Fundamental Principles of Good System Design

A. Terry Bahill, PE, University of Arizona
Rick Botta, BAE Systems

Abstract: This article presents dozens of fundamental principles of good system design that should help make a product better. Not surprisingly, many of these same principles will help make a product reusable in a new system and will help reduce redesign costs when requirements change. These principles apply to simple systems and complex systems. These principles come from hardware, software, system, and test design, but they are general and many can be applied in a large variety of fields (even non-engineering fields).

Keywords: System Design, Reuse, Requirements, Systems of Complex Systems

EMJ Focus Areas: Systems Engineering

Design is a creative activity—consequently, there is no process that will *guarantee* good designs, but there are some principles that will increase the probability of getting a good design. This article presents dozens of fundamental principles of good system design that should help make a product better. Using these principles will also make a product more reusable for future systems and it will help reduce redesign costs when requirements change. Of course, the customer may mandate or exclude the use of some or all of these principles. Some of these principles are as follows:

- Use models to design systems
- Use hierarchical, top-down design
- Work on high-risk items first
- Prioritize
- Control the level of interacting entities
- Design the interfaces
- Produce satisficing designs
- Do not optimize early
- Maintain an updated model of the system
- Develop stable intermediates
- Use evolutionary development
- Understand your enterprise
- State what, not how
- List functional requirements in the use cases
- Allocate each function to only one component
- Do not allow undocumented functions
- Provide observable states
- Rapid prototyping
- Develop iteratively and test immediately
- Create modules
- Create libraries of reusable objects
- Use open standards
- Identify things that are likely to change
- Write extension points
- Group data and behavior
- Use data hiding
- Write a glossary of relevant terms
- Envelope requirements
- Create design margins
- Design for testability
- Design for evolvability
- Build in preparation for buying
- Create a new design process
- Change the behavior of people

These design principles come from the experience of hundreds of engineers and managers. The particular references cited in the following paragraphs are not meant to be *the* authority—they are merely examples that have references in the literature.

Use models to design systems: System design can be requirements based, function based, or model based. Model-based system engineering and design has an advantage of executable models that improve efficiency and rigor. One of the earliest developments of this technique was Wymore's (1993) book entitled *Model-based System Engineering*, although the phrase "model-based system design" was in the title and topics of Rosenblit's (1985) PhD dissertation. Model-based systems engineering depends on having and using well-structured models that are appropriate for the given problem domain (Bahill and Szidarovszky, 2008). Bahill's models start with the use cases.

Use hierarchical, top-down design: Early on, translate the customer's needs into goals, capabilities, and functions; these provide guidance for all future development. Work on high-level functions first because, although high-level functions are less likely to change, when they do change, they force changes in many other functions. Decompose systems into subsystems, subsystems into sub-subsystems, etc. (Chapman and Bahill, 1992). In software, this decomposition is called layered architecture (Evans, 2004). Implementation is simpler if the dependencies and action initiations between these layers are unidirectional.

Work on high-risk items first: Work on high-risk items first in order to reduce risk; in addition, high-risk items are more likely to change, thereby producing changes in other entities, so working the high-risk items first will reduce the rework due to changing requirements. Furthermore, if it was impossible to satisfy the high-risk capabilities and the project was cancelled, you will have saved the money that otherwise would have been squandered satisfying low-risk requirements (Jacobson, Booch, and Rumbaugh, 1999). The original spiral model of Boehm (1988) advocated risk-driven development.

Refereed management tool manuscript. Accepted by Associate Editor John Farr.

Prioritize: Requirements, goals, customer needs, capabilities, risks, directives, initiatives, issues, activities, features, functions, use cases, technical performance measures, and weights of importance for the criteria in tradeoff studies should all be prioritized. Prioritization will help with budget, schedule, system architecture, customer satisfaction, and risk reduction (Botta and Bahill, 2007).

Control the level of interacting entities: Objects should exchange inputs and outputs with other objects at the same level, or perhaps at one level above or below (Bahill, Szidarovszky, Botta, and Smith, 2008). It is a mistake to have objects interact with objects two or more levels above or below (Simon, 1962). Some interlevel interactions may be specified as unidirectional (Evans, 2004); this simplifies the implementation.

Design the interfaces: Be sure to design your interfaces (Quintanar, 1999; Rechtin, 2000). Interfaces between subsystems and interfaces between the main system and the external world must be designed. Subsystems should be defined along natural boundaries. When the same information travels back and forth among different subsystems, a natural activity may have been fragmented. In the automobile industry, they do not build a car frame, send it across town to install the motor and then send it back to put on the axels and wheels. Subsystems should be defined to minimize the amount of information exchanged between the subsystems (Rechtin, 2000). Well-designed subsystems send finished products to other subsystems. Another way of stating this is to minimize the coupling between subsystems. Interfaces should connect things of similar levels (Bahill, Szidarovszky, Botta, and Smith, 2008). Feedback loops around individual subsystems are easier to manage than feedback loops around interconnected subsystems. When possible, different entities should use the same interface, rather than having a specialized interface for each entity (Schultz, Fricke, and Igenbergs, 2000). Browning (2002) uses a design structure matrix to analyze and modify the interfaces in a process. Special care should be given to interface design so that the interface does not have to change when its associated systems change. The name and the brief description of the interface should proclaim its intention.

