baseline demonstrated at PDR. It ranged from “little thoroughness (20%)” to “complete thoroughness (100%).”
16. Risk eliminated at PDR.This parameter assessed the level of uncertainty inherent in the project at PDR, after which full-scale development is initiated. It ranged from “little risk resolution (20%)” to “complete resolution (100%)”.

17. Requirements volatility during development. This parameter ranged from “many large changes” to “no changes”, characterizing the amount of process turbulence confronted by the project.

B.3 COCOMO II

The COCOMO II project [Boehm et al., 1995; Horowitz, 1997] is an effort being performed by the USC Center for software engineering, with the financial and technical support of numerous industry affiliates. The objectives of this project are threefold:

- To develop a software cost and schedule estimation model for the life-cycle practices of the 1990s and 2000s.
- To develop software cost database and tool support for improvement of the cost model.
- To provide a quantitative analytic framework for evaluating software technologies and their economic impact.

USC speculates that the post-2000 software marketplace will include five distinct populations:

18. End-user programmers (55 million) generating spreadsheets or database queries.
19. Component developers (60,000) generating end-users applications and composition aids.
20. Component integrators (700,000) building applications rapidly from existing GUI builders, database / object managers, middleware, and domain-specific components.
21. System integrators (700,000) dealing with larger scale systems, unprecedented systems, few-of-a-kind applications, embedded

systems requiring up-front engineering, and other substantial custom software development.

22. Infrastructure developers (750,000) developing domain-independent components such as operating systems, database management systems, networks, and user interface frameworks.

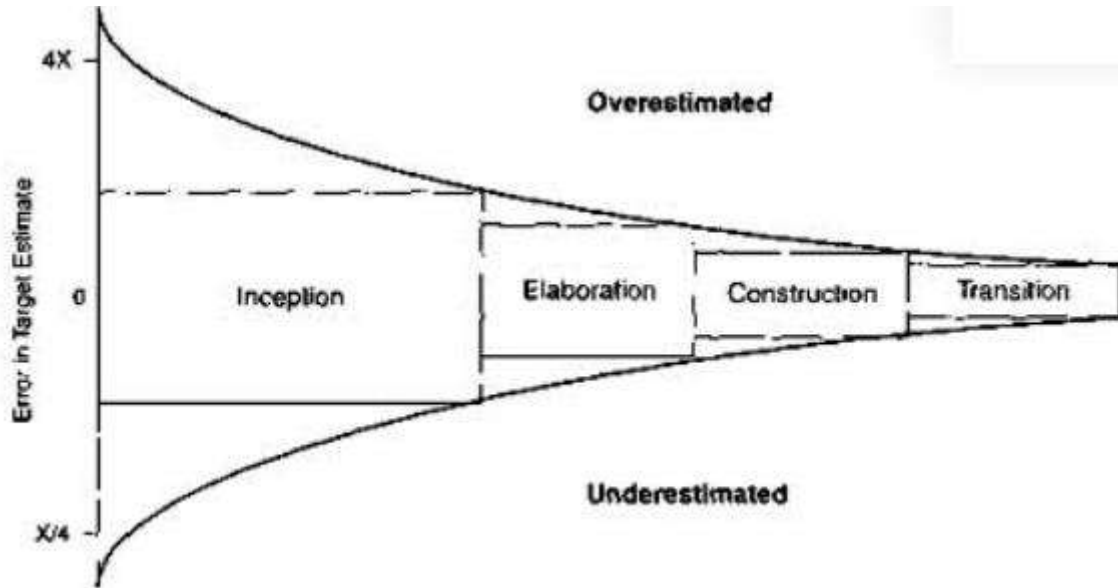
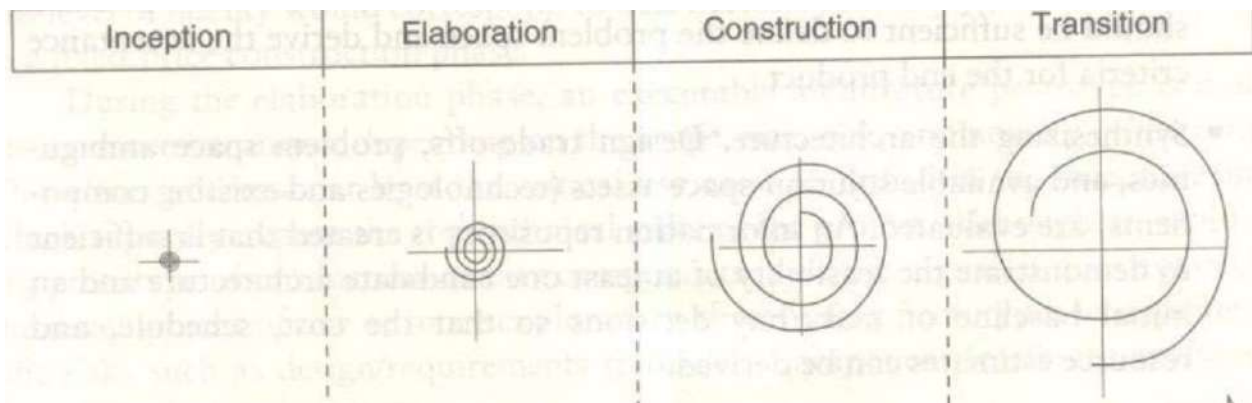


FIGURE H-2. Software estimation over a project life cycle



COCOMO II Cost Models

Prototyping model	Early Design Model	Post-Architecture Model
Coarse inputs Low-fidelity estimates Rough requirements Architecture concepts	Well-understood project Moderate fidelity estimates Well-understood requirements Well-understood architecture	Detailed project characterization High-fidelity estimates Stable requirements baseline Stable architecture baseline

COCOMO II estimation over a project life cycle

$$\text{Effort} = 2.5 E_{Arch} (\text{Size})^P$$

Where

Effort = number of staff-months

E_{Arch} = product of seven early design effort adjustment factors

Size = number of function points (preferred) or KSLOC

P = process exponent

The post-architecture cost estimating equation is as follows:

$$\text{Effort} = 2.45 E_{App} (\text{Size})^P$$

Where

Effort = number of staff-months

E_{App} = product of 17 post-architecture effort adjustment factors

Size = number of KSLOC (preferred) or function points

P = process exponent

Identifier	Complexity Effort Adjustment Factors
Product complexity	RELY-DATA-CPLX-DOCU
Required reuse	RUSE
Platform difficulty	TIME-STOR-PVOL
Personnel experience	AEXP-PEXP-LTEX
Personnel capability	ACAP-PCAP-PCON
Facilities	TOOL-SITE
Schedule	SCED

Early design model effort adjustment factors

Identifier	Effort Adjustment Factor	COCOMO II Perturbations
RELY	Required reliability	No change from COCOMO
DATA	Database size	No change from COCOMO
CPLX	Product complexity	No change from COCOMO
RUSE	Required reuse	No change from Ada COCOMO
DOCU	Documentation	Added; suitability of documentation to the life-cycle needs
TIME	Execution time constraint	No change from COCOMO
STOR	Main storage constraint	No change from COCOMO
PVOL	Platform volatility	Combined VMVH and VMVT parameters of Ada COCOMO into platform volatility
ACAP	Analyst capability	No change from COCOMO
AEXP	Applications experience	No change from COCOMO
PCAP	Programmer capability	No change from COCOMO
PEXP	Platform experience	Expanded platform experience from virtual machine experience
PCON	Personnel continuity	New parameter
LTEX	Language / tool experience	Changed to include both tool and language experience
SITE	Multiple – size development Team communications	New parameter for degree of collection and degree of automation
TOOL	Use of software tools	No change from COCOMO
SCED	Required development schedule	No change from COCOMO

