Proceedings of ICAD2004
The Third International Conference on Axiomatic Design
Seoul – June 21-24, 2004
ICAD-2004-13

SOFTWARE PRODUCT LIFECYCLE MANAGEMENT USING AXIOMATIC APPROACH

Sung-Hee Do
dosh@axiod.com
Axiomatic Design Solutions, Inc.
221 N. Beacon St., Boston, MA 02135, U.S.A.

ABSTRACT

Software products are managed by several activities over their lifecycle. Requirements, Development, QA and Documentation are common categories describing these activities. Although the lifecycle is often referred to in linear terms, the reality is an iterative dynamic between lifecycle activities. A change in one activity can propagate indefinitely over the product lifecycle. One major reason for the frequent failure of software projects is the failure to deal with changes to requirements. Requirements seldom remain constant throughout the product lifecycle and must therefore be systematically managed to avoid introducing chaos into the design process.

The axiomatic framework has been introduced to manage the design process in a systematic manner. It starts with requirements capture and continues the process by establishing linkages between domains (i.e. activities) over the design hierarchy. This requirements driven approach provides the insight to objectively gauge the impacts of changing requirements. The highly dynamic nature of software development provides an ideal demonstration of the framework's capabilities.

This paper explains how to manage the software lifecycle in conjunction with the Axiomatic Design framework. This includes capturing requirements, developing the software architecture in a solution domain, establishing development and QA processes, evaluating the design by matrix analysis from developed code, and producing documentation from requirements. The paper also details a simulation of how requirement changes impact the whole system, and how such changes are manageable within the axiomatic framework.

Keywords: Axiomatic Design, traceability, unified modeling language (UML), product lifecycle management (PLM), design matrix (DM), design structure matrix (DSM)

1 INTRODUCTION

The software industry is a fairly young industry compared to traditional design and manufacturing industries. Compared to other industries, the relative lack of costs for equipment, retooling, materials procurement, and packaging for example, changing manufacturing lines and their related fixtures due to design changes never applies for software products. The reduction in traditional barriers to entry creates a highly competitive market in which software products are under constant pressure to integrate new requirements or change existing ones. The total lifecycle for software products is almost universally shorter than that of the manufacturing industry.

Due to the relative lack of costs for traditional cost drivers for other manufacturing, software is particularly sensitive to product development costs. Often, these costs escalate (and are regularly deemed "out of control" on large projects) in response to the inevitable reality of changing requirements. As a result, systematic management is a key element in making software products successful.

One of the major barriers to systematic management in the software industry is the lack of traceability. A purely iterative process for satisfying changing requirements is not sustainable over time. In the manufacturing industry for example, the design changes can be visualized throughout the process by using modeling tools such as CAD, FEM analysis tools, etc. so that the possibility of detecting failures is high. Managers in the software industry, on the other hand, don't have comparable tools. Typically software development teams tend to leverage tools to manage and debug their code, but this is a low level task which doesn't address the fundamental issues related to evolving requirements. Other tools intended to fill this role, often fall subject to misuse and ultimately irrelevance. UML tools, for example, are often used only for reporting or documentation, rather than managing the lifecycle as intended. This is one of the reasons why most software design projects are not able to meet the given specifications or to deliver on time and on budget.

Over the past twenty years, Axiomatic Design has been applied in many different areas including mechanical engineering, material science, software, organizational design and so on [Suh (1990, 2001)]. One of the practical benefits of employing Axiomatic Design in a commercial environment is the generation of end to end traceability. This value proposition has only recently been exposed due to the availability of software to manage the decomposition software [ADSI].

Traceability can be performed using matrix analysis. Two types of matrix analysis are available. Axiomatic Design uses a design matrix (DM) to analyze adjacent domains (e.g. the functional and physical domains.) Domain-to-domain matrix analysis makes it possible to trace backwards and forwards

Copyright © 2004 by ICAD2004

through the entire design. A second type of matrix analysis – design structure matrix (DSM) – analyzes only a single domain [Ulrich, et al. (2003)]. One of the benefits of DSM is clustering, a technique which provides the capability to group given tasks and/or components based on their inter-relationships [Pimmler, et al. (1994)]. Through the use of these matrix driven traceability concepts, systematic management of the software lifecycle is made possible and overcomes the problems of rapidly changing requirements, as will be detailed in this paper.

2 EXTENDING THE AXIOMATIC PROCESS

Axiomatic Design (AD) proposes the use of four domains as shown in Figure 1. Design decomposition spans the functional, physical and process domains. As the hierarchies in each of these domains converge on an identical structure, these domains be called symmetric domains in this paper. In addition, Axiomatic Design defines a customer domain which captures external requirements (such as marketing requirements, safety regulations, etc.) Traditional Axiomatic Design was done mostly in mechanical engineering, where the customer needs are usually clear and seldom change. The static method of requirements handling of the traditional axiomatic approach doesn't support the dynamic nature of changing requirements very well, since every requirement change forces the design team to revisit the design decomposition and amend it appropriately. A key goal is to support frequent requirement changes by reducing the overhead involved in re-establishing the design decomposition hierarchies.