Produce satisficing designs: Engineers should create satisficing designs—they should not try to produce optimal designs because for complex systems it is impossible to do. Simon (1957) says that the key to successful design is "the replacement of the goal of maximization with the goal of satisficing, of finding a course of action that is 'good enough.' ... Since the [designer] ... has neither the senses nor the wits to discover an 'optimal' path—even assuming the concept of optimal to be clearly defined – we are concerned only with finding a choice mechanism that will lead it to pursue a 'satisficing' path, a path that will permit satisfaction at some specified level of all its needs."

Do not optimize early: If optimization is absolutely required, do it late in the design process. If you optimize early in the design process, you will have to reoptimize every time the design changes. At the risk of being too specific, we suggest that optimization should not be done before Critical Design Review (CDR). This does not mean that you should not strive to find a good architecture early in the design process. It means do not use integer programming to minimize part counts or wire lengths before CDR. It means do not use optimization methods for routing problems on networks with stochastic failures before CDR.

Maintain an updated model of the system: Engineering must create a model that simulates the intended system at every level from both the business and technical viewpoints (Bahill, Botta, and Daniels, 2006). A single model should underlie analysis, design, implementation, and team communication (Evans, 2004). Update this model throughout the design process (Douglas, 2004). Update this model to "as built" in the implementation phase and "as modified" in the test, operational, and retirement and replacement phases. When the requirements inevitably change, engineering should review all decisions, revise the system model, rerun all simulations, and show the effects of these requirement changes on cost, schedule, performance, risk, and the designs of other systems (Wymore, 2004).

Every operational system should have an accurate model. This model should be run to evaluate performance enhancements and possible disasters. If the system is a regional electric power grid and it has been suggested that its capacity could be increased by adding solar and wind generators, then the system performance should be evaluated on the model before it is installed on the real system. If the system is a large building, and evacuation is being considered because a hurricane is approaching, then run the model to estimate possible hurricane damage. Every Interstate highway bridge should have a model that degrades as the bridge degrades. When the model predicts a dangerous situation, the bridge should be shut down. When a nuclear reactor is to be decommissioned, its model will be run to help formulate the decommission plan.

Develop stable intermediates: "A large change is best made through a series of smaller, planned, stable intermediate states" (Rechtin, 2000). These intermediate states should be stable enough that the progression could be stopped at predetermined points. With stable intermediates, if a large complex system is cancelled, for example because of loss of political support, then something useful for mankind will still exist. The Central Arizona Project was designed this way, but the Superconductor Supercollider in Texas was not (Moody, Chapman, Van Voorhees, and Bahill, 1997).

Use evolutionary development: The Department of Defense (DoD) created the evolutionary acquisition process. Using it, you create a usable system and later add requirements and money to get a more complex usable system (DoD 5000.1). The B-52 airplane is a quintessential example: every half-dozen years the Air Force added requirements and money and got new capability.

Understand your enterprise: Understand how the system you are designing fits into your enterprise. Frameworks help people organize and assess completeness of integrated models of their enterprises. Several popular frameworks have been used to architect enterprises. The Zachman framework, like many others, considers multiple perspectives and multiple aspects of an enterprise (Bahill, Botta, and Daniels, 2006).

State what, not how: State what function needs to be performed, not how to implement the solution. For example, you should say, "Play music." Do not say, "Play a CD," or "Hire a pianist," or "Turn on the radio and tune in a music station." In the object-oriented software world, this is called polymorphism (Rumbaugh, Jacobson, and Booch, 2005); however, one man's floor is another man's ceiling. First, state *what* the customer needs. Then figure out *how* to satisfy this need. This "how" then becomes a system feature. State *what* this feature must do. Then figure out *how* to implement it. This implementation then becomes a requirement that states *what* the system must do, etc.

List functional requirements in the use cases: Traditional requirements are typically written with textual statements of imperative. They are the primary means by which systems engineers bound and communicate system functions, capabilities, and constraints. The rapid adoption of use case modeling for capturing functional requirements in the software community has caused systems engineers to adopt use case models for capturing system-level functional requirements. We advocate a hybrid requirements process in which use case modeling and traditional shall-statement requirements are applied together to effectively express both functional and non-functional requirements (Daniels and Bahill, 2004). The use cases should contain the main and alternate scenarios. These describe the system functions. The next section of the use case should formally describe the functional requirements.

Allocate each function to only one component: Each function should be allocated to only one physical component, and, therefore, each function would have only one owner. If there were two owners for a function, one might change his or her requirements, and this would change the system for the other. In the object-oriented world, this would be phrased as, "Do not allow multiple actors to have the same role" (Övergaard and Palmkvist, 2005). If two actors are trying to assume the same role, generalize them into one abstract actor. There are exceptions to this principle, but they are unusual. Violation of this principle is captured in the American proverb, "Too many cooks spoil the soup." In the 20th century, the functions of gathering intelligence and tracking terrorists were allocated to the FBI, the CIA, the NSA, the Pentagon, etc. Until 9/11/01 the results were mixed. This principle is discussed in more detail in Appendix A.

