

Chapter 9

Engineering Software-Intensive Systems

Contents

9.1 Engineering is the Key	9-3
9.2 What is Domain Engineering?	9-4
9.2.1 Domain Identification	9-5
9.2.2 Domain Analysis	9-6
9.2.3 Domain Design	9-6
9.2.4 Domain Implementation	9-7
9.2.5 Benefits of Domain Engineering	9-8
9.3 What is Systems Engineering?	9-9
9.3.1 Integrated Product Development (IPD)	9-11
9.3.2 Concurrent Engineering	9-13
9.3.3 The Case for Software Engineering	9-14
9.3.4 Domain Engineering and the Software Engineering Process	9-17
9.3.5 Relationship Among Enterprise Engineering, Domain Engineering, and Application Engineering	9-19
9.4 What is Software Engineering?	9-20
9.4.1 Software Engineering Goals	9-22
9.4.1.1 Functionality	9-22
9.4.1.2 Supportability	9-22
9.4.1.3 Reliability	9-22
9.4.1.4 Safety	9-23
9.4.1.5 Efficiency	9-23
9.4.1.6 Understandability	9-24
9.4.2 Software Engineering Principles	9-24
9.4.2.1 Abstraction and Information Hiding	9-25
9.4.2.2 Modularity and Localization	9-26
9.4.2.3 Uniformity, Completeness, and Confirmability	9-26
9.5 Managing Software Engineering	9-27
9.5.1 Software Engineering Information	9-28
9.6 What is Information Engineering?	9-29
9.6.1 Information Engineering Process	9-30
9.6.2 Information Engineering Architecture	9-31
9.6.3 IDEF	9-32
9.7 Success Through Engineering	9-34
9.8 References	9-36

9.1 Engineering is the Key

As we mature into the Information Age, the same forces that assured success during the Industrial Revolution are driving how we produce software. The world's industrial giants attained their status through superior mass production processes developed through advanced engineering. *Mass demand* and *global competition* are driving software production into the world of engineering, the same as they did for hardware.

Computer hardware engineering is quite mature and grew out of the manufacturing and electronic design processes. Within the hardware engineering discipline, “*hardware design techniques are well-established, manufacturing methods are continually improved, and reliability is a realistic expectation rather than a modest hope.*” Unfortunately, software has not advanced nor matured as quickly as the electronic hardware upon which it runs. In computer-based systems, where the hardware component is exceptionally stable with predictable fast-paced advances — the software is usually “*the system element that is most difficult to plan, least likely to succeed (on time and within cost), and most dangerous to manage.*” [PRESSMAN92]

New trends in development, however, are gradually removing the riskiest component stigma from software. By applying the engineering discipline that matured hardware beyond the risk threshold, software can now achieve expected levels of reliability, maintainability, and reusability. Software engineering is maturing software development, which has been historically characterized as a cottage industry populated by artisans, craftsmen, and skilled maverick developers. Engineering discipline is transforming software production into a mighty industrial machine characterized by a finely-tuned engineering process that predictably and consistently mass produces reliable software, on time, at competitive prices. The quickest, cheapest, highest quality way to build software is not to make mistakes during its development, and not to do any job more than once. Through years of experience and a well-defined, mature process, world-class software developers have learned how to do the right things, the right way — the first time, every time. Sound software engineering discipline is essential for software success. [ZELLS92]

“Above all, discipline; eternally and inevitably, discipline. Discipline is the screw, the nail, the cement, the glue, the nut, the bolt, the rivet that holds everything tight. Discipline is the wire, the connecting rod, the chain that coordinates. Discipline is the oil that makes machines run fast, the oil that makes parts slide smooth, as well as the oil that makes the metal bright. The principle of discipline here is divinely simple; you lay it on thick and fast, all the time.” — Private Gerald Kersh [KERSH90]

Software engineering discipline cannot be ignored; it must be laid on thick and fast — all the time. It must be institutionalized early in the life cycle. For large military software systems, this is particularly difficult because most requirements are based on subjective strategic and tactical demands, are dynamic, evolve over time, and are troublesome to precisely define. Developers and testers (software verifiers) must have procedures to identify and remove errors during requirements definition and design before they are translated into code. Quality can only be accomplished through the rigorous application of software engineering discipline and process knowledge to ensure that *quality is the norm — not the exception*. As you will learn throughout these Guidelines, successful program management strives for both process and product quality through methods to:

- Assess process and product status,
- Foster early process and product error identification and correction, and
- Continuously improve processes and product methodologies to prevent defects.

The implementation of a disciplined engineering process for software is a complex process. Software engineering for a system interacts with, and is dependent upon, related domain engineering, information engineering, hardware engineering, and systems engineering activities that occur in the production of a total, integrated system. Figure 9-1 illustrates (on a high level) the relationships among these engineering disciplines.

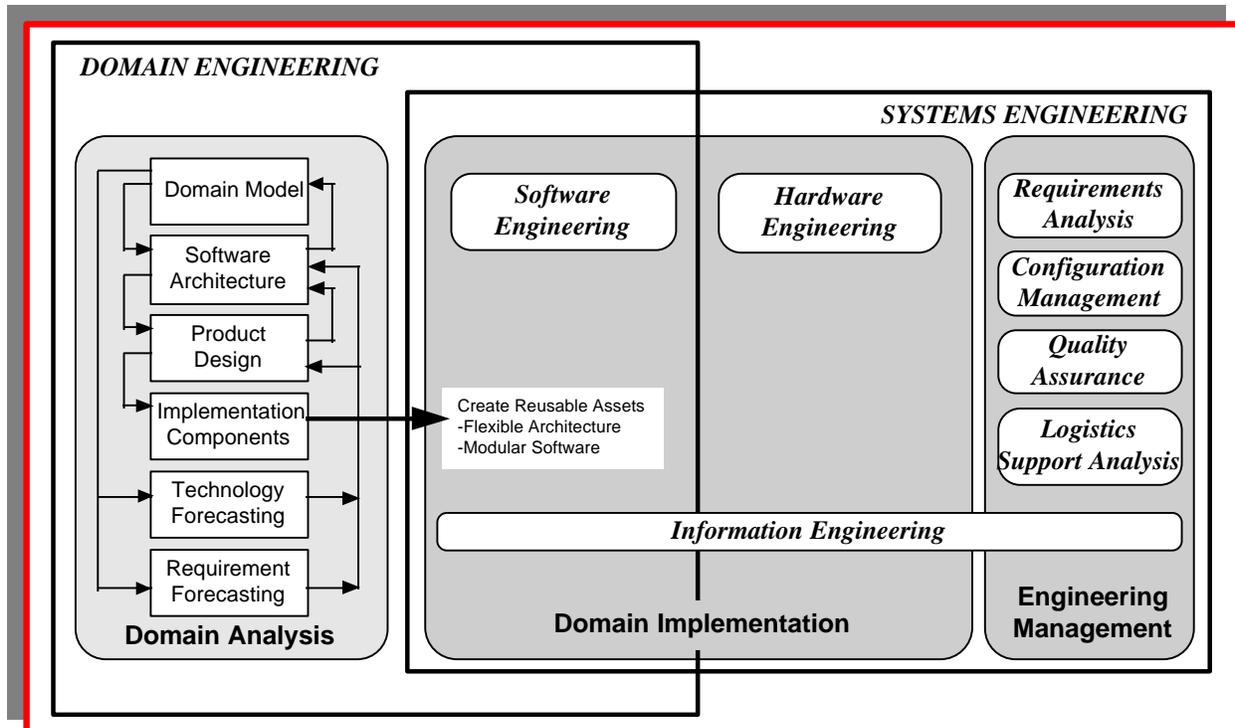


Figure 9-1. Total Quality Engineering

9.2 What is Domain Engineering?

Domain engineering refers to the techniques (i.e., methods and processes) used to engineer a family of similar or related systems (i.e., a domain or product-line). The focus of domain engineering is to capture engineering knowledge (requirements, architectures, components and other life cycle artifacts) within a particular domain for use on future or concurrent programs. This knowledge (captured by models, architecture specifications, etc.) is then used to configure a system architecture and develop (or select) reusable components based upon previous requirements analyses, design, coding, integration and testing efforts.

Domains are groups of related systems sharing a set of common capabilities. Domains can be described pictorially as having either vertical or horizontal relationships among each other, as illustrated in Figure 9-2. A vertical domain is a specialized class of system, such as an information system, command and control, or embedded weapon system. Horizontal domains consist of

general software functions applicable across multiple vertical domains. These can include user interfaces, common algorithms (e.g., data structures, strings, matrices, lists, stacks, queues, trees, graphs), common mathematical solutions (e.g., linear systems applications, integration, differential equations), and software tools or graphics packages. Although the domain engineering steps are presented here as sequential activities, in practice they are highly iterative. Major domain engineering steps include:

- Domain identification and scoping,
- Domain analysis,
- Domain design, and
- Domain implementation.

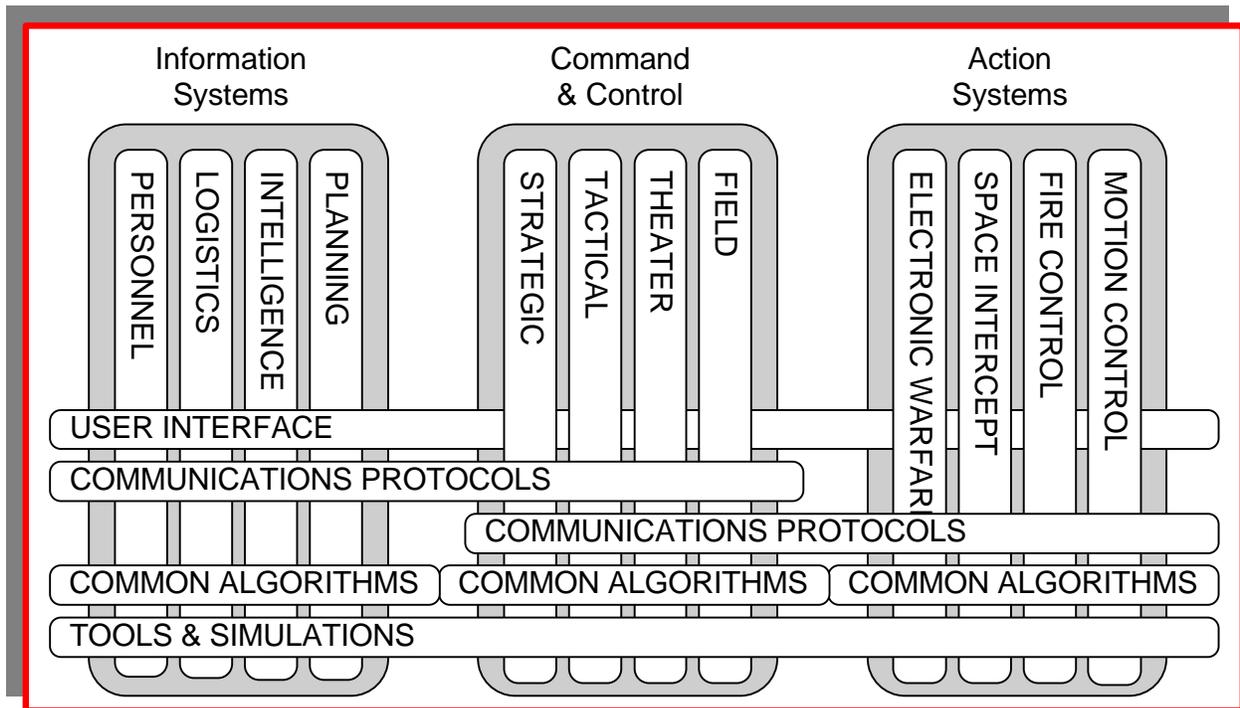


Figure 9-2. Vertical and Horizontal Domains

9.2.1 Domain Identification

The domain identification step is critical to overall program success. Your understanding of the subject domain and customer needs derived during this phase drive the entire engineering effort. Domain identification defines domain boundaries, interfaces, and dependencies. The knowledge gained during domain identification provides domain analysts with a common understanding of:

- Domain scope (inclusion or exclusion of domain applications),
- The relationship of the subject domain with other domains,
- The relationships among domain applications, and
- The inputs/outputs to and from the domain.

During domain identification, a number of domain-representative systems are identified. These “*exemplar*” systems highlight common, variable domain requirements. The number of exemplars must reflect the level of effort required to conduct the modeling effort with schedule constraints. Note that domain analysis addresses more than the exemplars — it also considers other information such as technology trends and anticipated future requirements.