COCOMO II uses the same exponent for the early design and the post – architecture models. The process exponents can range from (1.01...1.26) and is defined as the combined effects of the following five parameters:

1. **Application precededence**: the degree of domain experience of the development organization.
2. **Process flexibility**: the degree of contractual rigor, ceremony, and change freedom inherent in the project contractor, life-cycle activities, and stake-holder communications.
3. **Architecture risk resolution**: the degree of technical feasibility demonstrated before commitment to full-scale production.
4. **Team cohesion**: the degree of cooperation and shared vision among stake holders (buyers, developers, users, and maintainers, among others)
5. **Process maturity**: the maturity level of the development organization, as defined by the Software Engineering Institute's Capability Maturity Model

UNIT – V

MODERN PROJECT PROFILES

A modern process framework uses several critical approaches to resolve these issues. These approaches make project progress, risks, integration, and results more visible throughout the life cycle.

Protracted Integration and Late Design Breakage

Protracted integration and late design breakage are resolved by forcing integration into the engineering stage. This is achieved through continuous integration of an architecture baseline, supported by executable demonstrations of the primary scenarios.

In modern projects, integration is therefore performed early and continuously. This helps verify the design artifacts throughout development rather than waiting until the later stages.

Late Risk Resolution

Late risk resolution is addressed by emphasizing an architecture-first approach. In this approach, the high-leverage elements of the system are elaborated early in the life cycle.

As a result, the major technical risks are identified and resolved earlier, reducing the possibility of serious problems in the later stages.

Requirements-Driven Functional Decomposition

The analysis paralysis associated with requirements-driven functional decomposition is avoided by organizing lower-level specifications according to the content of releases rather than according to product decomposition.

Thus, specifications are organized around releases instead of being divided mainly by subsystems, components, and similar product elements.

Adversarial Stakeholder Relationships

Adversarial stakeholder relationships are avoided by providing more tangible and objective results throughout the life cycle.

Regularly demonstrating concrete results gives stakeholders a clearer understanding of the actual progress of the project and reduces conflicts between them.

Focus on Documents and Review Meetings

The conventional focus on documents and review meetings is replaced by a focus on demonstrable results and well-defined sets of artifacts.

Modern projects use more rigorous notations and extensive automation to support a paperless environment. The emphasis is therefore placed on producing and demonstrating meaningful results rather than depending mainly on documents and meetings.

Key Characteristics of Modern Project Profiles

In modern project profiles, both positive and negative trends become more tangible early in the life cycle. This allows the project team to recognize the actual direction of the project at an earlier stage.

Early and continuous integration serves as a verification activity for the design artifacts. This helps ensure that the developing system remains consistent with its intended architecture.

Most of the critical risks are expected to be resolved by the end of the elaboration phase. Therefore, the construction and transition phases, when the project is generally at its most costly and complex, should be free from major surprises.

The major milestones focus on demonstrated results. This means that project progress is judged through tangible and demonstrable outcomes rather than only through documents or review activities.

1. CONTINUOUS INTEGRATION:

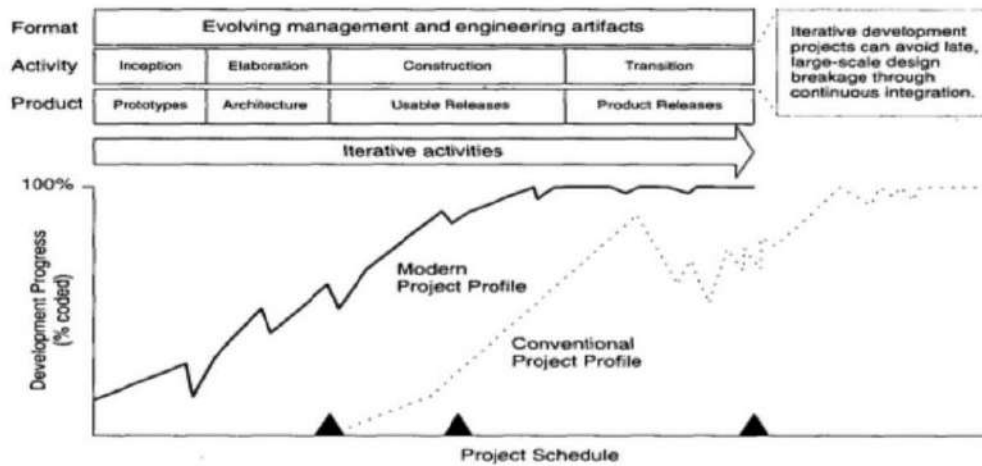


FIGURE 15-1. Progress profile of a modern project

CONTINUOUS INTEGRATION

Iterative development produces the **architecture first**. This allows integration to become a verification activity of the design phase and helps detect and resolve design flaws earlier in the life cycle.

Continuous integration avoids **big-bang integration** at the end of a project. Instead, integration is performed continuously throughout the project.

The architecture-first approach forces integration into the **design phase** through the construction of demonstrations. These demonstrations do not completely eliminate design breakage. Instead, they make design breakage occur during the engineering stage, where it can be resolved efficiently within the life cycle.

Continuous integration helps avoid the problems of downstream integration, late patches, and shoe-horned software fixes. The result is a more robust and maintainable design.

Progress Profile of a Modern Project

In an iterative development process, iterative activities continue throughout the project. The major phases are Inception, Elaboration, Construction, and Transition.

The product evolves through prototypes, architecture, usable releases, and product releases. Management and engineering artifacts also evolve throughout these phases.

The modern project profile shows development progress occurring continuously throughout the project. In contrast, the conventional project profile shows integration and major progress occurring much later.

Thus, continuous integration allows iterative projects to avoid late, large-scale design breakage.

Key Idea

Continuous integration means integrating the software continuously throughout the project instead of waiting until the end for a big-bang integration.

Workflow Cost Allocation

The table shows the difference in cost allocation between a **conventional process** and a **modern process**.

Differences in workflow cost allocations between a conventional process and a modern process

SOFTWARE ENGINEERING WORKFLOWS	CONVENTIONAL PROCESS EXPENDITURES	MODERN PROCESS EXPENDITURES
Management	5%	10%
Environment	5%	10%
Requirements	5%	10%
Design	10%	15%
Implementation	30%	25%
Assessment	40%	25%
Deployment	5%	5%
Total	100%	100%

The table shows that a modern process spends more effort on **management, environment, requirements, and design**, while a conventional process spends more on **implementation and assessment**.

The primary difference in a successful modern process is the **overall life-cycle expenditure for assessment and testing**. Conventional projects often spend about **40% or more of their total resources** on integration and testing because of inefficient integration and late discovery of major design problems.

Modern projects using a mature, iterative process can deliver a product with only about **25% of the total budget** consumed by assessment and testing activities. This indicates that continuous integration and early resolution of design issues can reduce the resources required for late integration and testing.