Do not allow undocumented functions: Do not hide the existence of system functions. Make sure that all system functions are identified and described appropriately. Undocumented functions can exist due to developmental testing activities, accidents, playfulness, or malicious intent. The PDP 11-45 computer had secret op codes. Microsoft Excel 97 had a built in flight simulator. Thousands of such Easter eggs are documented at http://www.eeggs.com/tree/1-1-111.html. Don't try this at home, but starting a line with "=rand(200,99) (Enter)" (excluding the quotes) in a Word document or a PowerPoint presentation, will give you 235 pages of "The quick brown fox jumps over the lazy dog," and perhaps a computer crash.

Provide observable states: System equivalence cannot be proven using input/output behavior. State behavior must be used to prove equivalence of dynamic systems (Wymore and Bahill, 2000). It would be best to provide the total system state, but it would still be useful to provide only a significant number of reset states or restore points (Botta, Bahill, and Bahill, 2006). Providing states will allow future designers to reuse existing systems, upgrade systems, use commercial off the shelf products, replicate field failures, verify that a physical system conforms to its design, and verify evolving systems (Wymore and Bahill, 2000; Botta, Wuersch, and Bahill, 2004). It will also allow them to find a system mode behavior function that can be placed in front of the input to the system being reused in order to produce the desired new system behavior (Wymore, 1993).

Accelerometers from automobile air bag systems can be reused as g-switches in missile safing systems if the states are observable. If the only indication the accelerometer gives is *Fired* or *NotFired*, then they are not reusable. But the state of the accelerometer can be defined as the g-force being measured. If this g-force can be observed, then they can be reused in missiles.

Rapid prototyping: Develop a prototype quickly. Get the stakeholders to use it. This will identify the true needed capabilities of the system. This is particularly useful when delving into unfamiliar design areas. The first prototype is often discarded, particularly in software systems.

Develop iteratively and test immediately: Build a large-scale system by constructing it as a series of smaller products of increasing completeness. Because the early entities are small, it is easier to get them right and testing can be completed immediately (Douglas, 2004).

Create modules: Design the system with many small independent but interacting modules (Parnas, 1972). The principle of independence means that changing one module should not force changes in other modules (Suh, 1990). Good examples are the modules of the Unix (or Linux) computer operating system and the libraries of Java. Modules should have low external complexity (loose coupling) and high internal complexity (strong cohesion) (Schultz, Fricke, and Igenbergs, 2000).

Create libraries of reusable objects: Patterns are example solutions for doing common things. There are many books with software patterns (see, for example, Fowler, 2004) and some with use case patterns (Övergaard and Palmkvist, 2005); however, this library principle is not restricted to software and use cases. For example, a company could create 100 generalized evaluation criteria (measures of effectiveness) that could be tailored and used in company tradeoff studies. Each criterion would contain the name of the criterion, description, weight of importance (priority), basic measure, units, measurement method, input (with expected values or the domain), output, and scoring function (type and parameters) (Daniels, Werner, and Bahill, 2001). This criterion library would form the nucleus of a tradeoff study, but many customized criteria would be added. This particular library should be a part of the company CMMI DAR process (Chrissis, Konrad, and Shrum, 2003). Libraries could also be created for generic requirements.

Use open standards: Open standards are publicly available specifications for achieving a specific task. Open standards are available for all to read and implement. Examples of open standards include ANSI/EIA-632, ANSI/EIA 731, IEEE 1220, ISO 15288, UML, SysML, OMG IDL, CORBA and USB. In contrast, proprietary standards are controlled by a single entity. Java Server Faces can be the interface between HTML code and logic code. The Java Server Faces is a mass-market product that is highly scalable, but it is still controlled by a single commercial entity, likewise with Microsoft's Source Code Control Interface (SCCI) Specification.

Identify things that are likely to change: Differentiate between aspects of a system that are likely to change and those that are likely to remain relatively constant. Identify the aspects that are likely to change and put extra effort into designing their interfaces. For example, an air transportation system will always have functions such as load people, load luggage, take-off, land, and comply with FAA regulations. It also has entities that are sure to change such as the type of aircraft, speed, size, routes, and cargo capacity. The system should be designed to accommodate changes in the things that are likely to change (Evans, 2004). For example, the processing speed of computers is expected to increase; therefore, hardware and software should be designed so that changes are not necessary when processor speeds do increase.

Write extension points: In use cases use extension points where change is expected (Cockburn, 2001; Kulak and Guiney, 2000; Övergaard and Palmkvist, 2005). That way the base use case does not need to be changed when the extending use case is changed.

Group data and behavior: Group together data and the behavior (logic) that operates on it, because data and the behavior that accesses that data often change together (Fowler, 2004). Software accomplishes this with header files and with class diagrams that have attributes (data) and operations (behavior). Hardware accomplishes this with distributed control where each board has a processor and memory.

Use data hiding: Hide data from objects that do not have "a need to know" (Parnas, 1972). That way if the data structure is changed, the other objects do not have to be notified about the change. This principle is also called information hiding and function hiding (Gomaa, 2000). In the UML, model elements such as attributes and operations can be marked with one of these visibility indicators: public, protected, private, or package. In public documents, describe the interfaces and only the public functions and data. This principle has nothing to do with security —its purpose is simply to improve the efficiency of making changes. Architecture hierarchies and levels also influence data hiding.

Write a glossary of relevant terms: Collect terms used in the domain and design models and provide definitions. Capture the jargon of the domain experts. Include names of functions, classes, methods, modules, and high-level organizing principles. Evans (2004) calls this collection of terms a ubiquitous language.