9.2.2 Domain Analysis

Domain analysis captures and models requirements information across a particular domain. Domain analysis is the process of identifying, documenting, and modeling common, variable requirements among domain systems. Domain analysis techniques include interviews, documentation review, and reverse engineering, to identify and categorize (i.e., model) domain requirements. Other inputs, such as enterprise models (e.g., data models in IDEF1X and operational models in IDEF0), are used during domain analysis. The resulting domain model(s) form a domain problem space (or domain requirements) representation. These models provide a domain perspective of domain systems in terms of data (or objects), functional capabilities, and control or (behavioral aspects). Along with these perspectives, a standard domain vocabulary is developed. The products of domain analysis vary depending on the analysis method used and typically include the following:

- **Information model.** This provides the domain data (object) perspective. During this activity, domain data requirements, essential for implementing domain applications are represented. Variability among exemplars is represented in the information model through alternative objects and/or attributes. Information entities are traced back to the exemplar sources from which they are derived.
- **Feature or functional model.** This captures the end-user’s understanding of domain application capabilities through a functional domain systems perspective. Alternative feature commonality and variability among the different exemplar systems are represented in the domain model. Features are categorized and traced back to their exemplar sources.
- **Operational model.** This identifies domain application control and data flow commonalities and differences from a behavioral perspective. This activity abstracts and then structures common domain functions, features, and sequencing into an abstract operational model from which individual application control and data flow are derived.
- **Domain dictionary.** A useful product of domain analysis, this defines the terms and/or abbreviations used in describing domain features, their textual description, and domain entities. The dictionary helps alleviate miscommunication by providing a central location for domain information users to search for unfamiliar terms and abbreviations or for definitions of terms used differently or specific to the domain.

9.2.3 Domain Design

The domain-specific software architecture (DSSA) is the foundation of systematic reuse and the maturing of software engineering maturity. The DSSA provides the high-level design for all domain (or product-line) systems and establishes the context for high-leverage, large-scale reuse. The domain model, created in the domain analysis step, is used during domain design to derive the DSSA, which specifies a set of solutions to the requirements represented in the model. The

DSSA accommodates domain model requirements variability by capturing context drivers leading to alternative solutions. [KOGUT94] The DSSA identifies:

- **Component classes.** These are derived through partitioning overall system functionality, as captured in the domain model. A component class represents a category of components with similar functionality (e.g., DBMS or Geographic Information Systems). Each functional requirement captured in the domain model is allocated to one or more component classes. Component class variability reflects the variability captured in the domain model by specifying alternative and optional classes.
- **Connections.** These describe how component classes are linked. Typical connections specifications include data flow, direction, and type (e.g., SQL query, protocol). Alternative connections result from variability in domain requirements.
- **Constraints.** These describe component class characteristics allocated from the domain model and implied by the architecture. That is, the constraints highlight functionality derived from the domain model, as well as the functionality dictated by component class connections. Because of the variability captured in the domain model, it is necessary to specify alternative and optional component class constraints.
- **Rationale.** This facilitates selection among reusable components. For example, suppose a more expensive DBMS will provide a faster response. This provides the rationale for choosing the more expensive DBMS when response time is critical.

Key to domain design is maintaining traceability between the derived architectural solution and domain modeled requirements. Domain designers can use this traceably to develop an initial systems architecture and to select or build a set of reusable components that best fit the new system's requirements. This allows selection of specific architectural solutions based on user/developer selection of specific domain requirements. This forms the basis for qualified components composition or new component specification and development based on constraints specified in the architecture.

9.2.4 Domain Implementation

Domain implementation refers to: (1) the process of creating new components or modifying existing components for a DSSA component class; and (2) altering components in response to changes in requirements or the detection of defects. These domain assets can be employed or modified to suit new systems development within the domain. Domain implementation can also include the development of automated tools that aid life cycle efforts, such as composition tools, generators, and analyzers. [MAYMIR95]

There are four main strategies for domain implementation: generation; new development; re-engineering; and identification of commercial-off-the-shelf (COTS) and government-off-the-shelf (GOTS) software. In addition, for re-engineered software, COTS, and GOTS, there is a separate domain-specific step for software qualification. Some combination of these strategies is employed to complete domain implementation. The cost and applicability of these strategies depends on tool support (especially for generation), the level of domain maturity (for re-engineering), and the availability of COTS or GOTS software. These approaches must be analyzed to determine an appropriate domain implementation strategy. Specific strategies are then developed to fit the needs of the customer. A brief discussion of these strategies follows.

- **Generation.** If a component class is mature and well-defined, generative techniques may be applied to automatically produce reusable assets that fit the generic architecture. Commercial generators are available to support this strategy in several narrowly focused areas. An example is the graphical user interface (GUI) builder. For component classes where commercial tools are not available, it may be necessary to build special-purpose *application generators* tailored to the class.
- **New development.** Reusable assets can be developed, as part of a normal software engineering effort, to satisfy generic architecture requirements. This strategy, frequently employed by domain engineering teams, is suitable when legacy software is nonexistent or not suitable for reuse.
- **Re-engineering.** Legacy systems (e.g., exemplars used during domain analysis) may include components *close enough* to the desired functionality and structure to warrant redesign and/or re-implementation to be reusable within the domain.
- **Qualification.** Many domain engineering teams are using available software to satisfy DSSA. Pre-existing software must be evaluated against DSSA requirements. Component qualification assesses how well a particular component fits into a DSSA component class and ensures components are reusable within a given architectural context. Qualification criteria for evaluating reusable components are dependent on domain characteristics and user needs.

If new software is chosen as the domain implementation strategy, DSSA requirements are used in creating new reusable software assets. Several efforts have focused on developing generic guidelines for creating reusable software (hence, the name “*design-for-reuse*”). However, guidelines differ based on development methodology (object-oriented, functional decomposition, etc.). An appropriate software development methodology must be used to create reusable software assets. Domain engineering provides necessary background for tailoring *design-for-reuse* guidelines for a selected development methodology.

9.2.5 Benefits of Domain Engineering

In September 1991, the Air Force and the Advanced Research Projects Agency (ARPA) selected the Air Force Space Command’s Space Command and Control Architectural Infrastructure (SCAI) program as the Air Force Demonstration Project for Software Technology for Adaptable and Reliable Systems (STARS) megaprogramming concepts. Demonstration programs were also awarded to the Army and Navy.

The demonstration program’s goals were to show the feasibility of using an architecture-based, product-line approach to system development. The SCAI program realized benefits in the areas of productivity, error reduction, and cost savings. Productivity went from 175 lines-of-code (LOC) per month to over 1,700 LOC per month. Defects decreased from 3+ errors per 1,000 LOC to about 0.35 errors per 1,000 LOC. Cost per 1,000 LOC also decreased from the typical \$140+ to about \$57. These benefits were realized because of the domain engineering approach.

For additional information on domain engineering, or assistance in selecting and implementing appropriate domain engineering methods, please contact one of the following organizations [*see Volume 2, Appendix A for addresses, phone numbers, and Web addresses*]:

- AF/Comprehensive Approach to Reusable Defense Software (CARDS) Program, and
- Software Engineering Institute (SEI).

9.3 What is Systems Engineering?

The first formalization of systems engineering for military development occurred in the mid-1950s on ballistic missile programs. [DSMC90] Since then, the systems engineering discipline has evolved to encompass both technical and management processes, and has expanded its applicability to cover the entire life cycle of a software-intensive system. A technically-oriented definition of systems engineering is:

“...an interdisciplinary approach encompassing the entire technical effort to evolve and verify an integrated and life cycle balanced set of systems people, product, and process solutions that satisfy customer needs.” [EIA632]

Army Field Manual 770-78, *Systems Engineering* (1979), provides a definition, not specific to any particular industry segment, that emphasizes the *leadership role* systems engineering plays in integrating other disciplines. It defines systems engineering as:

“The selective application of scientific and engineering efforts to:

- *Transform an operational need into a description of the system configuration which best satisfies the operational need according to the measures of effectiveness;*
- *Integrate related technical parameters and ensure compatibility of all physical, functional, and technical program interfaces in a manner which optimizes the total system definition and design; and*
- *Integrate the efforts of all engineering disciplines and specialties into the total engineering effort.”*

Whichever definition you prefer, both have the same goal — to effectively balance system elements by integrating them into a complete system that meets customer needs. Systems engineering is not a one time or single phase effort. It is an essential activity throughout the system’s life. During the early planning phase it assures flexibility and supportability are built into the design. In later years, it aids in smooth, effective change implementation and modification, often adding value and prolonging the system’s life, as illustrated in Figure 9-3. [EIA632]

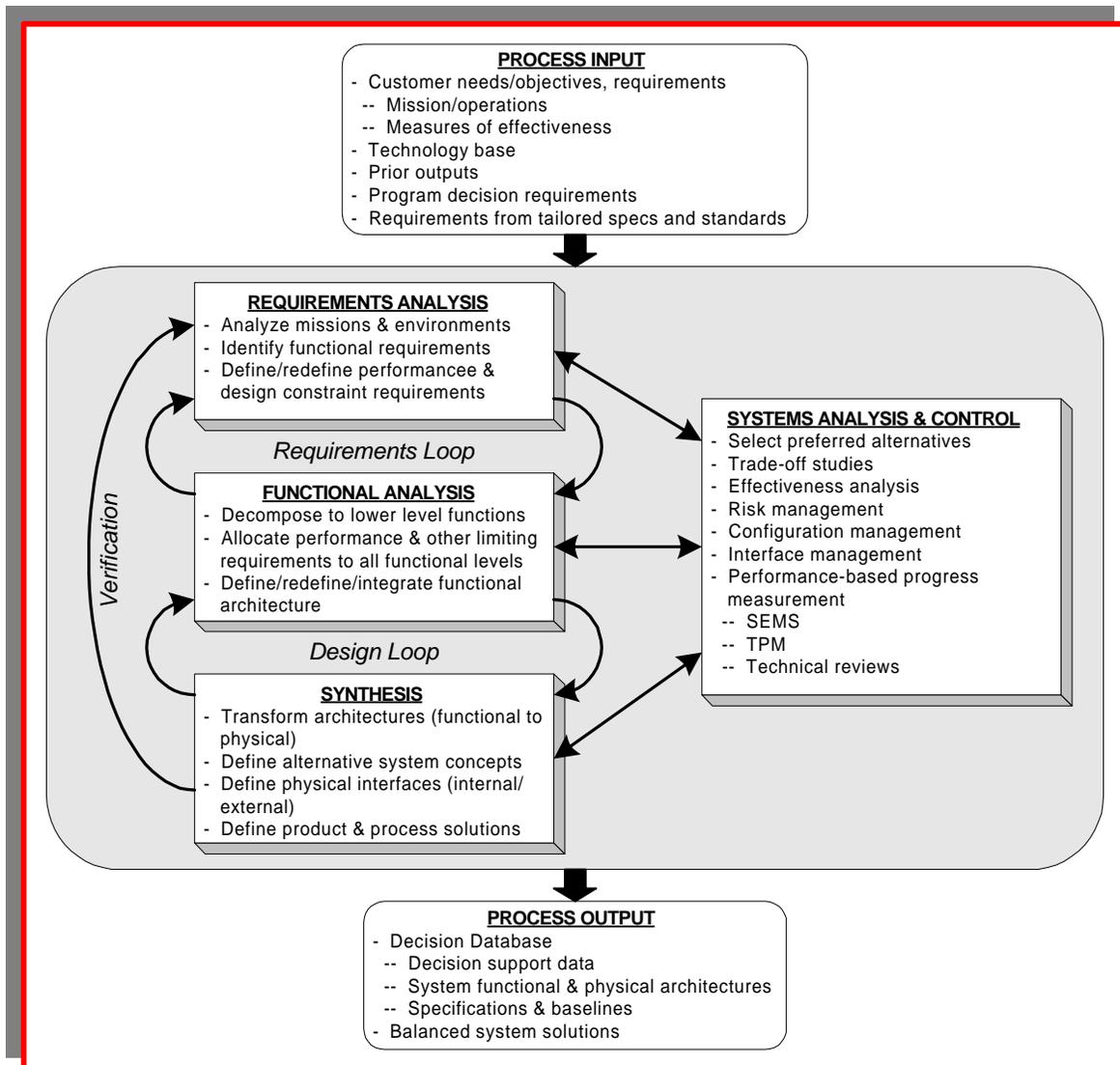


Figure 9-3. Systems Engineering Process [EIA632]

Applied iteratively throughout the system life cycle, the systems engineering process has four interrelated process steps the Electronics Industry Association (EIA) Standard 632 defines as:

1. Requirements Analysis,
2. Functional Analysis,
3. Synthesis, and
4. Systems Analysis and Control.

The exact activities of the systems engineering process should be documented in a Systems Engineering Management Plan (SEMP), while progress towards completion of these activities should be identified and tracked using technical performance measurements (TPMs).

NOTE: You are urged to obtain a copy of EIA Standard 632 or IEEE 1220 and follow the guidance found there.