2. EARLY RISK RESOLUTION

The engineering stage of the life cycle includes the Inception and Elaboration phases. It focuses on identifying and resolving major risks before making large resource commitments in the production stage.

Conventional projects usually complete the easy tasks first to show early progress. A modern process focuses first on the important 20% of requirements, use cases, components, and risks. This is the basic idea of “architecture first.”

The 80/20 principle provides a useful perspective for managing software risks.

80/20 Principle

80% of the engineering is consumed by 20% of the requirements. The important driving requirements should be completely understood before committing resources to full-scale development.

80% of the software cost is consumed by 20% of the components. The cost-critical components should be elaborated first so that cost drivers are understood early.

80% of the errors are caused by 20% of the components. The reliability-critical components should be elaborated first to provide enough time for proper assessment and maturity.

80% of software scrap and rework is caused by 20% of the changes. The change-critical components should be elaborated first so that broad-impact changes occur while the project is still flexible.

80% of resource consumption, such as execution time, disk space, and memory, is consumed by 20% of the components. Performance-critical components should be elaborated first so that trade-offs among reliability, changeability, and cost-effectiveness can be resolved early.

80% of the progress is made by 20% of the people. Therefore, the initial team responsible for project planning and architecture design should be of the highest quality.

An adequate plan and architecture can succeed with an average construction team. However, an inadequate plan or architecture may not succeed even with an expert construction team.

Thus, early risk resolution focuses on addressing the most important requirements, components, changes, resources, and people early in the life cycle. This helps major risks to be resolved before the production stage.

3. EVOLUTIONARY REQUIREMENTS

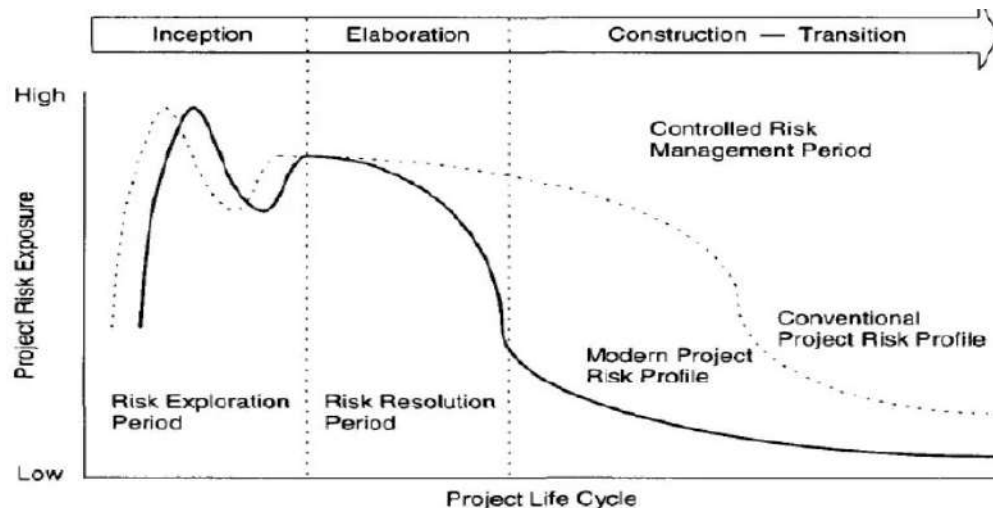


Figure 5.2 Risk profile of a typical modern project across its life cycle

EVOLUTIONARY REQUIREMENTS

Conventional Requirements Approach

Conventional approaches decompose **system requirements into subsystem requirements**, subsystem requirements into **component requirements**, and component requirements into **unit requirements**.

The requirements are organized so that **traceability is simple**. The approach gives importance to **requirements first and design second**.

The design structure naturally follows the requirements structure. Thus, **functional decomposition of the problem space** leads to functional decomposition of the solution space.

Modern Requirements Approach

Modern architectures use **commercial components, legacy components, distributed resources, and object-oriented methods**.

Therefore, requirements cannot always be easily traced to design elements. The relationships may be **1-to-1, many-to-1, 1-to-many, conditional, time-based, or state-based**.

Evolution of Requirements

Top-level system requirements are retained as the vision. Lower-level requirements are captured through **evaluation criteria attached to each intermediate release**.

These artifacts evolve along with the development process. As the life cycle progresses, requirements are understood with **increasing fidelity**.

This differs from conventional requirements management, where detailed understanding and traceability were pursued **too early in the life cycle**.

Risk Profile of a Modern Project

The figure shows the change in **project risk exposure** throughout the project life cycle.

The life cycle includes **Inception, Elaboration, Construction, and Transition**.

The early part is the **Risk Exploration Period**, where project risks are explored. It is followed by the **Risk Resolution Period**, where major risks are resolved.

The **Controlled Risk Management Period** continues as the project progresses.

In a **modern project**, risk exposure decreases earlier as major risks are resolved. In a **conventional project**, significant risk remains for a longer period.

Thus, the modern approach focuses on **early risk exploration and risk resolution** during the life cycle.

4. TEAMWORK AMONG STAKEHOLDERS

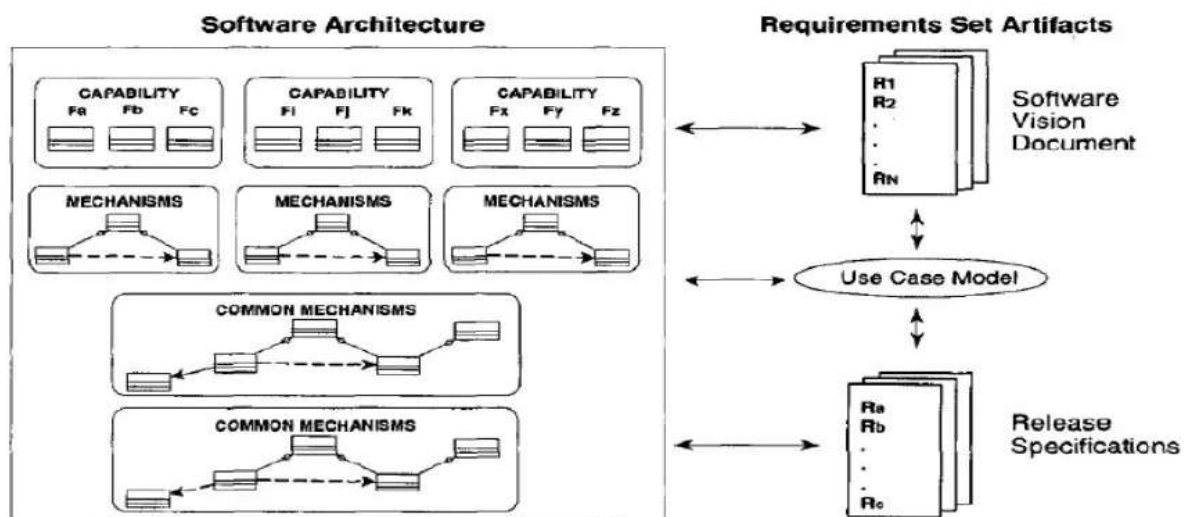


Figure 5.3: Organization of software components resulting from a modern process

Many aspects of the **classic development process** can create mutual distrust among stakeholders.

This makes it difficult to balance **requirements, product features, and project plans**. An iterative process helps stakeholders develop better working relationships.