Envelope requirements: Estimate requirements top-down and fill in details later, e.g., we need to test (measure) a DC voltage that surely will be less than 40 volts. Later we find that it will probably be between 5 and 20 volts. Finally, we get a specification of 12±2 volts. This principle ameliorates changing requirements than with reuse.

Create design margins: If your estimated need is two Gigahertz of bandwidth, specify four. But this is expensive: if everyone did this, the system would be gold-plated. It is better to give all of the flexibility in cost, schedule, and performance to the program manager and allow him or her to distribute it across the project. This principle is treated in detail in Appendix B.

Design for testability: Early in the design process, it should be determined how the system will be tested. The architecture should be selected to enable built-in self-test (BiST). BiST can be triggered externally or it can be performed whenever the system is not busy performing its normal functions (O'Conner, 2001).

Design for evolvability: The capacity of an existing system to successfully adapt to changing requirements throughout its life cycle is called evolvability (Christian and Olds, 2005). A system can adapt to change by reconfiguring existing system entities, by increasing the size of existing system entities, or by adding new entities. Often design, development, and testing take multiple years. Technologies can change during that period. Designs should have adequate flexibility and adaptability to take advantage of this fact rather than fighting it. The classic example is Motorola's satellite phone system that was made obsolete by cell phone technology before it was completed.

In designing the Boeing 747 airliner, the engineers designed weight growth capability into the airplane by giving it bigger wings and tail surfaces than it initially needed. The landing gear was more robust than needed and it was designed so that gear that is more capable could be substituted without the expense of a major redesign. This allowed a dozen different versions of the 747 to be built over 30 years (Sutter, 2006).

Build in preparation for buying: If a commercial off the shelf (COTS) product that suits your needs is available, then of course you should prefer to use it rather than develop a new product in-house. But if a COTS product is not available, then build a custom product, and monitor the market place: when a COTS product becomes available, switch to that COTS product. In hardware, sometimes the contractor will use commercial grade products while waiting for military standard parts to come along.

Create a new design process: When using a significant amount of COTS, products you must change your design process. These are some of the new activities that must be performed:

- Search the marketplace for candidate COTS products and technologies.
- Work with the customer in early project phases to refine the customer needs statement and get high-level, flexible objectives, instead of extensive requirements lists.
- Iteratively negotiate the requirements and the architecture with the customer.
- Change contracting so that funding for upgrades and marketplace research does not end with the project's period of performance.
- Create a satisficing design that may not meet all of the requirements, but one that provides the high priority capabilities and satisfies the customer's operational needs.
- Create test procedures that will be reused in the Operation and Maintenance phase of the system life cycle whenever upgrades or new COTS products are brought into the system.
- Plan for insertion of upgrades and technologies throughout the system life cycle.
- Continually search the marketplace for upgrades and new products that can replace existing COTS products, particularly in the Operation and Maintenance phase.

Choosing to design with a significant amount of COTS products is a decision that should not be made lightly. Indeed the BAE Systems' COTS-Based Engineering package already has a dozen documents containing three Megabytes of text.

Change the behavior of people: Finally, as a last resort, you can try to change bureaucracy and human nature so that you get stable requirements up-front. You can also try to change people to get rid of the *Not Invented Here* attitude. But good luck.

Systems of Complex Systems

More and more systems are now being composed of a multitude of complex systems. A system of systems is a large complex system that is composed of other large complex systems. (Systems composed of simple systems are not difficult to deal with and are not considered here.) First, the system of systems must be a system, meaning it must have states and inputs that are transformed by functions into outputs. Next, each of the constituent systems must be a system in its own right, meaning it must have states and inputs that are transformed by functions into outputs. The constituent systems must be able to function on their own as totally independent systems; therefore, all of the principles of good design that were presented in this article apply to the design of the constituent systems. Now, which of these principles applies to the high-level system of systems? Exhibit 1 shows the principles that might not.

The Department of Defense Architecture Framework (DoDAF) was created to help with the design of systems of systems. The DoDAF has the following four views of a system of systems:

Exhibit 1. Principles That Might Not Apply to the Design of Systems of Complex Systems

Principle	The reason why the principle might not apply
Use hierarchical, top-down design	Maybe system of systems design should be top-down. But presently most system of systems design is bottom-up, because usually the constituent systems (or at least their designs) already exist.
Write extension points Group data and behavior Use data hiding Envelope requirements Create design margins	System of systems design is not likely to get down to this level of detail. The design of the individual systems will get to this level, but the system of systems design will not.
Build in preparation for buying Change the behavior of people	This will not work for systems of systems.

- Operational View (OV): A description of the tasks, activities, operational elements, and information exchanges required to accomplish DoD missions
- Systems View (SV): A description of systems and interconnections supporting DoD functions
- Technical Standards View (TV): Standards and rules governing arrangement, interaction, and interdependence of system parts or elements, whose purpose is to ensure that a conformant system satisfies a specified set of requirements
- All View (AV): Information pertinent to the entire architecture, AV products set the scope and context of the architecture

The DoDAF, however, is not a design process. It is missing the following design artifacts: customer requirements, derived requirements, system test, system validation, concept exploration, tradeoff matrix, sensitivity analysis, schedule, budget, technical analysis, and risk management. Entities like these would have to be included in the design of the constituent systems, but it is possible that they could be omitted (as suggested by DoDAF) for the design of a system of systems.