Of a system's components (i.e., people, products, and processes), two of the more notable products are hardware and software. Systems engineering takes both into account, giving each equal weight in analysis, tradeoffs, and engineering methodology. In the past, the software portion was viewed as a subsidiary, follow-on activity. The new focus in systems engineering is to treat both software and hardware concurrently in an integrated manner. At the point in system design where the hardware and software components are addressed separately, modern engineering concepts and practices are employed for software, the same as they are for hardware. [MOSEMANN92]

Figure 9-4 illustrates how systems engineering, hardware engineering, and software engineering are concurrent processes. *The primary role of systems engineering is to ensure that the many diverse elements comprising a system are compatible and ready when needed.* This avoids the situation in which the hardware or software, when integrated into the system, fails to function harmoniously with other system components. Systems engineering concentrates on comprehensive planning and coordination throughout the development process to ensure integration problems are minimized and that final system implementation fulfills all mission requirements. Different approaches have evolved to implement the systems engineering process. One approach used by DoD is integrated product development (IPD) that focuses on the abatement of integration issues.

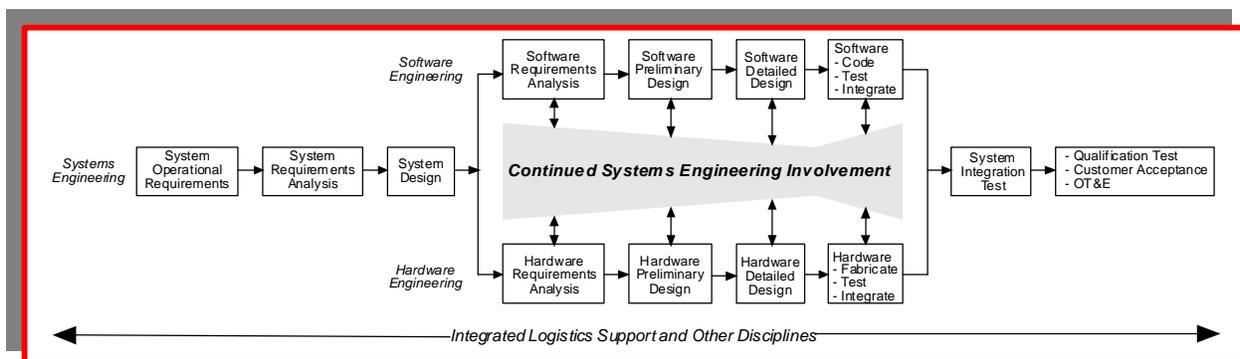


Figure 9-4. Relationship between Systems, Hardware, and Software Engineering

9.3.1 Integrated Product Development (IPD)

Integrated product development is “a team approach to systematically integrate and concurrently apply all necessary disciplines throughout the system life cycle to produce an effective and efficient product or process that satisfies customer needs.” [WAGNER95] The key ingredient in IPD is *teamwork*. IPD provides a technical-management framework for a multi-disciplinary team (comprised of multiple specialties) to define the product. The team includes users (both operational and support) to better address their needs and ensure developers consider all aspects of the system life cycle. IPD emphasizes up-front requirements definition, tradeoff studies, and the establishment of a change control process for use throughout the entire life cycle. This life cycle emphasis is why, according to Captains Gary Warner (USAF) and Randall White (USAF), the F-22 program refers to their IPD teams as *integrated product teams* (IPTs). The term “*development*” is omitted because the IPT continues into the operation and support phase by handling modifications and systems upgrades. [WAGNER95] An example of a multi-disciplined IPD team is illustrated in Figure 9-5.

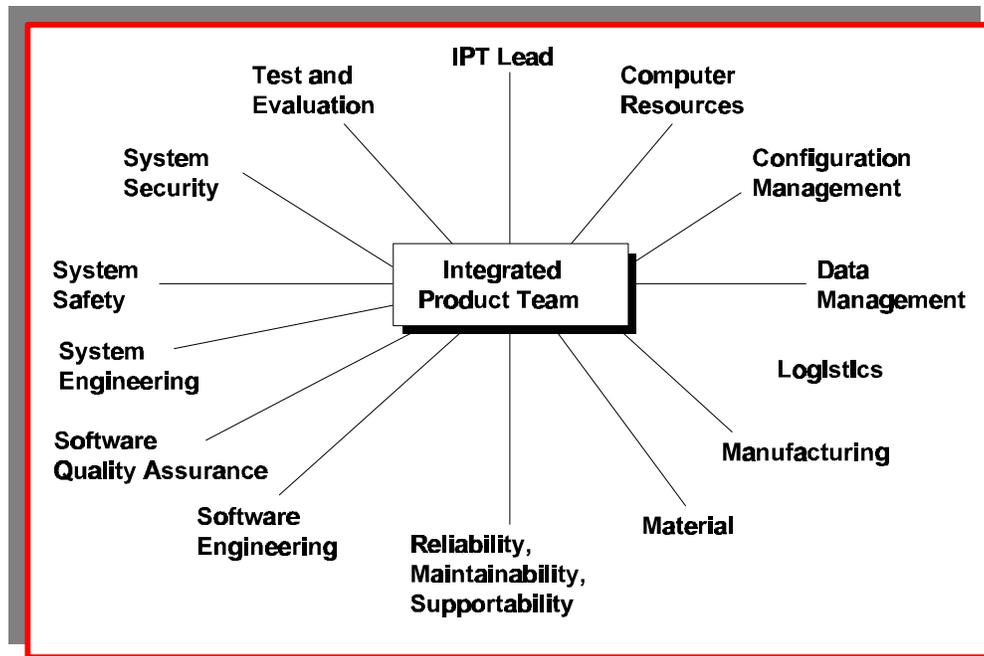


Figure 9-5. Example Integrated Product Team Members

Figure 9-6 illustrates the concept of IPD during the systems design phase. This figure is mainly conceptual, as several iterations through each filter step are often required. Four integration filters are shown in the overall process. As information is taken into the *traditional discipline filter*, emphasis is placed on traditional design techniques (such as structural stress analysis) required at any given design stage. Traditional design engineers rely heavily on current technology. At the same time, design documentation is developed and/or modified by *engineering specialists* who establish requirements independent of the emerging traditional design. They also review and modify the traditional design output. All requirements are then filtered by the unique demands of system products. Subsequently, requirements are described by specifications and drawings (or in some cases, prototypes) filtered through the user group to determine whether they satisfy needs.

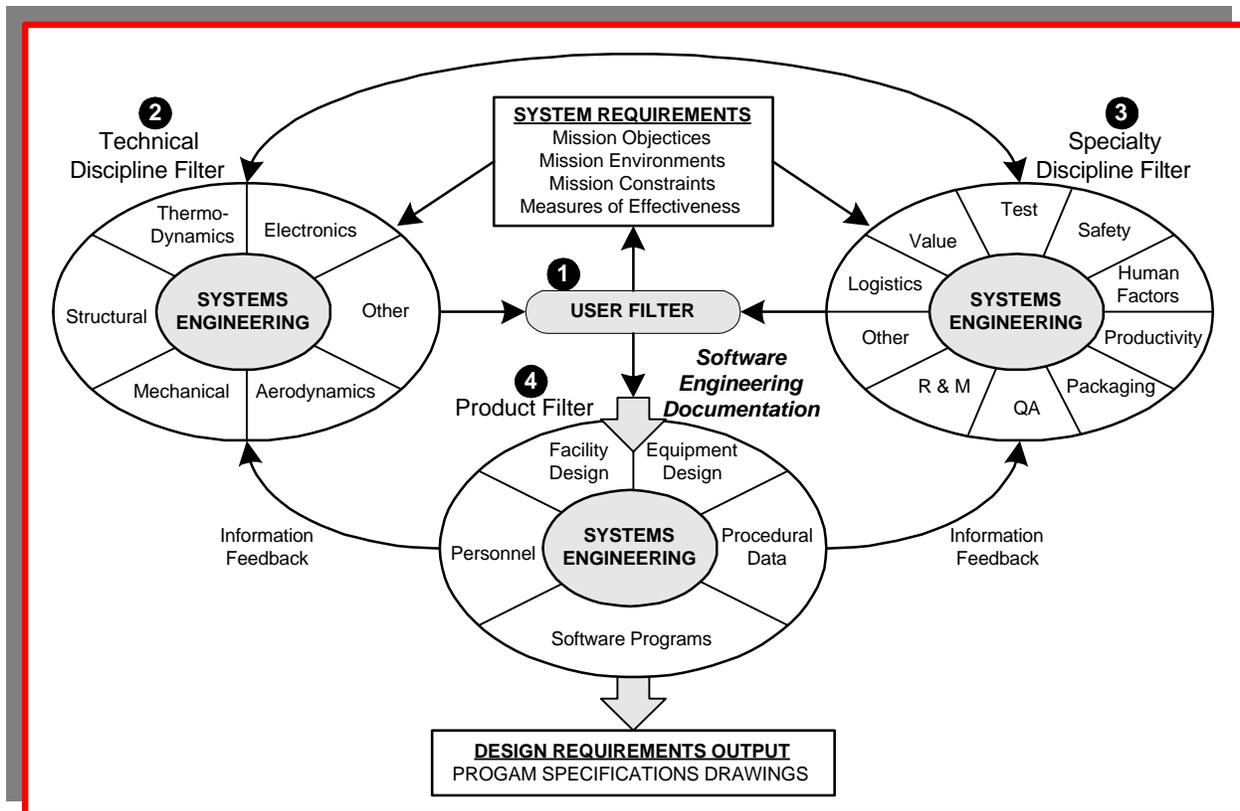


Figure 9-6. Integrated Product Development Process

9.3.2 Concurrent Engineering

Concurrent engineering [*not to be confused with concurrent acquisition*] is one of several systems engineering disciplines within the IPD approach used to define requirements and manage system acquisition and development. Concurrent engineering is “*a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support.*” [WAGNER95] Concurrent engineering, in this context, is the coordination, integration, and sequencing of the multi-discipline engineering activities that must occur to produce a major software-intensive system. The following summarizes concurrent engineering benefits achieved in three industrial applications:

“The Boeing Commercial Airplane Group is using it [concurrent engineering] to develop the giant 777 transport and expects to release design drawings a year and a half earlier than with the 767. John Deere & Co. used it to cut 30% off the cost of developing new construction equipment and 60% off development time. AT&T Co. adopted it and halved the time needed to make ... an electronic switching system.” — Sammy Shina [SHINA91]

Concurrent engineering ensures that people from many disciplines collaborate throughout the life of a product (*from-cradle-to-grave*) to ensure it performs to user’s needs and requirements. For example, people from engineering, software, operations, maintenance, and manufacturing work as a team from program onset to anticipate problems and bottlenecks and remove them

early. This approach avoids delays in fielding the product and prevents costly operational failures. Participation by contracts and logistics personnel also ensures a smooth acquisition process, low product cost, and availability of reliable supplies of parts and materials.

Automated design tools, computer-aided manufacturing systems, and information management tools are commercially available for concurrent engineering applications. A centralized graphical representation can help the team visualize key elements and relationships to achieve a multi-disciplinary design solution. Automated products enable the storing of supporting data so the results of historical design efforts can be applied to the task at hand. This hastens product improvement efforts beyond many life cycle iterations. [SHINA91]

An example of concurrent engineering in practice is the F-22 program. Colonel Robert Lyons, Jr., former co-leader of the F-22 System Program Office Avionics Group, explained that IPD is being employed throughout the F-22 program office, where he says they are using an expanded version of IPD with concurrent engineering. The program office is uniting program managers, as well as specialists in contracts, cost analysis, test, safety, logistics, and quality assurance to oversee product development. Lyons says that, “*Already in this program we [Government and industry] have laid on the table information that in other programs people wouldn’t have heard about for several years.*” He explains that the beauty of including everyone affected by the development in areas other than their own is that all program concerns and requirements are identified and addressed up-front. [LYONS92] Everyone, in this case, also includes the F-22’s customer, Air Combat Command (ACC). According to Captains Wagner and White, ACC is active in the F-22 Weapon System concurrent engineering effort, having local representatives who are “*active team members and provide on-the-spot inputs for requirement issues.*” This inclusion of the customer into the concurrent engineering team “*kept the user in the loop and provided a quick way of obtaining guidance on requirements.*” [WAGNER95] A recent assessment of IPD on the F-22 identified several key factors needed for successful implementation:

- Implement IPD from the beginning of the program with extensive planning to make it happen;
- Train and educate the team members on IPD, including new personnel coming in mid-program;
- Enhance communications with co-location of team members and use of electronic mail;
- Structure the system program office (SPO) to reflect the IPD system breakdown;
- Have the necessary integrated management tools to do the job [these include technical performance measurements, integrated master plans (or systems engineering management plans), and integrated master schedules (or systems engineering management schedules)]; and
- Establish an analysis and integration (A&I) team to integrate IPD team efforts to ensure “I” stands for *integrated* and not *independent*. [WAGNER95]

9.3.3 The Case for Software Engineering

The forces driving DoD towards *engineering* our software are primarily economic. Lloyd K. Mosemann, II, former Deputy Assistant Secretary of the Air Force (Communications, Computers, and Support Systems) explains that, “*military software must be engineered. There is too much of it and systems are too large to develop cost-effectively using the hand-tooled, cost-insensitive ‘software-as-art’ model.*” [MOSEMANN92²] He further states that, “*The definition and institutionalization of software engineering in the Air Force is now our highest priority.*”

[MOSEMANN91] From the DoD perspective, Paul Strassmann, former Director of Defense Information (ASD/C3I), reinforced the case for software engineering by declaring, “*The No. 1 priority of DoD, as I see it, is to convert its software technology capability from a cottage industry into a modern industrial method of production.*” [STRASSMANN91]

To understand what we mean by *software engineering*, the Software Engineering Institute (SEI) examined the mechanical and civil engineering disciplines which evolved from *ad hoc* solutions to *engineered* ones based on scientific principle. By scientific principle we mean:

“...an attempt to explain a certain class of phenomena by deducing them as necessary consequences of other phenomena regarded as more primitive and not in need of explanation”. [McGRAW89]

As practitioners within a discipline accept new explanations, the discipline shows a progression from crafted, *ad hoc* solutions to a formal, codified body of knowledge. Scientific principles, in the form of proven mathematical statements, are developed to explain and predict results. Initial solutions establish the foundations for creating new instances predicated on scientific principles. New and larger problems can then be addressed based on initial solutions. During this evolution, the state-of-the-practice constantly improves. Software engineering involves improving the practice through the codification of collective knowledge and experience. [HOLIBAUGH92]

In contrast to engineered solutions, crafted solutions are unique and problem-specific and the experience base is usually limited to that of the practitioner. Eileen Quann, president of Fastrak Training, Inc., equates the differences between software-as-art (or craft) and an engineered product to the different approaches required when building a dog house, a family home, and a skyscraper, as illustrated in Figure 9-7. In each case, building construction generally consists of assembly functions. For all three structures, a floor, walls, a roof, a door, and windows must be built.