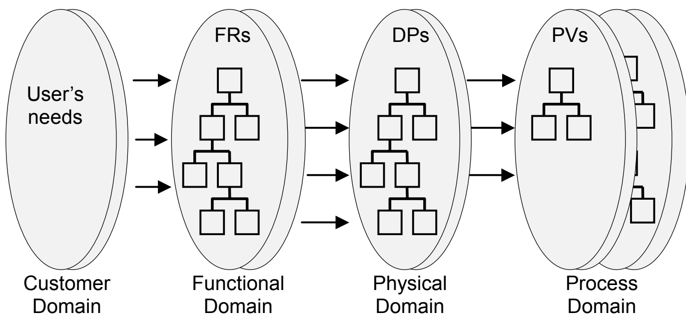


Figure 1. Concept of domains and mapping.

Another lesson learned from recent design activities using the axiomatic approach is that it takes a lot of time to decompose to the levels at which most design engineers feel comfortably in their territory of expertise. Starting from the zero level and decomposing to the conceptual level is difficult. If design engineers are trying to create a new "clean sheet" design in a solution neutral environment, the conceptual level design decomposition is very valuable. However, if they are trying to improve an existing design, decomposing the whole design from the very top level doesn't always provide value and becomes somewhat painful before they really begin to reap the rewards. In the re-design case, they need to have fast approach.

Recent design activities have demonstrated that the process of reverse engineering the design decomposition from a pre-existing design is often perceived to be more trouble than it's worth. Absent the ability to effect meaningful change to the fundamental design of the product (due to the investment in CAD drawings, customer trial prototypes, tooling, etc.), the process of obtaining the full FR/DP decomposition after the fact can be seen as more trouble than it's worth. In these instances a rapid approach to design capture for the purposes of improving some aspect of the design is critical.

Figure 2 is a modified control block diagram that defines an alternative decomposition process approach to resolve difficulties mentioned above. The detailed explanations for the Figure 2 are described below:

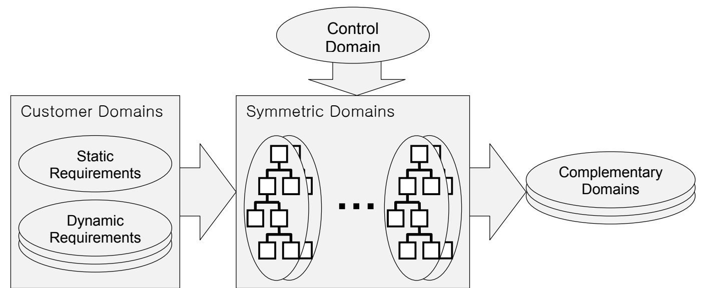


Figure 2. Extended domain concept.

Symmetric Domains: Design decomposition is a core activity for the proposed process. It is captured in symmetric domains (FR, DP and PV) to satisfy the independence axiom of Axiomatic Design which claims there should be a one-to-one mapping between domains. These symmetric domains may maintain several decomposition islands in order to avoid difficulties with top-down decomposition of existing systems. Each individual island is responsible for subsystems where design engineers begin decomposing their subsystem without worrying about the conceptual level organization. Managers may arrange these subsystems under zero level of their product decomposition or develop a conceptual level decomposition structure to gain a better system design understanding. Both of these approaches are valid since inter-relationships (dependencies) will be revealed when the full design matrix is completed. Since the full design matrix shows all of the leaf level relationships, by definition, it will show interactions between subsystems.

Customer domain: For practical reasons it is often helpful to consider the traditional customer domain as an abstract entity under which a variety of concrete domains can exist. Industry standards and government security requirements represent static requirements that must be satisfied over the product lifecycle. Functional flows from systems engineering and use cases in software systems are candidates for inputs to a category of requirements which represents the dynamic nature of requirement changes. Dynamic requirements have more impact on the symmetric domains since they change over the product lifecycle. DSM clustering analysis is useful for this dynamic requirement domain to categorize subsystem requirements. If such a proposed categorization is carried out from given requirements, efficient and logical structuring for individual subsystem islands is possible in the symmetric domains.

Control domain: The control domain is a new addition to the concept represented by the diagram. Constraints are the best

Copyright © 2004 by ICAD2004

example of this type of domain, since they impact overall product performance.