Design for Reuse

We can design new systems better, faster, and cheaper if we correctly reuse previously developed products. (Of course, this only applies if you do the job correctly; i.e., you should not reuse computer code, you should reuse software models.) These products might have been developed in-house or they might be commercial off the shelf. There are several facets to design for reuse: (1) refining customer needs and negotiating system requirements with the customer, (2) selecting and evaluating potential reuse products, (3) technology planning and insertion, (4) product lifecycle planning, and (5) designing products so they are more reusable. This article concerns the 5th topic—design for reuse. Designing systems with increased probability of reuse is not free – it requires extra effort and cost. This produces another facet of the design for reuse process—convincing your boss or customer that it is worth the extra cost to design a system for increased probability of reuse. This facet is beyond the scope of this article.

Summary

This article has presented dozens of principles of good design. Of course they would not all be applicable on all programs. Surprisingly, none contradict each other. If care is not taken, the principle of "Develop iteratively and test immediately" could confound the principles of "Use hierarchal, top-down design" and "Work on high-risk items first." But in a careful design, the high-level or high-risk items would be decomposed into smaller modules that could be tested immediately. Similarly, "Design for testability" and "Design for evolvability" are not contradictory, they are orthogonal. There are no other obvious interactions between these principles.

Some of these principles are just good design principles—they will help with reuse, changing requirements, testability, and evolvability, whereas others primarily ameliorate changing requirements, increase reuse potential, or enhance evolvability. Exhibit 2 shows these principles and indicates for which facets they are most applicable; however, the Xs should not be crisp—they should be fuzzy. The right column indicates principles that are likely to impact the short-term cost of the system, but, of course, the tradeoff between short-term cost and customer value is a complex issue (Browning, 2003).

The principles of good design that were presented in this article were specific, concrete, low-level suggestions for design. We did not cover basic system engineering principles such as stating the problem, discovering requirements, doing tradeoff studies (Pugh, 1991; Daniels, Werner, and Bahill, 2001; Smith, Son, Piattelli-Palmarini, and Bahill, 2007), configuration management, sensitivity analyses (Smith, Szidarovszky, Karnavas and Bahill, 2008), etc. Of course, these other activities must be done, but they are covered adequately in other publications (INCOSE, 2004). Examples of failures that result from violating these system design principles are also given elsewhere (Bahill and Henderson, 2005).

Most engineers and managers are probably familiar with the principles of good design that were presented in this article. We think that presenting them all in one place can serve as a checklist, to make sure they are all considered in designing systems.

Acknowledgment

This article was supported by AFOSR/MURI F49620-03-1-0377. We thank Bob Sklar and Greg Shelton for comments on design margins.

References

Bahill, A. Terry, Rick Botta, and Jesse Daniels, "The Zachman Framework Populated with Baseball Models," *Journal of Enterprise Architecture*, *2*:4 (November 2006), pp. 50-68.

Exhibit 2. Facets for Which Each Design Principle is Most Appropriate

The uppercase X means the principle has a great affect, the lowercase x means the principle has a large affect, the dollar sign means using the principle will cost money.

Principle	Reuse Potential	Changing Requirements	Evolvability	Good Design	Impact Cost?
Use models to design systems	x	x		X	
Use hierarchical, top-down design				X	
Work on high-risk items first				X	
Prioritize		x		X	
Control the level of interacting entities				X	
Design the interfaces				X	
Produce satisficing designs				X	
Do not optimize early				X	
Maintain an updated model of the system				X	$
Develop stable intermediates		x	X	x	
Use evolutionary development			X		$
Understand your enterprise			X	x	$
State what, not how				X	
List functional requirements in the use cases		x		X	$
Allocate each function to only one component		x		X	
Do not allow undocumented functions	x			X	
Provide observable states	x		x	X	$
Rapid prototyping		x		X	$
Develop iteratively and test immediately		x		X	
Create modules	X			X	
Create libraries of reusable objects	X				$
Use open standards	X		x		
Identify things that are likely to change		x	X		
Write extension points		x		X	
Group data and behavior				X	
Use data hiding				X	
Write a glossary of relevant terms	x			X	$
Envelope requirements		X			
Create design margins		X			$
Design for testability				X	$
Design for evolvability			X		$
Build in preparation for buying	X				$
Create a new design process	X				$
Change the behavior of people		X			$

Bahill, A. Terry, and Steven J. Henderson, "Requirements Development, Verification, and Validation Exhibited in Famous Failures," *Systems Engineering*, *8*:1 (2005), pp. 1-14.

Bahill, A. Terry, and Ferenc Szidarovszky, "Comparison of Dynamic System Modeling Methods," *Systems Engineering* DOI 10.1002/sys20118, published online (October 3, 2008).

Bahill, A. Terry, Ferenc Szidarovszky, Rick Botta, and Eric Smith, "Valid models require defined levels," International Journal of General Systems, published electronically, iFirst 37: (March 2008), pp. 1-20, DOI: 10.1080/03081070701395807, published on paper 37:5 (October 2008), pp. 553-571.

Boehm, Barry W., "A Spiral Model of Software Development and Enhancement," *Computer*, *21*:5 (1988), pp. 61-72.

Botta, Rick, and A. Terry Bahill, "A Prioritization Process," *Engineering Management Journal*, *19*:4 (2007), pp. 20-27.