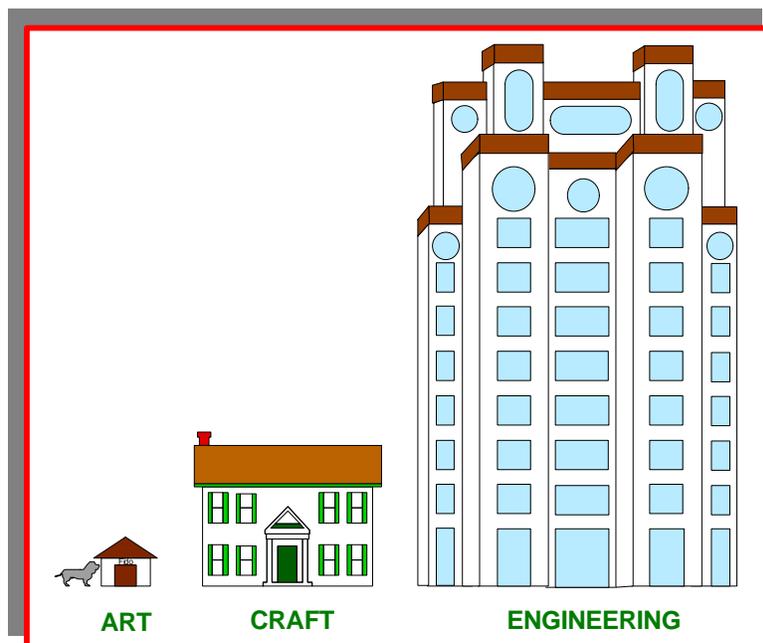


Figure 9-7. Order of Magnitude Between Software Engineering and Software-as-Art

The differences among the construction projects are found in the skills and tools needed to accomplish each job. Your teenage son can build a doghouse with a few nails and some wood. However, he is not qualified to build your family home which requires craftsmen skilled in reading blueprints, plumbing, wiring, roofing, flooring, insulation, and inspections. Similarly, the same craftsmen are not usually qualified to build a skyscraper, which requires additional skills in such areas as joining steel beams, installing tremendous amounts of glass and concrete that must withstand enormous physical stresses, and electrical wiring with demands much greater than a normal house. More importantly, while your teenage son may be able to both design and construct the doghouse, *when a program reaches the size of something like a skyscraper, design engineers must have more experience and knowledge than is required of a normal engineer/craftsman.* To design a large-scale building containing immense walls of glass, steel beams, and concrete, a design engineer must be an educated professional, proficient in topics such as the physics of structural stress. They must also have expertise in additional areas such as elevator dynamics, optimum space utilization, environmental power plants, and emergency and handicapped access requirements. Therefore, *it is just as important that your engineering design team be appropriately trained and experienced as it is that your construction personnel have the right skill and experience level.*

Software engineering is also required for economic reasons. Consider the fact that *the cost to support deployed DoD software system comprises 60% to 80% of total life cycle costs.* [Ada/C++91] The high cost of software support stems from products often so unique and hand-crafted no one other than the original developer can understand them. Supporting agencies have had to start from scratch when upgrading or enhancing the software they are responsible to maintain. Defects, not discovered until the software is deployed, are at times impossible to correct. Strassmann warns these practices are no longer acceptable, *“Because we don’t have the cash anymore to reinvent and reinvent and reinvent exactly the same routine.”* [STRASSMANN92] Figure 9-8 illustrates how software costs have historically been disbursed when software was developed as art. It also shows the difference in spending ratios when software is engineered. What Figure 9-8 does not show, however, is that the cost pie shrinks when software is engineered because software support costs are substantially reduced.

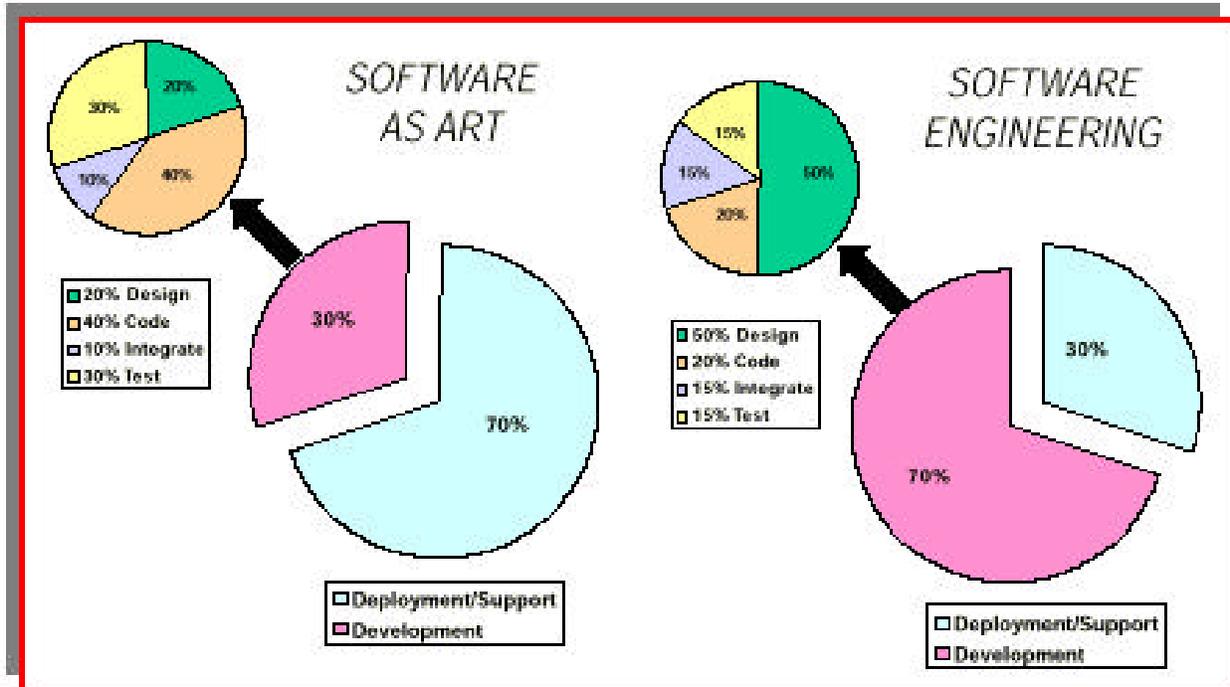


Figure 9-8. Software Life Cycle Costs

When you use the structured discipline imposed by the engineering process, costly software support problems are addressed up-front. Reliability, maintainability, and supportability are designed *into* the system instead of being included after development and deployment. Resources are planned and managed within a total life cycle framework. Large investments are placed in up-front design and engineering to gain savings over the life of the system.

To state that engineering is a solution to our software problems is no revelation. Simply put, systems and software engineering provide sound, proven discipline for achieving program success. As An Wang, founder of Wang Laboratories, aptly stated, “*Success is more a function of consistent common sense than it is of genius.*” [WANG86] By converting software production from a cottage industry into a modern industrial process, the same benefits can be attained as those gained through the engineering and mass production of hardware:

- Lower unit price,
- Lower maintenance costs,
- Reusable and interchangeable parts, and
- Greater reliability.

9.3.4 Domain Engineering and the Software Engineering Process

Mature engineering disciplines support clear separation of routine problem solving from the research and development required to address unprecedented aspects of systems within a well-defined product-line. Fundamental to such a discipline is the leveraging of a publicly-held, experience-based, and formally transitioned technology base that includes product models (e.g., designs, specifications, performance ranges) and practice models (tools and techniques to apply

the product models). A critical characteristic of mature engineering is that the products built from these models are well-understood and predictable before they are produced. Software engineering state-of-the-practice has yet to reach this level of maturity. Instead of basing new development on a technology base of well-understood models, current software engineering practice tends to start each new application development from scratch with the specification of requirements, and moves directly into design and implementation. By contrast, disciplined software engineering relies on a stable technology base of reusable assets, including requirements, designs, architecture, and software.

Figure 9-9 illustrates the role of domain engineering in establishing a mature, disciplined software engineering process and a product-line development strategy. The stable technology base, specific to the product-line, called the product-line asset base, is created and maintained by domain engineering, which while distinct and separate from the application engineering activity, defines, drives, and constrains application engineering. Domain engineering analyzes, selects, and produces the assets to populate the product-line asset base, which captures the commonality and variability across an entire product-line and includes models that facilitate understanding and specialization to a particular system. The application engineering process then uses these products and processes to develop software systems within the product-line. The application engineer draws upon these assets to develop reuse-based products (i.e., software systems). By using well-understood requirements, architecture models, well-documented processes, and high-quality reusable software, the engineer is able to quickly and cost-effectively build more reliable and predictable software systems for the product-line.

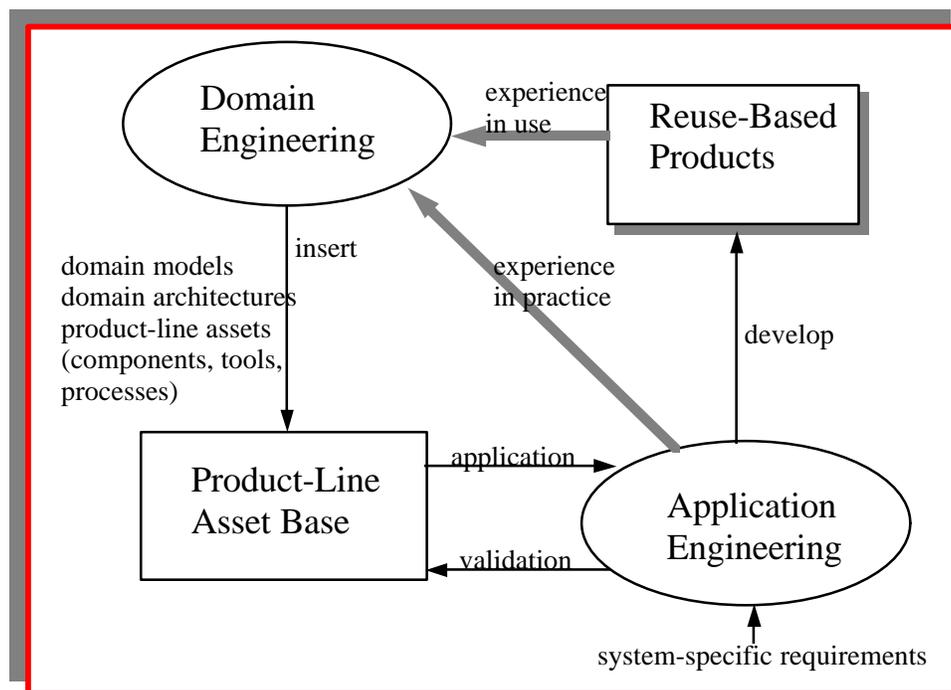


Figure 9-9. Domain Engineering and Software Engineering Discipline

The separation of domain engineering from application engineering highlights the need and significance of developing reusable *corporate assets* including domain models, architectures, processes, and components. The application engineering function then focuses on using, validating, and extending this technology base, instead of beginning with a blank sheet. In

addition to creating the initial set of domain assets, domain engineering processes continue to add and enhance the technology base according to the requirements associated with application engineering.