Working Relationships Among Stakeholders

An iterative process allows **trade-offs** to be based on a common and objective understanding.

Customers, users, and monitors should have both **application expertise and software expertise**. They should focus on delivering a usable system rather than only enforcing standards and contract terms.

The contractor should be allowed to make a reasonable profit through good performance. The development organization should focus on **customer satisfaction, high product quality, and profitability**.

Demonstration of Intermediate Results

Moving from mostly **paper artifacts** to demonstrations of intermediate results is important for promoting teamwork among stakeholders.

Major milestones provide **tangible results and feedback** from the user's point of view.

The project should not move forward until the objectives of the demonstration are achieved.

After a demonstration and major milestone, the objectives can still be renegotiated. The results help stakeholders understand the **trade-offs among requirements, design, plans, and technology**.

Understanding Negative Results

In a modern iterative process, some results may appear negative.

Stakeholders must understand the difference between an apparently negative result and evidence of real progress.

A **design flaw discovered early**, when it is still inexpensive to fix, can be viewed as positive progress. Early discovery allows the problem to be resolved within the life-cycle goals.

Results of Major Milestones

Apparent Result	Real Result
Early demonstrations expose design issues and ambiguities.	Important assets and risks are exposed early and can be resolved within life-cycle goals.
The design is noncompliant so far.	Understanding of compliance improves through important requirements and use cases.
Requirements issues are exposed, but detailed traceability is lacking.	Requirements changes are balanced with design trade-offs.
The design is considered “ guilty until proven innocent. ”	Engineering progress and issues become tangible and can be included in the next iteration's plans.

5. PRINCIPLES OF SOFTWARE MANAGEMENT

Top 10 Software Management Principles

The **top 10 software management principles** describe the important practices of a modern software process. They help the project manager achieve successful and predictable software development.

1. Architecture-First Approach

The process should follow an **architecture-first approach**. Early focus on architecture provides a strong foundation for requirements, components, use cases, risks, and errors.

Important architectural components should become stable and well understood first. This helps reduce **scrap and rework** during the project life cycle.

2. Iterative Life-Cycle Process

An **iterative life-cycle process** should be established to confront risks early. It provides better risk management and more predictable performance.

Resolving critical issues early helps make the construction phase more predictable. It also reduces exposure to cost and schedule uncertainty.

3. Component-Based Development

Design methods should emphasize **component-based development**. Software complexity is largely related to the number of human-generated artifacts.

Making the solution smaller reduces **management complexity**.

4. Change Management Environment

A **change management environment** should be established. Iterative development involves different teams working concurrently on shared artifacts.

Therefore, **highly controlled baselines** are necessary to manage changes effectively.

5. Round-Trip Engineering

Change freedom should be enhanced using tools that support **round-trip engineering**. Automation reduces the time spent on overhead tasks.

This allows teams to spend more time on **engineering activities**.

6. Model-Based Notation

Design artifacts should be captured using rigorous, **model-based notation**. Engineering notation helps control complexity.

It also supports **objective assessment and automated analysis**.

7. Objective Quality Control and Progress Assessment

The process should support **objective quality control and progress assessment**. Progress and quality indicators should be obtained directly from evolving software artifacts.

This provides meaningful information about project trends and their relationship with requirements.

8. Demonstration-Based Approach

A **demonstration-based approach** should be used to assess intermediate artifacts. Integration should occur early and continuously throughout the life cycle.

The intermediate results should be **objective and tangible**.

9. Intermediate Releases

Intermediate releases should be planned using groups of usage scenarios with **evolving levels of detail**.

Requirements, designs, and plans should evolve in balance. This allows **useful software releases to be available early** in the life cycle.

10. Configurable Process

A **configurable process** should be established so that it is economically scalable. Methods, techniques, tools, and experience should be applicable across a broad domain.

This provides an improved **return on investment** across a line of business.

Balance Among the Principles

The software project manager should maintain the proper **balance among all 10 principles**. The main objective is to give appropriate emphasis to each principle throughout the project.

This balance supports the **fundamental software economics** of the project.

6. SOFTWARE MANAGEMENT BEST PRACTICES

Many software management best practices have been identified by software organizations and experts. The Software Acquisition Best Practices Initiative, sponsored by the U.S. Department of Defense, focused on improving software acquisition management processes.

The Airline Software Council also developed best practices for managing large-scale software projects. The Council included software industry experts, government practitioners, and experienced program managers.

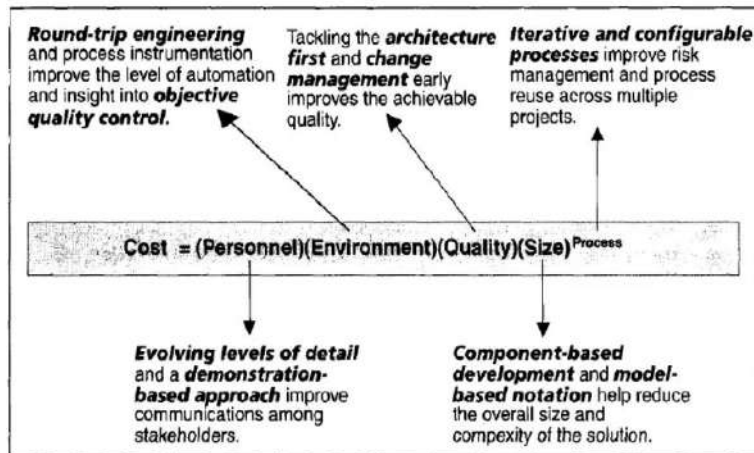


Figure 5.4: Balanced application of modern principles to achieve economic results

Its recommendations focused on improving software management and controlling project complexity.

1. Formal Risk Management

Formal risk management uses an **iterative process** to identify and address risks.

Risks are confronted continuously during the project instead of waiting until the end.

2. Agreement on Interfaces

Agreement on interfaces is related to the **architecture-first principle**.

Establishing the architecture early helps the team agree on external interfaces and important internal interfaces.

3. Formal Inspections

The assessment workflow should be used throughout the life cycle along with other engineering workflows.

Formal inspection is important because it helps identify critical architectural defects affecting multiple components.

4. Metric-Based Scheduling and Management

This practice is related to **model-based notation** and **objective quality control**.

Project progress and quality should be measured using metrics rather than depending only on subjective estimates.

5. Binary Quality Gates at the Inch-Pebble Level

Binary quality gates should be applied at the **inch-pebble level**.

Detailed planning should be performed only after the requirements and architecture are sufficiently understood.

This approach helps maintain a realistic development plan when requirements or architecture change.

6. Programwide Visibility of Progress Versus Plan

Project progress should be visible across the entire program.

Continuous communication among project team members helps compare actual progress with the planned progress.

7. Defect Tracking Against Quality Targets

Defect tracking is related to the **architecture-first** and **objective quality control** principles.

Defects and quality targets should be clearly identified and continuously tracked because they are important for project success.

8. Configuration Management

Configuration management helps control project complexity and track changes to project artifacts.

Automation is important in large projects because manual methods can become costly and error-prone.

9. People-Aware Management Accountability

People-aware management accountability is a management principle that recognizes the importance of people and management responsibility.

It emphasizes accountability in effective software management.

Additional Modern Principles

The Airlie Software Council's practices have significant overlap with modern software management principles.