Botta, Rick, Zach Bahill, and A. Terry Bahill, "When are Observable States Necessary?" *Systems Engineering, 9*:3 (2006), pp. 228-240.

Botta, Rick, William Wuersch, and A. Terry Bahill, "The Need for Observable States Affects the Make-Reuse-Buy Decision," *Proceedings of the 14th Annual International Symposium of the International Council on Systems Engineering (INCOSE),* (June 2004), \content\papers\533.pdf.

Browning, Tyson R., "Process Integration Using the Design Structure Matrix," *Systems Engineering*, *5*:3 (2002), pp. 180-193.

Browning, Tyson R., "On Customer Value and Improvement in Product Development Processes," *Systems Engineering*, *6*:1 (2003), pp. 49-61.

Chapman, William L., A. Terry Bahill, and A. Wayne Wymore, *Engineering Modeling and Design*, CRC Press (1992).

Christian, John A., and John R. Olds, "A Quantitative Methodology for Identifying Evolvable Space Systems," *1st Space Exploration Conference*, (January 2005) AIAA 2005-2543.

Chrissis, Mary Beth, Mike Konrad and Sandy Shrum, *CMMI: Guidelines for Process Integration and Product Improvement,* Pearson Education Inc. (2003).

Cockburn, Alistair, *Writing Effective Use Cases*, Addison-Wesley (2001).

Daniels Jesse, and A. Terry Bahill, "The Hybrid Process That Combines Traditional Requirements and Use Cases," *Systems Engineering*, *7*:4 (2004), pp. 303-319.

Daniels Jesse, Paul W. Werner, and A. Terry Bahill, "Quantitative Methods for Tradeoff Analyses," *Systems Engineering*, *4*:3 (2001), pp. 199-212.

The Defense Acquisition System, DoD Directive, Number 5000.1, May 12, 2003. http://www.dtic.mil/whs/directives/corres/pdf2/d50001p.pdf

Douglas, Bruce P., *Real-time UML: Advances in the UML for Real-time Systems,* 3rd Edition, Addison-Wesley (2004).

Evans, Eric, *Domain-driven Design: Tackling Complexity in the Heart of Software,* Addison-Wesley (2004).

Fowler, Martin, *UML Distilled: A Brief Guide to the Standard Object Modeling Language,* 3rd Edition, Addison-Wesley (2004).

Fricke, Ernst, and Armin P. Schultz, "Design For Changeability (DfC): Principles to Enable Changes in Systems Throughout Their Entire Lifecycle," *Systems Engineering, 8*:4 (2005), pp. 342-359.

Gomaa, Hassan, *Designing Concurrent, Distributed, and Real-time Applications with UML,* Addison-Wesley (2000).

INCOSE Systems Engineering Handbook v2a, 2004. Retrieved March 2006, http://www.incose.org/ProductsPubs/incosestore.aspx.

Jacobson, Ivar, Grady Booch, and James Rumbaugh, *The Unified Software Development Process,* Addison-Wesley (1999).

Kulak, Daryl, and Eammon Guiney, *Use Cases: Requirements in Context*, Addison-Wesley (2000).

Moody, Jay A., William L. Chapman, F. David Van Voorhees, and A. Terry Bahill, *Metrics and Case Studies for Evaluating Engineering Designs*, Prentice Hall PTR (1997).

Rosenblit, Jerzy W., *A Conceptual Basis for Model-Based System Design*, Ph.D. dissertation in Computer Science at Wayne State University, published by University Microfilms International (1985).

O'Connor, Patrick D. T., *Test Engineering*, John Wiley (2001).

Övergaard, Gunnar, and Karin Palmkvist, *Use Cases: Patterns and Blueprints,* Addison-Wesley (2005).

Parnas, David L., "On the Criteria to be Used in Decomposing Systems Into Modules," *Communications of the ACM*, *15*:12 (1972), pp. 1053-1058. Also available at http://www.acm.org/classics/may96/

Pugh, Stuart, *Total Design*, Addison-Wesley (1991).

Quintanar, Gerard J., "An Age of Interfaces," *Distributed Computing*, (November 1999), pp. 15-18.

Rechtin, Eberhardt, *Systems Architecting of Organizations: Why Eagles Can't Swim*, CRC Press (2000).

Rumbaugh, James, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, 2nd Edition, Addison-Wesley (2005).

Schultz, Armin P., Ernst Fricke, and Eduard Igenbergs, "Enabling Changes in Systems Throughout the Entire Life-Cycle - Key To Success?" *Proceedings of the 10th Annual International Symposium of the International Council on Systems Engineering (INCOSE),* (July 2000), pp. 599-607.

Simon, Herbert A., *Models of Man: Social and Rational,* Wiley (1957).

Smith, Eric D., Ferenc Szidarovszky, William J. Karnavas and A. Terry Bahill, "Sensitivity analysis, a powerful system validation technique," The Open Cybernetics and Systemics Journal, http://www.bentham.org/open/tocsj/openaccess2.htm, doi: 10.2174/1874110X00802010039, 2: (2008), pp. 39-56.

Simon, Herbert A., "The Architecture of Complexity," *Proceedings of the American Philosophical Society*, *106* (1962), pp. 467-482.