9.3.5 Relationship Among Enterprise Engineering, Domain Engineering, and Application Engineering

There has been a service-wide effort to develop organizational enterprise models, which combined comprise the DoD Enterprise Model. While the synergy between domain and application engineering has become better understood, the connection to enterprise engineering has remained weak. Domain engineering represents an intermediate level of abstraction between knowledge captured at the enterprise level and the array of systems developed at the application engineering level. Domain engineering reduces the complexity (and hence the risks) of leveraging enterprise-wide common data and functions in the development of individual applications using classical *divide-and-conquer* techniques. Figure 9-10 illustrates the basic methods (at a very high level) associated with the three major software engineering processes: enterprise engineering, domain engineering, and application engineering. Each of these processes attacks the problem space, the solution space, and the implementation space at different levels of abstraction. Application engineering is concerned with a single system/application, whereas domain engineering takes into account multiple similar or related systems; and enterprise engineering looks at an entire enterprise's (organization's) high-level data and operational needs.

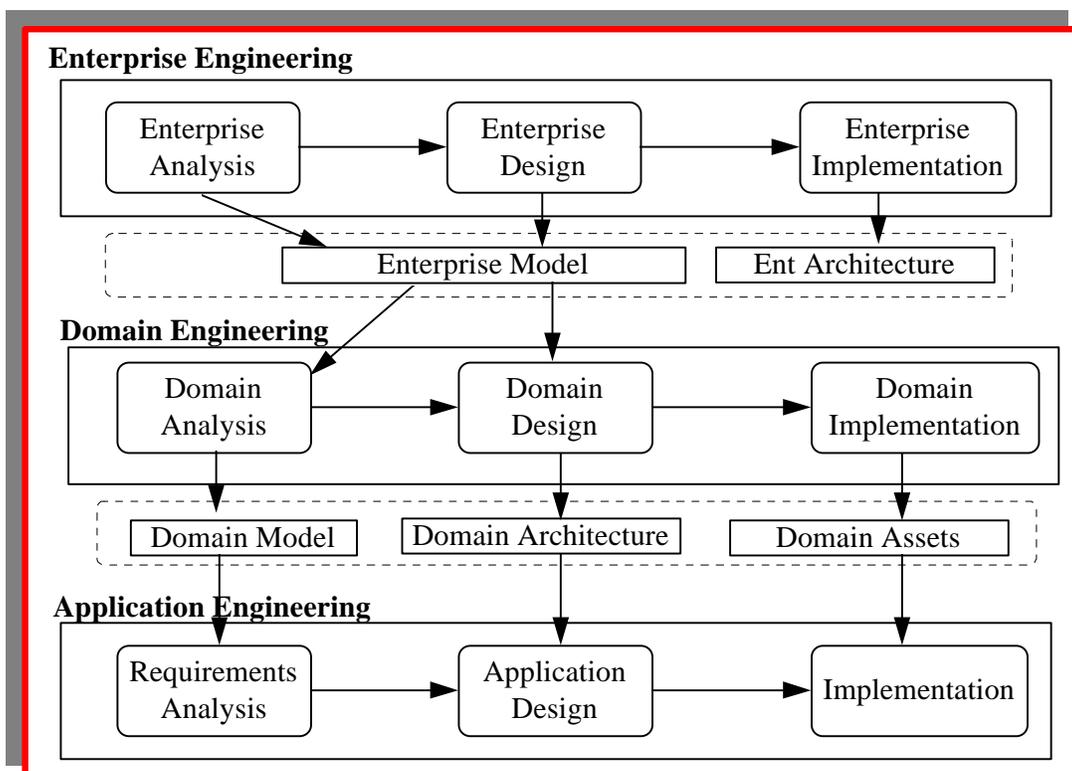


Figure 9-10. Three-tiered View of Organizational Engineering Processes

Viewed top-down, the enterprise for an individual organization can be broken down into multiple domains, which in turn can be broken down into multiple applications. Managing and engineering software from these three different levels helps mitigate risks. It also ensures that information and insight developed at higher-levels of abstraction are leveraged at lower levels.

9.4 What is Software Engineering?

Mosemann defines software engineering in a meaningful context for software managers when he explains,

“By software engineering, I mean simply the application to software of the traditional engineering process, which encompasses the following kinds of activities:

- *Iteration between formal analysis and design,*
- *Heavy use of earlier designs,*
- *Tradeoffs between alternatives,*
- *Handbooks and manuals,*
- *A pragmatic approach to cost-effectiveness, and*
- *Attention to economic concerns.”* — Lloyd K. Mosemann, III [MOSEMANN92³]

DoD 5000.2-R, *Mandatory Procedures for Major Defense Acquisition Programs (MDAPs) and Major Automated Information System (MAIS) Acquisition Programs*, applies the traditional engineering process to software. It states that,

“Software shall be managed and engineered using best processes and practices that are known to reduce cost, schedule, and technical risks. It is DoD policy to design and develop software systems based on systems engineering principles...”

The principles include:

- Use of open system concepts;
- Exploiting software reuse opportunities;
- Use of appropriate programming languages in government-supported applications;
- Use of DoD standard data;
- Selecting contractors with domain experience, a successful past performance, and a demonstrable mature software development capability and process; and
- Use of software metrics.

Additionally, software engineering structures the complexity of software development with a defined set of techniques and methods to measure and control the process. [ZRAKET92] Pressman identifies methods, tools, and procedures as the basic elements necessary to ensure a quality product:

“[Software engineering is]...an outgrowth of hardware and systems engineering. It encompasses a set of three key elements—methods, tools, and procedures—that enable the manager to control the process of software development and provide the practitioner with a foundation for building high-quality software in a productive manner.” — Roger S. Pressman [PRESSMAN92]

- **Software engineering methods** define the technical *how-to's* for software development. Methods cover a range of tasks that include:
 - Program planning and estimation,
 - System and software requirements analysis,
 - Architecture design,
 - Algorithm procedure and data structure development, and
 - Coding, testing.
- **Software engineering tools** give automated (or semi-automated) support to the methods. A variety of tools support each of the methods listed above. Computer-aided software engineering (CASE) is an integration of different tools where information created by one tool can be used by other tools. CASE combines software, hardware, and a software engineering database of information about analysis, design, code, testing, and metrics.
- **Software engineering procedures** merge the methods and tools for rational, timely software development. Procedures establish the order in which the methods are applied, deliverables (reports, documents, capabilities, functions) are required, and controls (ensuring quality and coordinating change) are enacted. They also define the milestones needed to evaluate software development progress.
- **Software engineering discipline** consists of defined steps combining the methods, tools, and procedures forming the basis for process improvement activities. These steps are often referred to as software engineering paradigms (or models). Paradigms must be selected based on your type of program and application, the methods and tools used, and the constraints and deliverables required. [PRESSMAN92] Figure 9-11 summarizes the components of software engineering.

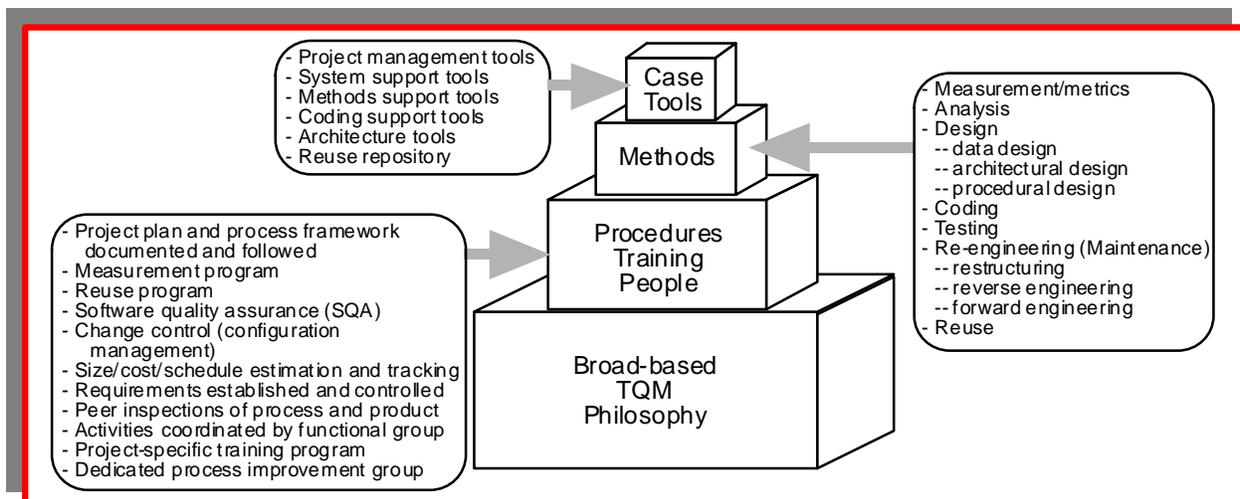


Figure 9-11. Software Engineering Elements

As with systems engineering, numerous approaches have evolved for implementing the software engineering process. For weapon systems, various approaches are integrated within the systems engineering process. The IPD example is one such approach. For Management Information Systems (MIS), information engineering (IE) and Integrated Computer-Aided Manufacturing Definition Language (IDEF) are approaches that implement the software engineering process. What brings the methods, tools, and procedures of software engineering together are its goals and a set of principles that must be accomplished to achieve engineered software. The defined

set of principles, transcending all engineering efforts, support and implement the key goals. They also help with the complexity of managing the development of a major software-intensive system.

9.4.1 Software Engineering Goals

Remember, the primary goal for all software engineering efforts is for the software solution to meet user needs by fulfilling stated requirements. However, in many large, complex software-intensive systems, requirements often evolve over the life of the system with the greatest costs incurred during the maintenance phase. Given that *change is a constant* in the life cycle, the discipline of software engineering is founded on six main goals. The main software engineering goals are *functionality, supportability, reliability, safety, efficiency, and understandability.*

9.4.1.1 Functionality

If the software doesn't provide a solution to user needs, it has no purpose. Functionality of the software is its primary goal. If this is not met, all other goals are moot. If the functionality goal of a car is transportation, then the car must be able to take you where you want to go before other goals, like color, fuel efficiency, comfort, etc. are worth considering. Because we tend to specialize, we sometimes put too much emphasis on specialty goals, while forgetting to overall purpose of the project.

9.4.1.2 Supportability

Supportability [*discussed in detail in Chapter 12, Software Support*] is the ability to perform maintenance and enhance, upgrade, or otherwise change the software. Component aspects of supportability include maintainability, adaptability, and modifiability. Weapon system software requires a high degree of supportability, as it must be changed to keep pace with evolving threats. Both weapon system and MIS software must also be regularly altered to keep up with the evolution of user, operational, and support requirements. These environmental and evolutionary factors result in *controlled software changes*. Defect correction is viewed as a controlled software change.

To effectively support a software system, all the *explicit* and *implied* design decisions comprising the solution must be honored. This requires that the design rationale be captured in a manner that software support personnel can use during the system's normal 10 to 30 year operational life of the system. If this information is not captured and considered, new software will end up being patched into the original code by breaking apart the logical basis of the design. Also, if the software is initially poorly designed and constructed, after several block updates the original structure will tend to deteriorate and get lost, complicating future software support efforts. *Well-engineered software systems are easily supportable and can accommodate changes without increasing the complexity of the original design.*

9.4.1.3 Reliability

Reliability is a determinant of system quality and a critical goal where the cost of failure is high (e.g., in terms of equipment replacement or human lives). Software Reliability is the probability that the software in a system will perform without failure under specified conditions or use.

Critical software reliability issues must be addressed early in the design process. Reliability must be built in up-front to prevent errors during conception, design, and development, as well as to recover from failures during operations. Reliability can only be designed in! It cannot be tested in nor can it be included as a retrofit due to an after thought. The goal of optimum system performance is well-engineered software which is 100% reliable. If the software can be repaired instantaneously without disrupting its operation, it also has 100% availability, regardless of how often it fails. [HUMPHREY89]

9.4.1.4 Safety

Software safety is closely tied to software reliability, and is the guarantee that the system will not fail under stressed operational conditions. Like software reliability, the issues of software safety must be addressed squarely during program planning and development. Ideally, this means the software will be free of defects. However, because software is created by humans, no matter how carefully the system is designed, coded, and tested, the probability for defects caused by human error are always be present. Until we have conquered the human factor through techniques such as highly automated development environments, safeguards must be built into all software upon which human safety, and indeed survival, are dependent. The cost for building in these safeguards must, therefore, be factored into your cost, schedule, and resource estimates.

While Defense Nuclear Agency regulations cover the area of software safety for nuclear weapons, software safety must be a topic of concern especially for all weapon system and C3 systems. One technique for enhancing safety is to employ software fault-tolerance methods in critical applications or application segments. One example of software fault-tolerance is the recovery block technique, where a failure in the primary program is bypassed by executing an independent alternate program that, hopefully, will execute successfully. Other fault-tolerance techniques are multi-version programming and exception handling. It is essential to require stringent fault-tolerance methods such as exception handling for flight control systems, C3 surveillance and sensor systems, and other systems where system aborts requiring restarts are unacceptable and potentially life-threatening.