Some additional principles are **configurability, component-based development, model-based development, and demonstration-based development**.

Component-based and model-based development help reduce development complexity.

Demonstration-based development encourages continuous integration throughout the life cycle and improves stakeholder communication.

Configurability was less important to the Council because its focus was mainly on large-scale, nationally important systems.

The two practices not included among the preferred principles were **inspections** and **binary quality gates at the inch-pebble level**. Although useful, they were considered overemphasized compared with other important principles.

NEXT GENERATION SOFTWARE ECONOMICS

Next-generation software economics is being practiced by some advanced software organizations. Many of the techniques, processes, and methods discussed in this book have already been practiced for several years.

However, a mature modern process is not yet the normal practice for the average software organization. This chapter presents several hypotheses about the future of software economics.

A general structure is proposed for a software cost estimation model that is better suited to the process framework. This approach can improve the accuracy and precision of software cost estimates and accommodate improvements in software economies of scale.

These improvements are enabled by advances in software development environments. Boehm's benchmarks of conventional software project performance

are also used to describe how the process framework can improve overall software economics.

NEXT-GENERATION COST MODELS

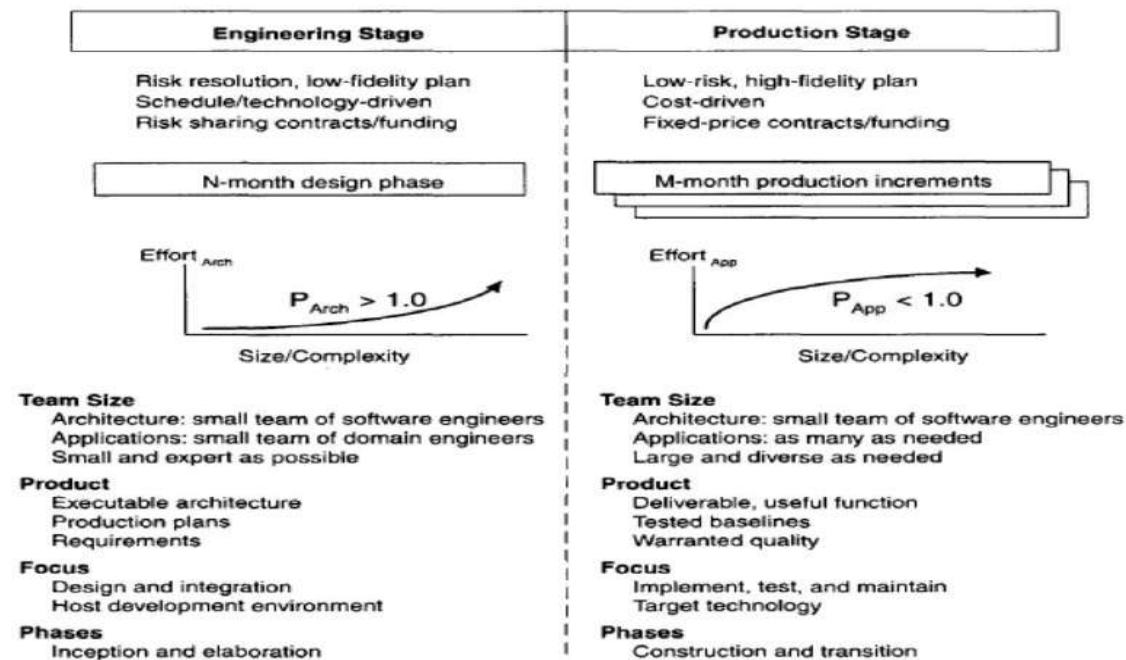


FIGURE 16-1. Next-generation cost models

Software experts have different opinions about software economics and its effect on software cost estimation. Important debates include source lines of code versus function points, economy of scale versus diseconomy of scale, and productivity measures versus quality measures.

Other debates include Java versus C++, object-oriented versus functionally oriented development, and commercial components versus custom development.

These disagreements make it difficult to separate facts from exaggeration. They also show that the software industry is still developing rapidly, with many competing technologies and techniques maturing at the same time.

One result is the continuing difficulty of predicting the resources required for a software development effort. Accurate estimates are possible, but honest estimates are often imprecise.

Improving estimation models is difficult when the project data used by them is noisy and highly uncorrelated. The models are also based on different process and technology foundations.

Many popular software cost models are not well matched to an iterative software process based on an architecture-first approach.

Although software cost estimation tools have improved their project experience databases, many cost estimators still use conventional process experience to estimate modern projects.

A next-generation cost model should be structured to support estimation of a modern software process. However, many software cost models are still mainly theoretical, and there is not enough empirical evidence to prove that these approaches are more accurate than today's models.

SEPARATION OF ENGINEERING AND PRODUCTION

A next-generation software cost model should explicitly separate architectural engineering for application production from an architecture-first process.

The cost of designing, producing, testing, and maintaining the architecture baseline depends on scale, quality, technology, process, and team skill.

The architecture cost model should still show some diseconomy of scale, with an exponent greater than 1.0. This is because architecture is mainly driven by research and development-oriented concerns.

Once an organization achieves a stable architecture, production costs should become an exponential function of size, quality, and complexity, with a more stable range of process and personnel influence.

The production-stage cost model should reflect an economy of scale, with an exponent less than 1.0.

Next-generation software cost models should therefore estimate large-scale architectures with economy of scale. As the system becomes larger, there is greater opportunity to automate and reuse common processes, components, and architectures.

NEXT-GENERATION COST MODELS

The cost model represents effort as :

$$**\text{Effort} = F(T_{\text{Arch}}, S_{\text{Arch}}, Q_{\text{Arch}}, P_{\text{Arch}})$$

- $F(T_{\text{App}}, S_{\text{App}}, Q_{\text{App}}, P_{\text{App}})**$

Time is represented as:

$$**\text{Time} = F(P_{\text{Arch}}, \text{Effort}_{\text{Arch}})$$

- $F(P_{\text{App}}, \text{Effort}_{\text{App}})**$

Here, T is the technology parameter, representing environment automation support.

S is the scale parameter, such as use cases, function points, and source lines of code.

Q is the quality parameter, such as portability, reliability, and performance.

P is the process parameter, such as process maturity and domain experience.

ENGINEERING STAGE

The engineering stage focuses on risk resolution and a low-fidelity plan.

It is mainly schedule/technology-driven and uses risk-sharing contracts or funding.

It contains an N-month design phase. The architecture effort shows a diseconomy of scale, where $P_{\text{Arch}} > 1.0$.

The architecture team is small and consists of software engineers. Application teams are small teams of domain engineers, with the smallest possible expert team.

The main products are the executable architecture, production plans, and requirements.

The focus is on design and integration and the development of the host development environment.

The major phases are inception and elaboration.

PRODUCTION STAGE

The production stage has low risk and uses a high-fidelity plan.

It is cost-driven and uses fixed-price contracts or funding.

It consists of M-month production increments. The production effort shows an economy of scale, where $P_{App} < 1.0$.

The architecture team remains small, while application teams can be as large and diverse as needed.

The main products are deliverable useful functions, tested baselines, and warranted quality.

The focus is on implementation, testing, and maintenance using the target technology.

The major phases are construction and transition.