Smith, Eric D., Young-Jun Son, Massimo Piattelli-Palmarini, and A. Terry Bahill, "Ameliorating Mental Mistakes in Tradeoff Studies," *Systems Engineering*, *10*:2 (2007), pp. 222-240.

Suh, Nam P., *The Principles of Design*, Oxford University Press (1990).

Sutter, Joe, with Jay Spenser, *747: Creating the World's First Jumbo Jet and Other Adventures from a Life in Aviation*, Harper Collins Publishers (2006).

Wymore, A. Wayne, *Model-based Systems Engineering*, CRC Press (1993).

Wymore, A. Wayne, "The Nature of Research in Systems Engineering," paper presented at the Stevens Institute -- University of Southern California Workshop on Research in Systems Engineering, (April 2004).

Wymore, A. Wayne, and A. Terry Bahill, "When Can We Safely Reuse Systems, Upgrade Systems or Use COTS Components?" *Systems Engineering*, *3*:2 (2000), pp. 82-95.

About the Authors

Terry Bahill, PE, is a professor of systems engineering at the University of Arizona in Tucson. While on sabbatical from the University of Arizona, he did research with BAE Systems in San Diego. He received his PhD in electrical engineering and computer science from the University of California, Berkeley in 1975. Bahill has worked with BAE Systems in San Diego, Hughes Missile Systems in Tucson, Sandia Laboratories in Albuquerque, Lockheed Martin Tactical Defense Systems in Eagan MN, Boeing Information, Space and Defense Systems in Kent, WA, Idaho National Engineering and Environmental Laboratory in Idaho Falls, and Raytheon Missile Systems in Tucson. For these companies he presented seminars on systems engineering, worked on system development teams, and helped them describe their systems engineering processes. He holds a U.S. patent for the Bat Chooser, a system that computes the Ideal Bat Weight for individual baseball and softball batters. He

received the Sandia National Laboratories Gold President's Quality Award. He is the Founding Chair Emeritus of the INCOSE Fellows Selection Committee.

Rick Botta is the director of systems engineering for BAE Systems in San Diego. He holds a Bachelor of Science degree in Computer Science from California Polytechnic State University, San Luis Obispo. Rick has a quarter century experience in a wide variety of engineering, engineering management, and program management roles involving development and integration of complex, software intensive systems.

Contact: A. Terry Bahill, PE, Systems and Industrial Engineering, University of Arizona, Tucson, AZ, 85721-0020; phone: 520-621-6561, terry@sie.arizona.edu

Appendix A
Principles of Function Design

Functional decomposition is a popular system design technique. When using it, you identify the top-level function that the system must perform. Decompose that function into subfunctions. Then decompose those subfunctions into sub-subfunctions. Continue this decomposition until you get functions that can be implemented with commercial off-the shelf products. When performing this design task, creating the functions and allocating them to components should not be arbitrary or capricious. The functions should be designed and there are principles that can help you design good functions.

Allocate Each Function to Only One Component. Each function should be allocated to only one physical component, and, therefore, each function will have only one owner. If there were two owners for a function, one might change his or her requirements, and this would change the system for the other. In the object-oriented world, this would be phrased as, "Do not allow multiple actors to have the same role" (Övergaard and Palmkvist, 2005). If two actors are trying to assume the same role, generalize them into one abstract actor. My wife notes that in a theater play it would also be mistake to allow two actors to assume the same role. For some systems and customers, this principle may be difficult to implement. Let us now look at some examples of following and violating this principle.

One function allocated to two components, bad design: Violation of this principle is captured in the American proverb, "Too many cooks spoil the soup." In the 20th century, the function of tracking terrorists was allocated to the FBI, the CIA, the NSA, the Pentagon, etc. Until 9/11/01, the results were mixed. In a baseball game, if the runner on first base is likely to steal second base, then the function of covering second is allocated by agreement to the second baseman or the short stop, but never to both. This agreement prevents bad design. This principle may be the reason back-seat drivers are so distained.

One function allocated to two components—exceptions that proof the rule: One valid reason for assigning a function to more than one component would be that the function is performed by one component in a certain mode and by another component in another mode. For example, to decelerate an airplane, when the wheels first hit the runway, reverse the thrusters on the engines. Later, when preparing to move from the runway to the taxiway, apply the brakes on the wheels. Another reason for assigning a function to more than one component would be deliberate redundancy to enhance reliability—allowing one portion of the system to take on a function if another portion fails to do so, as on the Space Shuttles, which have five flight control computers. Another reason is truly distributed functionality: you can decelerate a car with either front wheels or back wheels. For decelerating an airplane, the reverse thrusters on the engines and the air brakes on the wings might be for redundancy or distributed functionality.

One system with two subsystems and two functions, OK design: A claw hammer pounds nails or pulls nails. A pencil makes lines or erases lines. A teapot heats water and whistles. A clock radio wakes up people or plays music. A heating, ventilation, and air conditioning system heats a house or air-conditions a house. Computers do many things at the same time.

One object with two functions, OK design: A drawbridge allows cars to move over the river or boats to move under the bridge. Female breasts attract males and feed offspring resulting from that. Mammal sex organs are used for reproduction and waste elimination. An air conditioner cools air and removes humidity from the air. A dog can guard the house or be a pet.