9.4.1.5 Efficiency

Efficiency is an important software capability which refers to the highest and best use of critical resources. Processor cycles and memory locations are considered critical resources. Efficiency is a performance requirement that must be addressed during software requirements analysis. Efficiency can also be achieved during the coding phase — the last point where nanoseconds or bits can be squeezed out of software performance. Three factors should be considered when addressing efficiency requirements: (1) software should be as efficient as required — not as efficient as possible; (2) good design can improve efficiency; and (3) code efficiency and code clarity go hand-in-hand and should not be sacrificed for nonessential improvements in performance.

Source code efficiency is a direct result of algorithm efficiency defined during detailed design. Many compilers have optimizing features that automatically produce efficient code by breaking down repetitive expressions, using fast arithmetic, and applying efficiency-related algorithms.

In the MIS world, memory efficiency is not equated to the minimum memory used. Memory efficiency takes into account the paging characteristics of an operating system. Code location or maintenance of functional domains by way of structured components is one way well-engineered software reduces paging, and hence, increases efficiency. In the weapon system software world, memory restrictions are a real and critical concern, although low-cost, high-density memory is rapidly evolving. [History has shown, ironically, that whatever memory is available is how much the software will need!] Memory restrictions are generally a product of the size and weight limitations for housing and shielding processors. If system requirements demand minimal memory, compilers must be carefully evaluated for memory compression, or as a last resort, assembly language may have to be used. Unlike other software system characteristics that must be juggled against each other, techniques for run-time efficiency can sometimes lead to memory efficiency. The key to well-engineered software with high memory efficiency is keep it simple.

There are two classes of input/output (I/O) efficiency, external and internal. External I/O efficiency measures the user interface. Input supplied by the user and output produced for the user are efficient when the information supplied is easily understood. Internal I/O efficiency evaluates the I/O directed from one device to another device (e.g., from a computer to a disk or to another computer) or among modules within the same system. This measure is usually expressed in terms of hardware speeds, but the true measure is in throughput that includes processing time required to process data and transmit it from one module to another. [PRESSMAN92]

9.4.1.6 Understandability

Understandability is an important goal for the management of complexity. It is the link between the statement of the problem and the corresponding solution. For software to be understandable, it must reflect a natural view of the world. Achieving of this goal involves producing a solution to the stated problem in the form of an effective, understandable architecture. Capturing such a structure in software is necessary for it to be supportable, efficient, and reliable.

Different factors make software understandable. Well-engineered software is readable as a result of proper coding and proper documentation (including interface documentation). Well-engineered software also represents an accurate, understandable model of the real world. Understandability is achieved when the data structures (objects) and algorithms (operations) in the software solution are easily distinguished from one another. Understandability is also dependent on the programming language chosen to express the solution. [BOOCH94]

9.4.2 Software Engineering Principles

The goals discussed above are generic in nature and applicable to any software system — large or small. Once you understand these goals, you must employ a structured, disciplined development approach to achieve them. Administering sound engineering practices, based on the following principles, produces solutions that are functional, supportable, reliable, safe, efficient, and understandable.

9.4.2.1 Abstraction and Information Hiding

- **Abstraction.** One reason major software-intensive acquisitions fail is our inability to deal with software complexity. Abstraction is the software engineering principle for managing *complexity*. The purpose of abstraction is to separate the essential characteristics of a process, or its data dependencies, from all nonessential details. Abstraction is also performed during software design where the problem is decomposed into increasing levels of detail (or decreasing levels of abstraction). A analogy can be made from the road map example. When you plan a trip across country you start with a map of the United States (high-level). As you go through each state you use a map of the state in which you are traveling (mid-level). As you approach your destination and are looking for a particular address, you use a city map (low-level).

The software engineering process is, itself, an example of the abstraction principle. Each step in the process is a refinement of the abstraction level of the end product — the solution. During systems engineering, software is abstracted to a component of a software-intensive system. During software requirements analysis, the software solution is defined in terms that relate to the problem environment or function it must perform. The level of abstraction is reduced further as you proceed from architectural design to detailed design. Ultimately, the lowest level of abstraction is reached when the source code is written.

As the solution is decomposed into its component parts, each module in the decomposition becomes a part of the abstraction at a given level. Abstraction can be applied to both the algorithms and data in the solution. Thus, the logic of a software solution can be expressed in terminology that describes the problem domain rather than in software-dependent terms. Eventually, the details of expressing the problem will have to be addressed in software terminology — and ultimately in code. However, they can be deferred to lower levels where attention to essential details can be worked and/or reworked without impact on other system levels. Thus, the number of items tackled at one time are reduced to a manageable amount because attention is focused on the current level of decomposition. Abstraction promotes the goals of understandability and maintainability.

- **Information hiding.** Where abstraction separates essential and nonessential details at any given level, the purpose of information hiding is to make inaccessible those details that do not affect other parts of the software system. The principle of information hiding is to design modules such that the information contained within a module is inaccessible to other modules having no use for it. *Hiding* means that modularity can be effectively accomplished by defining a set of independent modules. The only information passed among the modules is that which is necessary to achieve functionality.

Abstraction promotes software maintainability and understandability by reducing the number of details a developer is required to know at any given level. It also details the procedural (or informational) entities making up the software. Hiding defines and governs access limitations to procedural information within a given module and any local data structure used by the module. By including information hiding as a design criterion for modular systems, well-engineered software reaps the greatest benefits when modifications are made during testing and later during software maintenance. With most data and procedures hidden from other parts of the application, reliability is enhanced. In addition, inadvertent defects introduced during modifications are less likely to spread to other modules. [BOOCH94]

The benefits of abstraction and information hiding apply to all software engineering goals. Abstraction supports modifiability and understandability by reducing the amount of detail at any given level. Information hiding enhances software reliability, because at each level of abstraction, only essential operations are permitted. Operations that obstruct or confuse the logical structure are also hidden.

9.4.2.2 Modularity and Localization

- **Modularity.** The principle of modularity has been around for almost 40 years and applies to the physical software architecture. Organizing very large applications into discrete, separately named and addressable modules allows us to intellectually manage software complexity. Also, with the right selection of module contents, the physical architecture can be made to correspond with the logical architecture, making the overall system more supportable and extendable.

Modules can be *functional* (procedure-oriented) or *declarative* (object-oriented). Because reliability must be *built in*, well-engineered software has well-defined *interfaces* connecting its modules. No matter how well-defined a module is, it must be able to interact with other modules. Coupling is the measure of interface tightness between modules. *Loosely coupled* modules can be treated relatively independently from others, and are easier to interface once integrated. How tightly bound or related the internal module elements are to one another is called cohesion. Modules with *strong cohesion* are desirable because their internal components have similar functionality and logical interdependence, making the modules basically self-contained. Self-contained modules are conceptually easier to handle and permit teams of programmers to work independently from each other.

- **Localization.** Applying the principle of localization helps create modules with loose coupling and strong cohesion. The principle of localization deals mainly with *physical location*. A module that has strong cohesion has a collection of logically-related resources physically located within it. Localization also implies that modules are as independent of other modules as possible (i.e., well-engineered software has a loosely coupled organization among its modules).

The principles of modularity and localization support the goals of modifiability, reliability, and understandability. In well-structured software, any given module is understandable — independent of other modules. Since design decisions are localized in given modules, modification can be limited to a small set of modules. In addition, if modularization has been successful, there will be limited and looser interconnections among modules. [BOOCH94] This results in greater reliability as defects in loosely coupled modules do not impact the performance of neighboring modules to the extent that tightly coupled ones do.

9.4.2.3 Uniformity, Completeness, and Confirmability

Abstraction and modularity are the most important principles used to control software complexity. But they alone do not ensure that the software is consistent and accurate. Uniformity, completeness, and confirmability provide these properties.

- **Uniformity.** The principle of uniformity directly supports the goal of understandability by ensuring modules use consistent notation and are free from unnecessary differences. Uniformity results from good coding practices with a consistent control structure and calling sequences for operations where logically-related objects are represented the same at any level.
- **Completeness.** The principles of completeness and confirmability support the goals of reliability, efficiency, and modifiability by aiding in the development of solutions that are accurate. Where abstraction extracts the essential details of a given problem set, completeness ensures that all important elements are included. Abstraction and completeness guarantee that the modules developed are *necessary* and *sufficient*. Efficiency can be improved because lower-level implementation can be fine-tuned without affecting higher-level modules.
- **Confirmability.** The principle of confirmability means the software is decomposable so it can be readily tested, thus enabling modifiable software. The principles of completeness and confirmability are not easily applied. A programming language with strong typing (such as Ada) facilitates the production of confirmable software. Software management tools are also used to ensure software is complete and confirmable.

9.5 Managing Software Engineering

Management is *the* key element in the engineering process as it permeates the entire life cycle. To conduct a successful software acquisition program, you must understand the scope of the work to be accomplished, the risks you will incur, the resources required, the tasks to be performed, the milestones to be tracked, the effort (including cost) to be expended, and the schedule to be observed. To be a successful manager, you must understand all facets of your program and rely on educated, experienced software professionals who understand and can implement software engineering process complexities. Sound management starts before the technical work begins, continues as the software matures from a concept to a functional reality, and only ends when you or the system is retired. [PRESSMAN92]

There are three basic activities you must perform as a manager to ensure program success. These activities are:

- You must plan;
- You must manage; and
- You must measure, track, and control.

Having made the software engineering commitment, as policy prescribes, the following items *must* be addressed to ensure your program is on the right track and that your developer is *engineering* your software. These software engineering management activities include the following [discussed in the indicated chapters]:

- Risk Management [Chapter 6],
- Software Development Maturity [Chapter 10],
- Software Estimation, Measurement and Metrics [Chapter 13],
- Reuse,
- Software Tools, and
- Software Support [Chapter 12].

Figure 9-12 illustrates how the software engineering management activities discussed here flow into the software life cycle. As you can see, process improvement and risk management are performed continuously throughout the system's life. Consideration of software development maturity and the contractor's commitment to continuous improvement is essential during source selection. As process improvement succeeds, software development maturity will advance. Once requirements are specified, an architecture can be defined that addresses the system from a domain perspective with regards to the need for open *systems*. The detailed design concentrates on building in quality attributes which include the optimum implementation of *reuse* and *COTS*. Prototyping and demonstrations are used to reduce risk and validate that the design addresses user and technical requirements. An appropriate design language should be chosen and used for coding. Models are used throughout the life cycle to define development procedures and analyze metrics data, which are collected throughout. Software engineering tools encompass the entire spectrum of development, and should be used to aid in the implementation of software engineering methods and life cycle activities.

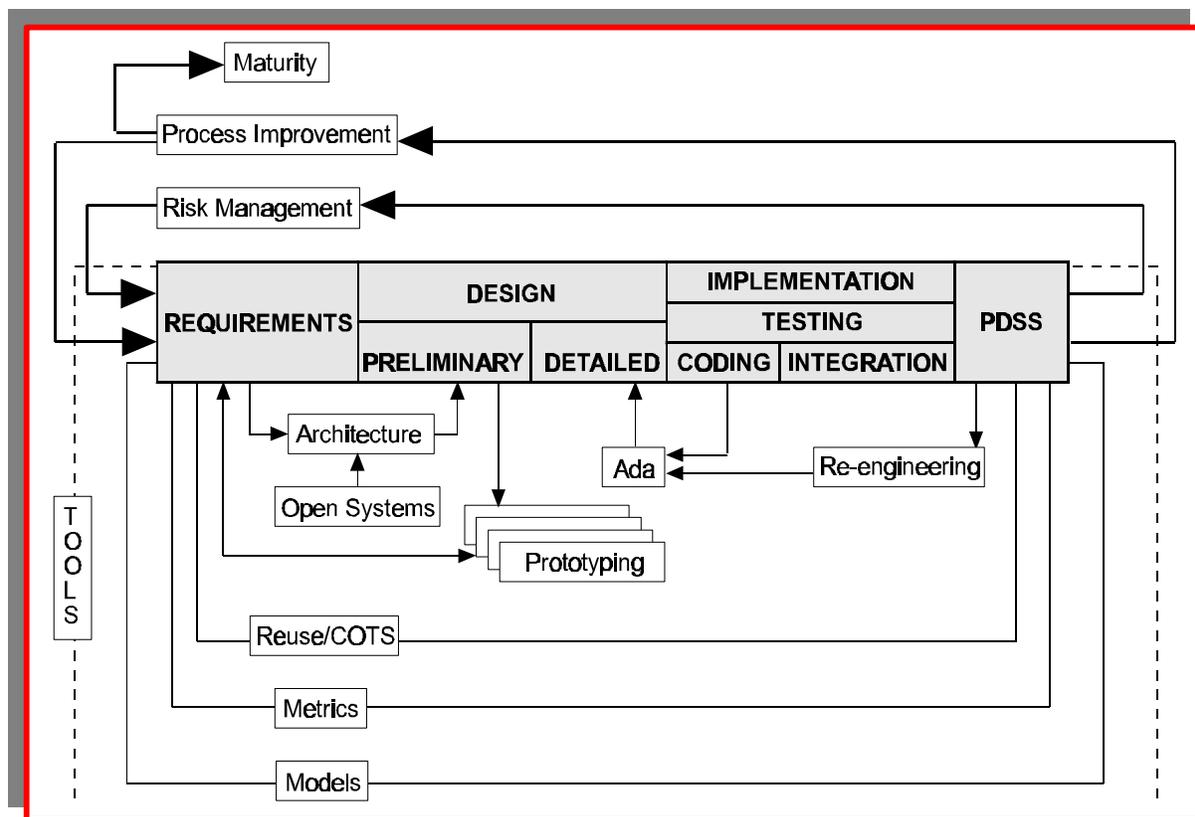


Figure 9-12. Software Engineering Relationship to the Software Life Cycle

9.5.1 Software Engineering Information

An excellent source of information on software engineering is *CrossTalk*, the monthly publication of the Software Technology Support Center (STSC). [See box below for information on how to subscribe.] The need for continued advances in software engineering management and methods is magnified as our reliance on commercial practices and products increases, which also compete in the global marketplace. Timely, topical articles are a means of keeping informed and up-to-

date on the latest developments in technology and professional practices amid the momentum of change within the software industry. *CrossTalk* is a high-quality, accurate information link between software managers and practitioners throughout the field. Distributed without charge, *CrossTalk* is highly recommended reading for its currency and technical content.