AUTOMATION AND ECONOMY OF SCALE

In the conventional process, only a minimal level of automation supported planning, project control, and change management.

This resulted in labor-intensive workflows and a diseconomy of scale. The same problem existed for both individual projects and organizations handling multiple projects.

Next-generation environments and infrastructures are moving toward the automation and standardization of management activities.

This reduces the percentage of effort required for overhead activities as the scale of the project increases.

REUSE AND ECONOMY OF SCALE

Reusing common processes across multiple iterations of a project, multiple releases of a product, or multiple projects in an organization reduces many sources of diseconomy of scale.

Many sources of scrap and rework can be eliminated by applying previous experience and mature processes.

Trustworthy plans based on credible project performance and the reuse of reliable components can further reduce scrap and rework.

The reuse of components reduces the size of the production effort. Reuse of processes, tools, and experience directly affects the economies of scale.

PROBLEM SPACE AND SOLUTION SPACE

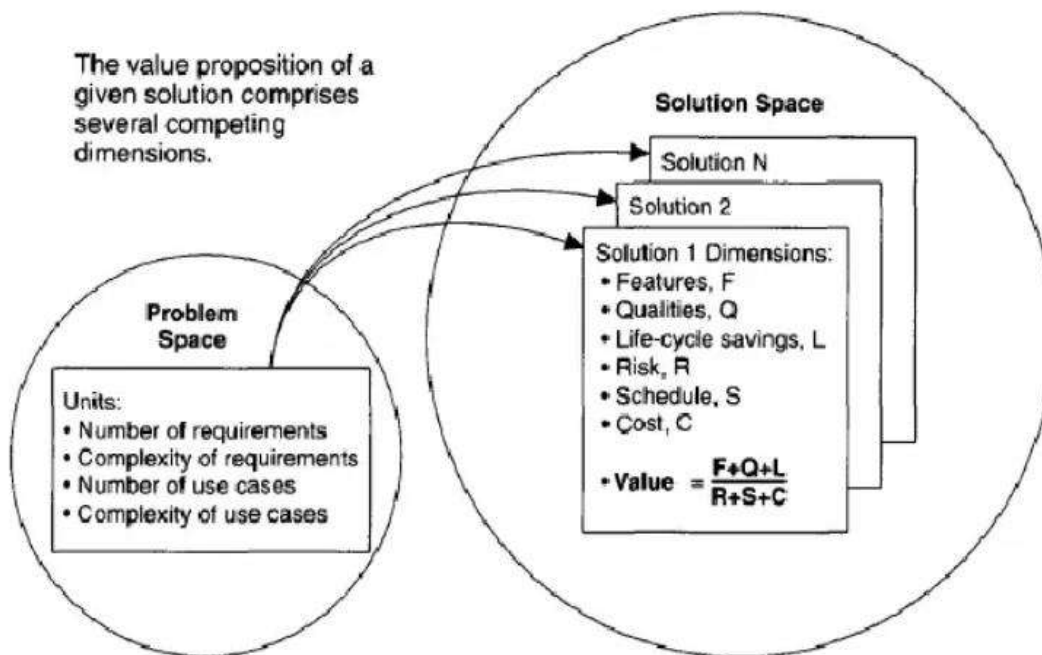


Figure 5.5 Differentiating potential solutions through cost estimation

Another important difference in the cost model is that architectures and applications have different units of mass, scale, and size.

Scale can be measured using architectural elements such as classes, components, processes, and nodes.

Size can be measured in SLOC or megabytes of executable code.

These measures are different from the problem space, which is described using discrete requirements or use cases.

The problem-space description mainly drives the definition of the solution space. However, a given problem can have many possible solutions.

Each solution can have a different value proposition. Therefore, cost is an important discriminator among potential solutions.

Cost estimates that are more accurate and precise can be derived from specific solutions to specific problems.

DIFFERENTIATING POTENTIAL SOLUTIONS THROUGH COST ESTIMATION

The problem space includes the number of requirements, complexity of requirements, number of use cases, and complexity of use cases.

The solution space contains several possible solutions to the problem.

Each solution can be evaluated using dimensions such as features (F), qualities (Q), life-cycle savings (L), risk (R), schedule (S), and cost (C).

The value proposition is represented as:

$$\text{Value} = (F + Q + L) / (R + S + C)$$

Thus, cost estimation helps differentiate potential solutions.

If none of the value propositions provides an acceptable solution to the problem, further candidate solutions must be considered or the problem statement must be changed.

FUNCTION POINTS VERSUS SLOC

There is a continuing debate between function point advocates and SLOC advocates.

Function points are considered more accurate for quantifying the scale of the architecture required.

SLOC more accurately represents the size of the components that implement the solution.

The collection of SLOC can be easily automated, and its precision can be easily improved.

However, the accuracy of SLOC as a measure of size is ambiguous. It can lead to incorrect interpretations when SLOC is used for absolute comparisons between different projects and organizations.

This problem is particularly important during the early phases of a project.

Later in the life cycle, SLOC can be useful for measuring the relative changes from month to month as the project moves toward releasable versions.

VALUE OF FUNCTION POINTS

Function points are useful because they better represent the overall scale of the solution, independent of the actual size and implementation language used for the final realization.

However, function points are not easily extracted from a rigorous representation format. Therefore, their automation and change tracking can be difficult or ambiguous.

A rigorous notation for design artifacts is necessary to improve the accuracy with which software scale can be estimated.

In the future, there may be an opportunity to automate a new measure of scale directly from design representations in UML.

TWO MAJOR IMPROVEMENTS IN NEXT-GENERATION COST ESTIMATION

Separation of Engineering and Production

Separating the engineering stage from the production stage will force estimators to distinguish between architectural scale and implementation size.

This will provide greater accuracy and more honest precision in life-cycle estimates.

Use of Rigorous Design Notations

Rigorous design notations such as UML can provide standardized units for measuring software scale.

These measures can be automated and tracked. They can also be traced more directly into the costs of production.

QUANTIFYING ARCHITECTURAL SCALE

Quantifying the scale of software architecture in the engineering stage is an important area for research.

Over the next decade, two major breakthroughs in the software process are expected through technology advances in the supporting environment.

The first breakthrough is the availability of integrated tools that automate the transformation between requirements, design, implementation, and deployment elements.

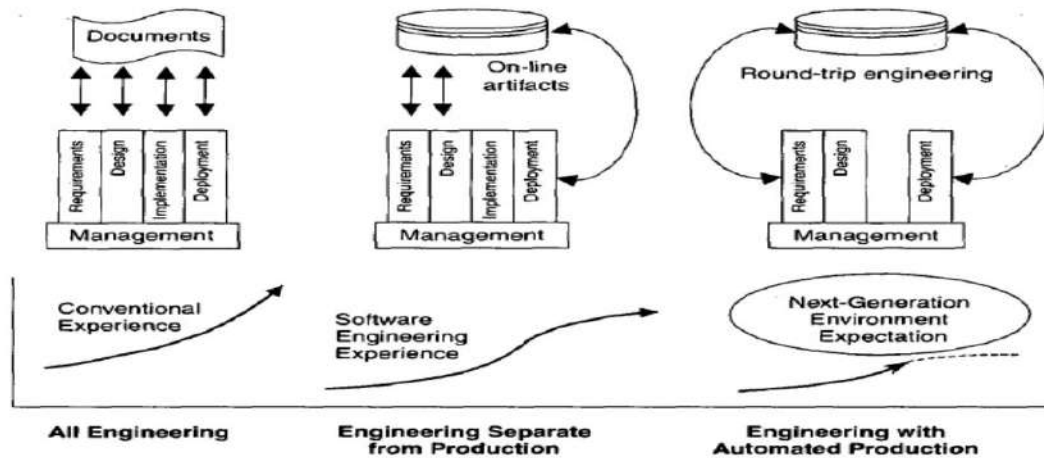
These tools would allow more comprehensive round-trip engineering among engineering artifacts.