One object with two functions, marginal design: Sports stadiums have been built to present both football and baseball games. Such stadiums are not popular now—they must not have been very successful. A combination washer and dryer washes clothes and dries clothes (these are not currently popular). An item with functions of create and operate the same object is a bad design (Evans, 2004). If you buy a new car and it needs service, you do not take it back to the factory that manufactured it—you take it to a separate service center.

One function with two purposes, OK design: An oven cooks food and kills bacteria. It has one function (heat food) but two purposes. A dishwasher cleans dishes and kills bacteria.

Make Functions Independent. Functions should not depend upon each other. If they are independent and one is changed, then the other one will not have to be changed. Normally I let my clock radio wake me up with music. But if I am in a strange town, I use the buzzer, just in case the radio station is off the air or the radio is mistuned when I want to get up. Allowing the wakeup function to depend on the play music function is dangerous, and I avoid it when the wakeup function is critical. Fricke and Schultz (2005) present a technique to help keep functions independent,

A bad design: On most company telephones, dialing 9 gets an outside line, and then dialing 1 starts a long distance dial. At this point if the 1 button is inadvertently hit again, then the caller is connected to 911, the emergency line. If you hang up quickly, they send a policeman to save you.

Limit the side effects: Functions should do what their names imply and nothing else. For example, in an automobile, the ignition switch turns off the ignition system, but it also has a side effect of disconnecting electric power to the radio. Renaming this switch the electric power switch would solve this side effects problem.

These principles apply to *object-oriented design* as well. In use cases, the functions are described in the Functional Requirements part of the Specific Requirements section (Daniels and Bahill, 2004). In use case diagrams, functions are associated with the roles. In activity diagrams, functions are the activities. In state machine diagrams, functions are the actions and activities. In class diagrams, functions are the operations listed in the third part of the box. In sequence diagrams, functions are the commands, so functions appear in many places.

In an object-oriented design, it would be a mistake to give two functions the same name and it would be a mistake to assign one function to two classes; however, it is very natural to have one function appear in a use case, a class diagram, a state machine diagram, and a sequence diagram. It would be specified in one place and used in multiple diagrams.

These principles of good design do not apply to biological systems that evolved instead of being designed.

Appendix B
Design Margins

The term *design margin* is used for at least four different purposes: safety factors, budget reserves, tolerances, and performance capabilities. When designing a system, parameters with large uncertainties should have bigger safety factors than parameters that are well known. For instance, if it is believed that the safe upper limit for voltage is 200 ± 5V, then rate the system at 190V. Whereas, if you think a safe upper limit for voltage is 200 ±20V, then you could rate the system at 140V.

The maximum safe pressure for a cylindrical steel boiler is given by

$$\text{Maximum Pressure} = \frac{\text{Wall Thickness} \times \text{Tensile Strength}}{\text{Shell Radius}}$$

Suppose that the values for a particular boiler design yield a maximum safe pressure of 1000 psi. According to the 1915 ASME Boiler Code, you should rate it for 200 psi: that is, the code required a safety factor (design margin) of 5. Studies of the service history of pressure vessels and failure analyses led to a more thorough understanding of the behavior of steel materials and better manufacturing techniques led to better materials. As a result, in 1951 the ASME design margin was changed to 4, which would allow 250 psi for the boiler above. In 1999, this design margin was further reduced to 3.5. As the system becomes better understood, the design margin is reduced.

Reliability engineers increase the reliability of a system by limiting the ratio of the operating stress to the rated stress to, for example, 50%, which implements a 100% design margin. For example, a capacitor that is rated for 10V would be restricted to a maximum operating voltage of 5V.

Often nonfinancial budgets are created for critical design parameters.

Exhibit B1. Weight Budgets

Module	Target Weight	Maximum Allowable Weight	Reserve
A	15	20	5
B	20	30	10
C	40	50	10

But it is a better practice (particularly with schedule) to sweep up all of the reserve and give it to the Program Manager.

Exhibit B2. Allocation of Reserves

Module	Target Weight
A	15
B	20
C	40
Program Manager's Reserve	25

Things that are often budgeted include computer memory, weight, power, cost, and schedule. These items are usually traded off with each other and also between the subsystems.

When designing a system, it is important to give tolerances for manufacturing parameters. These design margins are usually given as the tolerance about a nominal value. For example,

Output voltage	5 V ± 0.1
Weight	5 kg +0.1, -0.5
Hole diameter	5 cm +0.1, -0

In a similar vein, parameters will change due to operation, wear, aging, and replacement. Tolerances must be given for maintenance procedures. For example, when tuning a Datsun 240Z, set the spark plug gap between 0.8 and 0.9 mm and set the distributor point gap between 0.45 and 0.55 mm.

At Sandia Laboratories a "high-margin" nuclear explosive package design generates its nominal (or higher) yield over the widest possible range of input and environmental parameters. New nuclear weapons are being designed with increased "design margins," which some people think will decrease the "safety margin."

Other considerations: assume a design has five layers of integration. How we allocate tolerances can have a huge impact on yield at each level. If we assume that statistically we need to constrain the lower level components with a 6-sigma margin to allow for a 4-sigma yield as we integrate all the pieces together, then we will have a high cost at those lower levels but a lower cost at the top level. When considering field service, the low-level field replacements must work with a system that has been in the field for a considerable period of time; therefore, the design margins and tolerances must be controlled precisely at the low levels to allow for these field replacements to have a high confidence of success.