Another periodical, *Chips*, is a DoD magazine sponsored by the Navy. It has a different focus than *CrossTalk*, covering microcomputer issues, including contracting, networking, software development, policy, training, etc. However, it occasionally has some software engineering-related articles. *Chips* is distributed free to all government users. Contact the Naval Computer and Telecommunications Area Master Station LANT through e-mail at chips@email.chips.navy.mil or view *Chips* electronically on the Internet at <http://www.chips.navy.mil>.

TO SUBSCRIBE: For more information on a free subscription to *CrossTalk*, contact the Software Technology Support Center (STSC), Attention: Customer Service, OO-ALC/TISE, 7278 Fourth Street, Hill AFB, Utah 84056. Phone: (801) 775-5555 or DSN 775-5555, Fax: (801) 777-8069, or DSN 777-8069. E-mail: consulting@stsc1.hill.af.mil or Internet at <http://www.stsc.hill.af.mil>.

9.6 What is Information Engineering?

If you have ever visited the sunny Silicon Valley in California, one of the most popular local curiosities is an enormous house built around the turn of the century by the rifle heiress, Sarah Winchester. As Sarah grew older, she believed she was being haunted by the ghosts of people killed by her husband's rifles. Terrified of meeting these angry souls in the hereafter, she employed two full-time spiritualists who advised her that she would never die as long as her house kept living and changing. For 38 years, the construction of towers, wings, chimneys, rooms, and gardens was nonstop. Sarah's vast fortune was employed to guarantee the constant sound of workers pounding nails, laying cement, digging holes, and chiseling wood. Everything needed to keep the operation going was on-site — wood shops, cement mixers, warehouses, and supply yards. Because the construction engineers never had a set of overall blueprints showing where the house was going, some rooms were remodeled more than a dozen times. Over the years, throughout this frenzy, oddities began to appear. The house has stairways leading into ceilings, windows blocked by walls, more halls and passages than rooms to connect, a three-story chimney that fails to meet the roof, and many rooms that serve the same purpose.

Like the Winchester Mystery House, the information systems of many large organizations and corporations are under perpetual construction — growing, changing, duplicating, multiplying. There are expanding databases here, new input screens there, spreadsheets everywhere — some systems are changed, updated, and enhanced more than a dozen times. Often vast fortunes are spent keeping these activities going with everything needed to do the job on-site. Over the years oddities begin to appear. The collection of software systems contains masses of unused reports, more bridges and interfaces than systems to connect, data that are inconsistent, redundant, inaccessible, and in incompatible formats, with many systems serving the same purpose. These enormous mystery systems live and change without a set of overall blueprints for the data, systems, and technology needed to support the enterprise. [SPEWAK93]

In the early 1980s, to help stop the constant custom building and replacing of systems with costly odd features, James Martin developed the *information engineering* methodology. [MARTIN81] Intending it to contrast with, and complement, software engineering, Martin defined IE as,

“The application of an interlocking set of formal techniques for the planning, analysis, design, and construction of information systems on an enterprise-wide basis across a major sector of the enterprise.” [MARTIN89]

Information engineering is a form of domain engineering oriented towards the MIS domain, which has also proven successful in analyzing C2 systems. IE is predicated on the realization that the procedures for conducting business are in constant flux due to frequent restructuring and changes in organizational focus; whereas, the data requirements of the enterprise are stable. In traditional approaches, database design is dictated by application data requirements developed to automate specific procedures. Every time procedures change, the database must be redesigned. Changing database design to accommodate a change in one procedure has a snowball effect requiring maintenance on all other system components accessing the changed part. Understandably, systems designed this way have extremely high maintenance costs. The goal of IE is to capture the stable data requirements of the enterprise in a database design that remains stable throughout the software life cycle. Dynamic elements are captured in those applications (modules) always subject to change. This process results in substantial maintenance cost savings. [MICAH90] Because IE focuses first on data rather than on procedures, it is called a *data-driven* method.

9.6.1 Information Engineering Process

According to Finkelstein, Martin’s co-author, the IE process is characterized by two distinct stages: a technology-independent and a technology-dependent stage, as illustrated in Figure 9-13. The starting point is strategic business planning which allows for continual evaluation and refinement of the Strategic Plan at all stages of development, as illustrated in Figure 9-14. Using this method, feedback is quick, exact, and effective, with clear communication links and precise implementation. [FINKELSTEIN92]

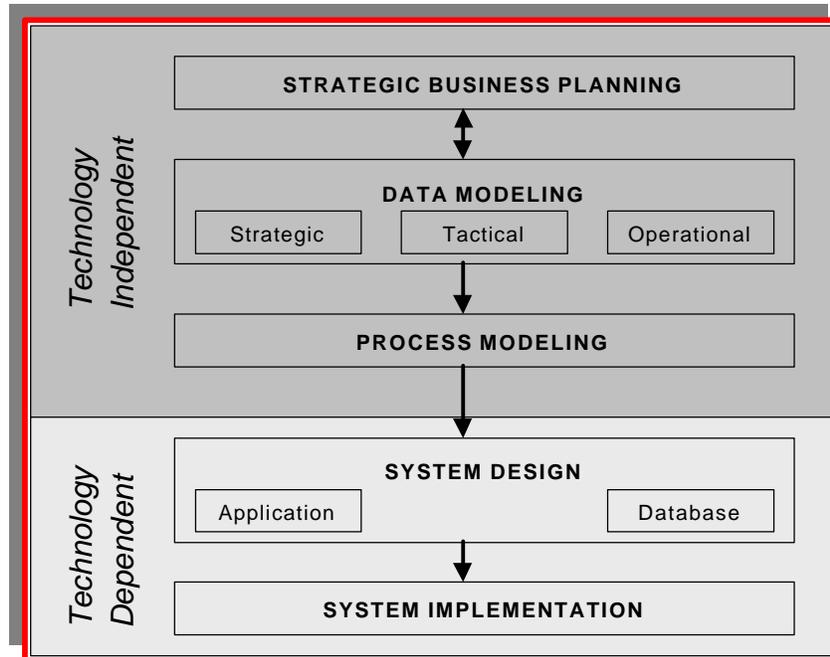


Figure 9-13. Information Engineering Phases [FINKELSTEIN92]

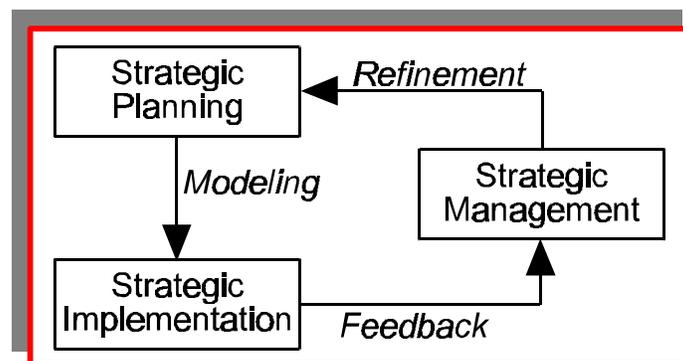


Figure 9-14. Strategic Management Planning [FINKELSTEIN92]

9.6.2 Information Engineering Architecture

IE addresses four architectural levels which separate data and process, thus creating databases and applications that are flexible and facilitate rapid changes and enhancements in response to competitive pressures. These levels are illustrated in Figure 9-15.

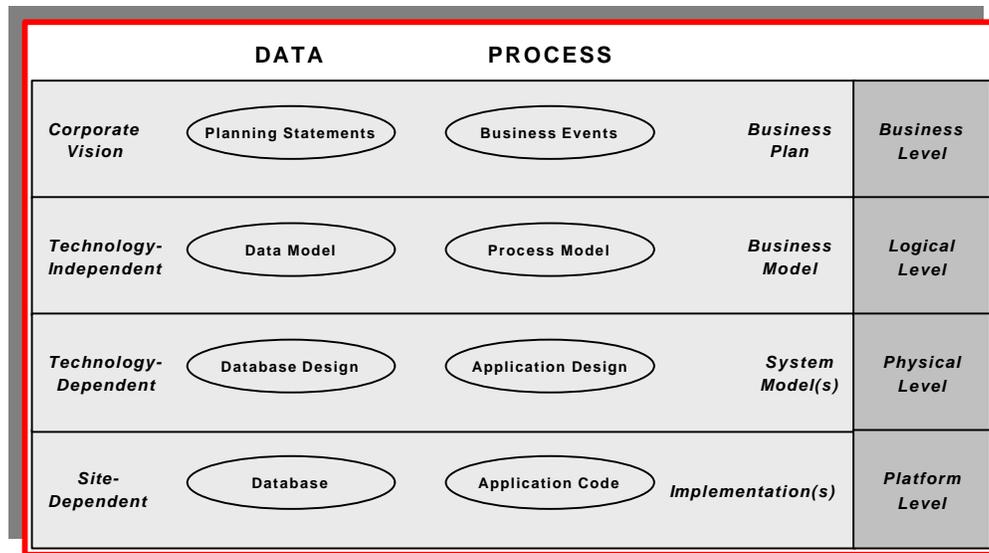


Figure 9-15. Information Engineering Four-Level Architecture [FINKELSTEIN92]

- **Level 1: Business.** This reflects the corporate vision because data based on strategic planning statements are defined at all management levels. The business plans operate on these data, based on planning statements and business events which process the data.
- **Level 2: Logical.** Planning statements based on the corporate vision are used to develop technology-independent data models. Process models, represented by the business model, are developed from data models and business events based on the Business Plan.
- **Level 3: Physical.** Technology-dependent database designs are developed based on the data and process models. These database designs and relevant process models (representing the system models) provide input to application design (which also feeds back to database design).
- **Level 4: Platform.** The database design is physically implemented as site-dependent databases. Application code operating against them implements the databases and application design. Databases can be implemented and applications executed on specific platforms that employ the best available hardware, software, and communications technologies. [FINKELSTEIN92]

NOTE: Make sure someone in your program office understands IE well enough to interpret and review contractor IE products. Otherwise, there is the risk that the stacks of paper produced by this process will be incorrect or ignored.

IE is discussed as an example of one method for modeling data. Other methods for MIS are also viable, such as essential systems analysis [McMENAMIN84], Enterprise Architecture Planning [SPEWAK93], object-oriented analysis [COAD90], and IDEF [described next]. The approach you adopt requires research and an understanding of your program to determine which is most applicable to your program-specific needs.

9.6.3 IDEF

Integrated Computer-Aided Manufacturing Definition Language (IDEF) is a modeling technique that supports IE. It was initially developed in the 1970s for Air Force Logistics Center support programs in the manufacturing environment. [See FIPS Pub (Federal Information Processing

Standards Publication 183: Integration Definition For Function Modeling (IDEF0) In 1989, an IDEF users' group was formed to establish a methodology for implementing the IDEF approach which provides a framework for classifying important information about an enterprise. *The main goal of the IDEF exercise is to identify areas for process improvement.* Improvements can be in the areas of:

- Manual procedures and techniques,
- Product and service quality,
- Industrial processes and factory automation,
- Information systems and computer automation,
- System development methods, and
- Business procedures, to name a few.