The second breakthrough focuses on automating activities associated with human-generated source code.

This can eliminate the need for a separate implementation set. Technology advances could allow executable programs to be produced directly from UML representations without human intervention.

Visual modeling tools can already produce code subsets from UML models. The future goal is to produce complete software systems from UML models.

AUTOMATION OF THE CONSTRUCTION PROCESS



E 16-3. Automation of the construction process in next-generation environments

The next-generation environment aims to automate the construction process.

Engineering artifacts such as requirements, design, implementation, and deployment can be connected and managed through an integrated environment.

The development process is expected to move from conventional experience, to software engineering experience, and finally toward engineering with automated production.

ROUND-TRIP ENGINEERING

Round-trip engineering maintains relationships among different engineering artifacts.

Changes made to one engineering artifact can be reflected in related artifacts through the integrated development environment.

This provides better coordination among requirements, design, implementation, and deployment.

AUTOMATED PRODUCTION

The second major breakthrough is the automation of production activities.

It aims to produce implementation and deployment artifacts in an error-free, automated environment.

Software development could therefore change significantly, similar to the transition from traditional production to an automated “printing” process.

The overall objective is to move toward an environment where software can be produced automatically from engineering representations.

MODERN SOFTWARE ECONOMICS

1. Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases

Modern processes, component-based technologies, and architecture frameworks help reduce this cost difference.

An **architecture-first approach** can greatly improve the identification and resolution of architectural errors.

Therefore, modern processes give more importance to **early architecture work, insight, and risk-confronting activities**.

2. You can compress software development schedules 25% of nominal, but no more

This metric mainly applies to the **engineering stage** of the life cycle.

During this stage, the system’s intellectual content is developed.

After achieving a stable architecture, construction plans, and requirements, the **production stage** can be made more flexible.

Schedule compression can be achieved by reducing engineering effort across multiple projects or through **incremental development**.

3. For every \$1 you spend on development, you will spend \$2 on maintenance

This metric is difficult to generalize because different projects use different **maintenance models**.

Comparing only development and maintenance costs may not give meaningful results.

A better comparison is between **development productivity and maintenance productivity**.

In iterative development, the boundary between development and maintenance becomes less clear.

A mature iterative process and good architecture can reduce **scrap and rework**.

Therefore, the traditional 1:2 relationship may move toward a **one-for-one relationship**.

4. Software development and maintenance costs are primarily a function of the number of source lines of code

Traditionally, **source lines of code (SLOC)** were considered an important measure of software size and cost.

With modern component-based technologies, this relationship is becoming less clear.

Commercial components, reusable components, and automatic code generators reduce the importance of individual source lines.

Construction cost is also affected by the complexity of the **bill of materials**.

More components and more integration effort can increase development costs.

Therefore, future cost models should depend less on **SLOC** and more on the **number of components and their integration effort**.

5. Variations among people account for the biggest differences in software productivity

In software engineering, important productivity factors include **personal skills, teamwork, and motivation**.

A modern process should use highly capable people during the **engineering stage**.

During the production stage, teams may become larger.

At this stage, the process should depend less on individual experts and more on a **well-defined process**.

6. The overall ratio of software to hardware costs is still growing

In **1955**, the software-to-hardware cost ratio was **15:85**.

In **1985**, it became **85:15**.

The relationship may be different today because of changes in personal computers and software costs.

Hardware costs continue to become cheaper.

Processing power, storage, and network bandwidth provide more opportunities for **automation**.

Software environments can automate activities such as **bookkeeping, analysis, configuration control, and quality assurance**.

The next areas include **automated production and automated testing**.

7. Only about 15% of software development effort is devoted to programming

Software development has shifted from only focusing on programming languages and compilers.

Modern development also focuses on **process maturity, visual modeling, automated software quality, components, configuration management, and metrics**.

The amount of programming effort is still roughly **15%**.

However, modern programming is performed at a **much higher level of abstraction**.

Modern tools allow programmers to produce much more functionality with fewer human-written source lines.

8. Software systems and products typically cost 3 times as much per SLOC as individual software programs. Software-system products cost 9 times as much

This represents a **diseconomy of scale**.

Larger and more complex software systems can cost more for each source line than individual programs.

Modern processes and technologies can reduce this diseconomy.

When systems use a **common architecture, common environment, and common process**, an **economy of scale** can be achieved.

9. Walkthroughs catch 60% of the errors

This metric should not be treated as one of the most important software metrics.

Human inspections and walkthroughs may not expose the most critical issues.

They are mainly useful for helping to **resolve identified problems**.

The environment can detect many first-level inconsistencies and errors.

Important architectural issues can be identified through **demonstration and early testing** and resolved through human analysis.

10. 80% of the contribution comes from 20% of the contributors

This is a long-standing relationship in software management.

It provides an important philosophy for a **modern software management process**.

The principle should be considered throughout the **planning and conduct of software projects**.

MODERN PROCESS TRANSITIONS

Modern Process Transitions

Introduction

Successful software management is difficult. New technologies, processes, and tools can make management easier, but **management discipline** remains important for project success.

New technologies create new opportunities, new application areas, new automation possibilities, and new customers with different priorities.

These changes can affect traditional software management values. However, balancing **requirements, designs, and plans** remains the basic objective of software management.

The modern software management framework is not completely new. Many projects have already used some of these practices.

However, adopting these practices requires a **paradigm shift**. Some stakeholders and organizations may resist these changes.

Therefore, successful transition requires both process changes and important **culture shifts**.

CULTURE SHIFTS

Several culture shifts are required to move successfully to a modern software management process.

These changes help distinguish a genuine transition from a project that only creates the appearance of adopting a modern process.

1. Lower-Level and Mid-Level Managers Are Performers

Lower-level and mid-level managers should also be performers. There should be no “**pure managers**” in an organization or suborganization with 25 or fewer people.

Managers should understand the project directly and participate in planning and estimation.

A manager should take ownership of the plan and should participate in developing it.

2. Requirements and Designs Are Fluid and Tangible

Modern processes treat requirements and designs as **fluid and tangible**.

The conventional process focuses heavily on documents and less on producing tangible product increments.

Major milestones should therefore be based on actual product progress rather than only on completed documents.

An iterative process produces progressively more complete systems. These systems help validate the technical approach and resolve requirements and customer issues.

Stakeholders should focus on **real milestones** and useful incremental functionality.

3. Ambitious Demonstrations Are Encouraged

Early demonstrations are mainly used to **expose design flaws**.

They should not be used only to create an appearance of progress.

Stakeholders should not overreact to early mistakes or immature designs because early releases are based on goals rather than final requirements.

At the same time, development teams must properly follow up and resolve identified problems.