IDEF activity modeling captures and graphically depicts the specific steps, operations, and data elements needed to perform an enterprise activity. An activity is defined as a named process, function, or task that occurs over time and has recognizable results. As illustrated in Figure 9-16, each activity is represented by a rectangle. Entering, exiting, or linking activities are those factors that change the activity. These fall into the categories of:

- **Input data** or material for the activity (e.g., program requirements),
- **Controls** that regulate the activity (e.g., engineering principles, existing policies),
- **Output data** or materials produced by the activity (e.g., quality software), and
- **Mechanisms** comprised of people or machines that perform the activity (e.g., new technology).

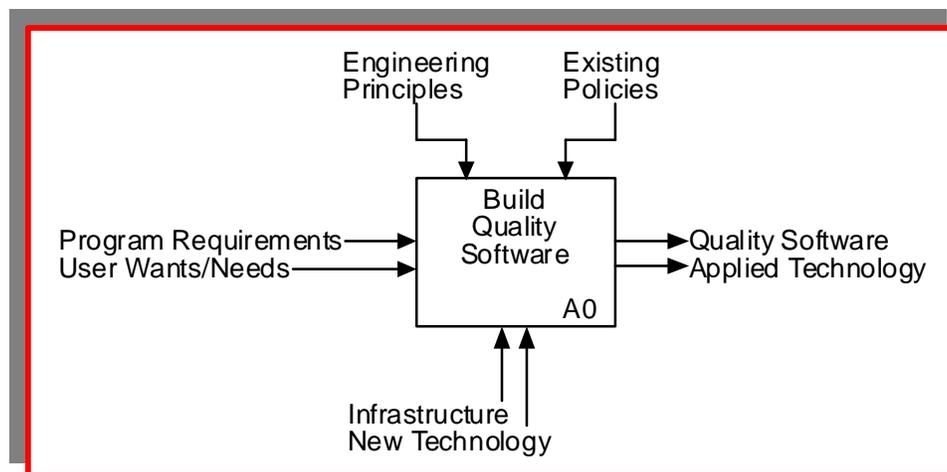


Figure 9-16. IDEF (Level A0) for Nominal Program

The interrelationships among activities are modeled by using node trees, as illustrated in Figure 9-17. An activity can be decomposed into subactivities which can be further decomposed into sub-subactivities. Context diagrams and decomposition diagrams are used to provide both overall and more detailed breakdowns of activities. In a typical program, the scope and requirements are defined first. Then the information required to support the activities is gathered through a series of working sessions that include users and systems experts. These data are finally captured in an automated tool for documentation.

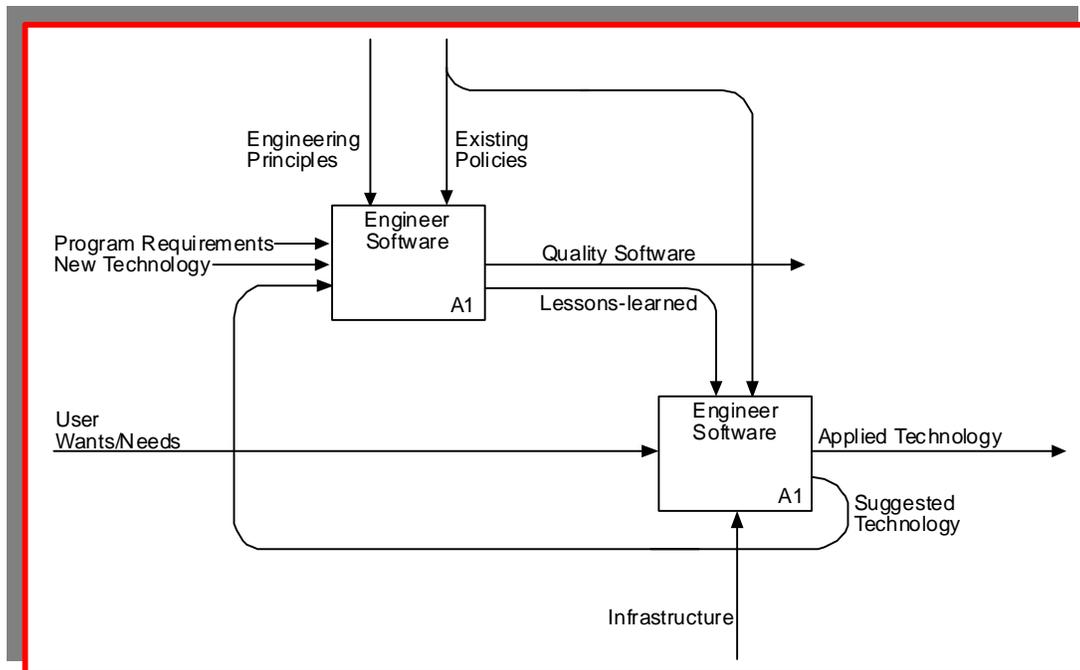


Figure 9-17. IDEF (Decomposition of A0) Model for a Nominal Program

The IDEF approach is recommended *before* starting new MIS programs. It aids in mission area analysis, functional analysis, and the strategic planning. IDEF can be also used to model *alternative views* of the enterprise. The IDEF approach uses hierarchically decomposed function models and entity-relationship (E-R) diagrams. [An E-R diagram identifies data objects and their relationships through a graphical notation.] These views then become the baseline upon which to plan and implement process improvement. The IDEF modeling approach leads to an understanding of the total enterprise by integrating business tasks, rules, and objectives to work together in a productive way.

9.7 Success Through Engineering

Mosemann summarized why engineering is the solution for successful software-intensive systems acquisition and management when he emphasized that,

“...we’ve got to adopt an engineering focus. We have got to concentrate on cost-effective solutions, solutions that are built from models, and on using capable, defined processes, rather than focusing on perfect systems that meet 100% of our wishes. Again, this is a management challenge, not a technical challenge. There’s just no way to manage or to control the configuration, to control the side-effects, in these kinds of large software developments unless we use engineering discipline.”
[MOSEMANN92¹]

As illustrated in Figure 9-18, software engineering requires more emphasis and resources up-front during the development effort. This change from a traditional software-as-art approach (where most of the resources are spent in the support phase) to a software engineering approach reduces the total amount of resources necessary, since the resultant software support costs are substantially reduced. Well-engineered software lays a solid foundation for the system to evolve

into its operational environment by employing sound development practices and procedures. Through the software engineering discipline you will have available to you:

- Comprehensive methods for all software development phases,
- Better tools for automating these methods,
- More powerful building blocks for software implementation, and
- An overriding philosophy for coordination, control, and management. [PRESSMAN92]

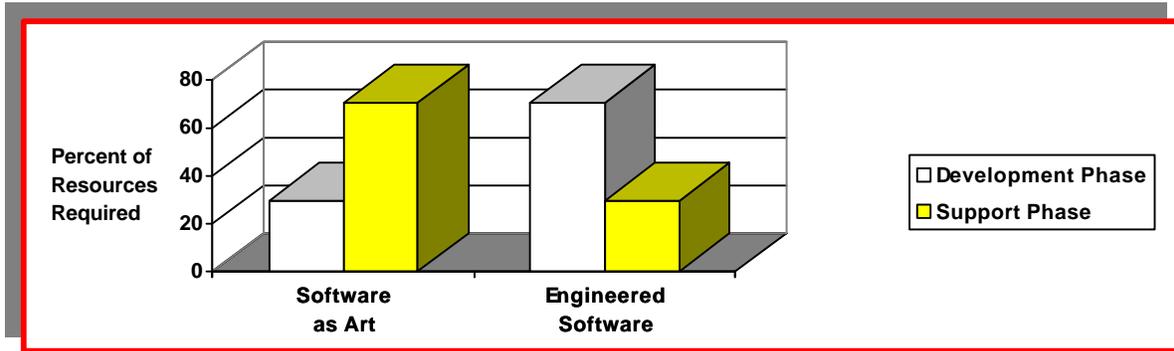


Figure 9-18. Software Engineering Builds a Solid Foundation Up-front

The challenge is to acquire and develop systems that meet the user's needs given the usual performance, life cycle cost, and schedule criteria. However, the only way to meet this challenge and achieve acquisition success is to use engineering discipline in all aspects of software development. Anything less will only produce schedule slips, cost overruns, and systems that do not meet user needs.

9.8 References

- [Ada/C++91] *Ada and C++: A Business Case Analysis*, Office of the Deputy Assistant Secretary of the Air Force, Washington, DC, June 1991
- [BOOCH94] Booch, Grady and Doug Bryan, Software Engineering with Ada, Third Edition, Benjamin/Cummings Publishing Company, Redwood City, California, 1994
- [COAD90] Coad, Peter and Edward Yourdon, Object-Oriented Analysis, Yourdon Press, Prentice Hall, Englewood Cliffs, New Jersey, 1990
- [DSMC90] Systems Engineering Management Guide, Defense Systems Management College, U.S. Government Printing Office, Washington, DC, 1990
- [EIA632] Electronic Industry Association (EIA), Interim Standard 632, (draft), *Systems Engineering*, September 20, 1994
- [FINKELSTEIN92] Finkelstein, Clive B., "Information Engineering: Strategic Systems Development," Jessica Keyes, editor, Software Engineering Productivity Handbook, Windcrest/McGraw-Hill, New York, 1992
- [HOLIBAUGH92] Holibaugh, Robert, "STARS Domain Analysis Survey" briefing, Software Engineering Institute, June 15, 1992
- [HUMPHREY89] Humphrey, Watts S., Managing the Software Process, Software Engineering Institute, Addison-Wesley Publishing Company, 1990
- [KERSH90] Kersh, Pvt Gerald, as quoted by Robert A. Fitton, ed., Leadership: Quotations from the Military Tradition, Westview Press, Bolder, Colorado, 1990
- [KOGUT94] Kogut, Paul, Kurt Wallnau, and Fred Maymir-Ducharme, "Software Architecture and Reuse," TriAda '94, Baltimore, MD
- [LYONS92] Lyons, Lt Col Robert, as quoted by David Hughes, "Digital Automates F-22 Software Development with Comprehensive Computerized Network," *Aviation Week & Space Technology*, February 10, 1992
- [MARTIN81] Martin, James, and Clive B. Finkelstein, Information Engineering, Savant Institute, Carnforth, Lancs, United Kingdom, 1981
- [MARTIN89] Martin, James, Information Engineering, Book 1 (of 3): Introduction, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989
- [MAYMIR95] Maymir-Ducharme, Fred and David Weisman, "US Air Force Comprehensive Approach to Reusable Defense Software (CARDS) Technology Transition Program: Reuse Partnerships," Reuse '95, Morgantown WV 1995
- [McGRAW89] Dictionary of Scientific and Technical Terms, Fourth Edition, Sybil P. Parker, Editor-in-Chief, McGraw-Hill Book Company, New York, 1989
- [McMENAMIN84] McMenamin, Steve and John Palmer, Essential Systems Analysis, Yourdon Press, Prentice-Hall, Englewood Cliffs, New Jersey, 1984
- [MICAH90] "Micah Systems, Inc. and Information Engineering," October 24, 1990
- [MOSEMANN91] Mosemann, Lloyd K., II, "Air Force Software in the Year 2000," keynote luncheon address, STSC-HQ USAF/SC Joint Software Conference, Salt Lake City, Utah, April 16, 1991
- [MOSEMANN92¹] Mosemann, Lloyd K., II, "Comments on MIL-STD-499B," October 23, 1992
- [MOSEMANN92²] Mosemann, Lloyd K., II, "Software Management," keynote closing address, Fourth Annual Software Technology Conference, Salt Lake City, Utah, April 16, 1992
- [MOSEMANN92³] Mosemann, Lloyd K., II, "Software Measurement and Quality," keynote address, Fourth Annual REVIC User's Group Conference, Fairfax, Virginia, March 25, 1992
- [PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, Inc., New York, 1992
- [SHINA91] Shina, Sammy G., "Concurrent Engineering," *IEEE Spectrum*, July 1991

- [SPEWAK93] Spewak, Steven H., with Steven C. Hill, Enterprise Architecture Planning: Developing a Blueprint for Data, Applications, and Technology, QED Publishing Group, Boston, 1993
- [STRASSMANN91] Strassmann, Paul A., as quoted by Bob Brewin, "Corporate Information Management White Paper," *Federal Computer Week*, September 1991
- [STRASSMANN92] Strassmann, Paul A., "Joining Forces to Engineer Success: The DoD Context," opening keynote address, Fourth Annual Software Technology Conference, April 14, 1992
- [WAGNER95] Wagner, Capt Gary F., and Capt Randall L. White, "F-22 Program Integrated Product Development Teams: How One Major Aircraft Program Developed Integrated vs. Independent Product Teams," *Program Manager*, Defense Systems Management College Press, July-August 1995
- [WANG86] Wang, An, as quoted in *Boston Magazine*, December 1986
- [ZELLS92] Zells, Lois, "Learn from Japanese TQM Applications to Software Engineering," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992
- [ZRAKET92] Zraket, Charles E., "Software Productivity Puzzles, Policy Changes," John A. Alic, ed., Beyond Spin-off: Military and Commercial Technologies in a Changing World, Harvard Business School Press, Boston, Massachusetts, 1992