4. Good and Bad Project Performance Is More Obvious Earlier

In iterative development, both success and failure become visible earlier in the life cycle.

Early planning and architecture activities can strongly affect the success of the project.

Therefore, capable teams should perform the planning and architecture work properly.

If these activities are weak, even experienced programmers and testers may not achieve success.

5. Early Increments Will Be Immature

Initial deliveries cannot be expected to be complete, fully reliable, or at the required quality and performance level.

However, development organizations must improve the product through successive increments.

Measurements should show improvements, fixes, upgrades, and progress toward the required specification.

Customers may find early flaws difficult to accept, but later increments should demonstrate clear improvement.

6. Artifacts Are Less Important Early, More Important Later

During early stages, too much attention should not be given to artifact details such as **traceability, thoroughness, and completeness**.

Artifacts first need to reach a useful and stable baseline.

Detailed quality analysis becomes more useful after the artifacts become sufficiently stable.

Otherwise, valuable engineering time may be wasted improving artifacts that soon become obsolete.

7. Real Issues Are Surfaced and Resolved Systematically

Successful projects understand that requirements and designs evolve together.

They require continuous **negotiation, trade-off, and balancing of best value**.

A healthy project should clearly distinguish real issues from apparent issues.

This culture shift can affect almost every team in the project.

8. Quality Assurance Is Everyone's Job

Quality assurance should be **everyone's responsibility**, not only the responsibility of a separate QA group.

Quality assurance should be included in every role, activity, and artifact.

It should be measured using tangible progress and objective data rather than only checklists and inspections.

The project manager or designer should ensure that quality assurance is integrated into the process.

Traditional inspection by a separate team is replaced by **self-policing** in a mature organization.

9. Performance Issues Arise Early in the Life Cycle

Early performance issues are expected in a successful project.

Such issues do not necessarily indicate an immature design. They can indicate a **mature development process** because problems are identified early.

Early demonstrations allow developers to study performance and make necessary trade-offs before later releases.

10. Investments in Automation Are Necessary

Iterative development requires extensive **automation**.

Therefore, organizations should provide sufficient investment in automation and an integrated development environment.

Without proper automation, communication may return to paper-based methods and conventional process problems.

Such investments may be opposed by people who focus mainly on short-term financial results or individual project solutions.

11. Good Software Organizations Should Be More Profitable

In commercial software, profitability is important. In software contracting, especially government contracts, profit can become a complicated issue.

Contracting processes often try to keep contractor profits within an acceptable range, typically **5% to 15%**.

Excellent performance, good value engineering, or significant reuse may sometimes produce profits above the initial acceptable range.

When customers become aware of such additional profits, pressure may be applied to use those resources for out-of-scope changes.

This can replace the simple profit motive with complex contractual incentives and producer-consumer conflicts.

Contractors may have little incentive to implement major cost savings or take risks that could produce large returns.

They may also consume large amounts of money without producing sufficient results and with little accountability for poor performance.

For the software industry to prosper, **good contractors should be rewarded with more profit, while poor contractors should receive less profit.**

A customer receiving a good product at a reasonable price should accept that the contractor also earns a good profit.

Allowing poorly performing contractors to continue is not beneficial to the software industry.

The textbook notes that the **commercial domain is more effective than the government contracting domain** in this respect.

DENOUEMENT

Conventional Software Process

The conventional software process moves **sequentially from requirements to design, coding, and testing.**

It aims to achieve **100% completeness of each artifact** at every life-cycle stage.

Requirements, artifacts, components, and other work products are treated as **equals.**

It also maintains **high-fidelity traceability** among all artifacts at each stage of the life cycle.

Modern Iterative Development Process

A modern iterative development process supports **continuous round-trip engineering** from requirements to testing at evolving levels of abstraction.

It develops a **high-fidelity understanding of the important drivers (the 20%)** as early as practical.

Artifacts are evolved in **breadth and depth** according to **risk-management priorities**.

Completeness and consistency analysis are **postponed until later** in the life cycle.

Modern Process Framework

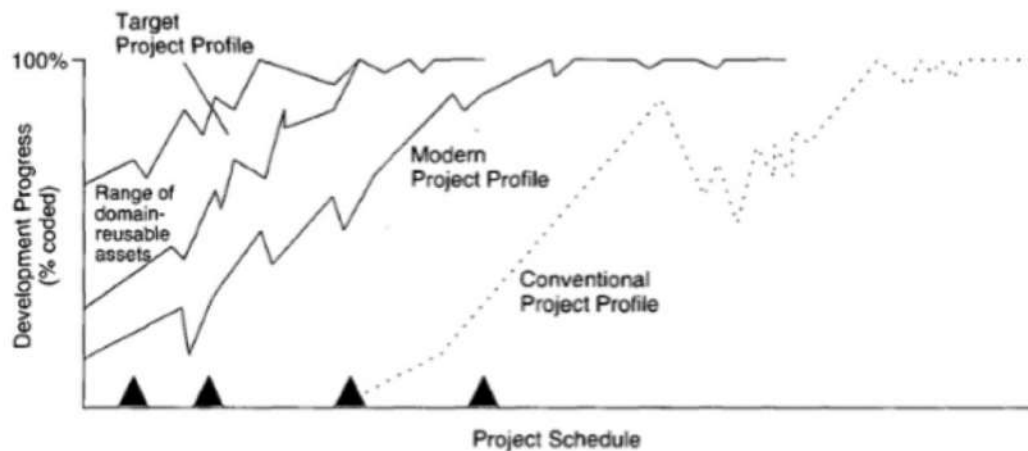


FIGURE 17-1. Next-generation project performance

A modern process framework addresses the major sources of **diseconomy of scale** found in the conventional software process.

Development progress is measured as the **percentage of code that can be demonstrated in its target environment**.

The modern project profile shows faster and more continuous progress compared with the conventional project profile.

A modern process using an **advanced integrated environment and component-based architecture** can improve software productivity.

Transition to New Techniques and Technologies

When an organization adopts new techniques and technologies, there is always concern about possible failure.

The organization may prefer to maintain the existing method because it appears safer.

Successful transition requires two important actions. New techniques should first be tried on a **small pilot program**.

The organization should also be prepared to spend **additional resources, money, and time** on the first project making the transition.

Organizational Paradigm Shift

Small pilot programs are useful, but they rarely produce a major **paradigm shift**.

A small experiment may show good initial results, but it may not reveal the **critical path of the organization**.

Therefore, such experiments may not receive enough management attention or resources.

Successful paradigm shifts are more likely when the organization selects a **critical project**, provides adequate resources, and demands improved results.

If the organization expects the transition project to fail, it may provide insufficient support. This can become a **self-fulfilling prophecy**.

Transition to a Mature Iterative Development Process

A better transition approach is to adopt a mature iterative development process supported by **automation technologies and modern architectures**.

The transition can be organized into three stages: **Ready, Aim, and Fire**.

Ready

The organization should study modern approaches and technologies.

It should define, improve, or optimize its process and support it with mature **environments, tools, and components**.

The project should also be planned thoroughly.

Aim

The organization should select a **critical project** for the transition.

The project should have the right combination of complementary resources and should demand improved results.

Fire

The organization should execute the **organizational and project-level plans with vigor and follow-through.**

This ensures that the planned transition is properly carried out.