Complementary domains: Complementary domains are also added to address additional needs specific to the industry or problem domain. In the manufacturing industry for example, this domain should capture the data - that are controlled by product data management (PDM) systems, such as product structures, cost units, CAD models, bill of materials (BOM) and other manufacturing related items. In the software industry, example domains designed to snapshot and check the development status along with the design are Quality Assurance (QA), testing, risk management, and source code reverse engineering.

Linking among domains: The correct establishment of the relationships between these domains is the key requirement for running the proposed traceability model successfully. As long as the correct relationships between the objects in the domains are preserved, managers are able to review the genesis and consequence of design artifacts over time.

3 SOFTWARE PRODUCT LIFECYCLE MANAGEMENT

Software is not manufactured in the classical sense. Instead, the engineering function serves the role of the manufacturing process. Also, software performance doesn't degrade over time. Given these characteristics, the principle performance concern is related to software failure – commonly referred to as "bugs". Figure 3 demonstrates how software failure profile responds in response to the introduction of changes to the system over time. Each change fundamentally degrades the failure profile for the system [Pressman, (1997)]. Many software process models have been introduced in an attempt to address this problem. They include the waterfall model, the rapid application development (RAD) model, the spiral model, the rational unified process (RUP) [Kroll et al. 2003] and so on. All of these process models are converging on an iterative approach due to the fundamental realities of significant requirements change. However, most of these processes tend to handle each iteration step as a discrete event that often becomes an island removed from the core process of the system. Unlike manufacturing, one last critical difficulty is the difficulty of visualizing the part (in this case code) that is being actively developed. The axiomatic approach offers matrix manipulations as a visualization method. It thus provides a whole new way of managing the software lifecycle.

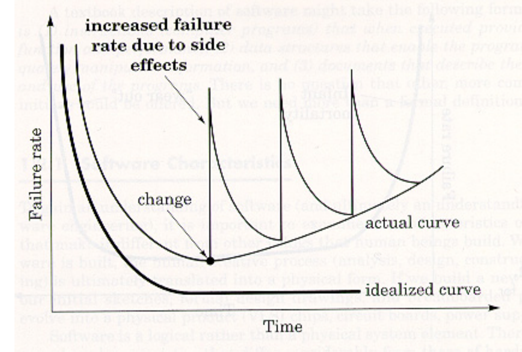


Figure 3. Failure curve for software.

Using matrix traceability, the axiomatic process model recommends cycling through each step iteratively. Each new requirement should be processed via the flow in Figure 2, from the customer domain, to the symmetric domains and the complementary domains. An example of this process specific to software development follows. This process focuses on the object-oriented development paradigm. However, it could be adapted to other software methodologies as well.

Requirement handling in the customer domain

The object oriented approach to software engineering is widely accepted and practiced within the software industry. In the object oriented approach, each subject or task is modeled as an object. An object encapsulates and exposes its behavior to other objects via the use of methods. These methods, in turn, utilize data maintained by the object itself. Ideally, the only direct access to the data within the object is via the object's methods. This approach, in theory, allows strong separation between the behavior of the object and the internal details necessary for clients of the object to leverage its behavior.

At a process level, use cases are a common means of defining and managing requirements for a system. Use cases are often articulated as a textual decomposition of processes the system must support, and can in turn be mapped to use-case or sequence diagrams represented in UML that capture the object interactions required to support each use-case.

The axiomatic process model recommends establishing a use case domain in order to model the dynamic nature of requirements in software. Every customer requirement should be mapped into the use case domain and scenario and decomposed in accordance with the use case's process steps. In this model, customer requirements drive use cases that, in turn, drive functional requirements and other artifacts of symmetric decomposition. While it is possible to drive from customer requirements directly to functional requirements (as defined in a core Axiomatic Design process), using a use case centric decomposition process allows a clear separation between the underlying functions of the system (FRs) and the processes which leverage those functions (as captured in the form of use cases). This makes for a more consistent and easier to implement decomposition.

Once use-cases are defined, the next recommended step is to utilize DSM clustering techniques to identify highly interdependent aspects of the system. In this technique, highly

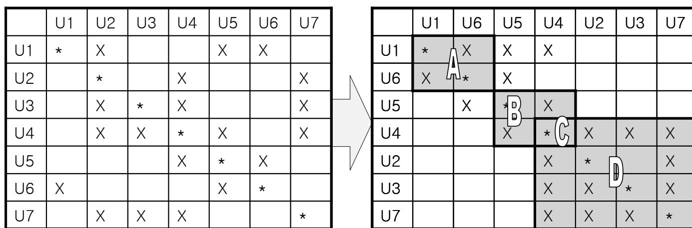
Copyright © 2004 by ICAD2004

interdependent aspects are grouped into architectural chunks. Because of the interdependent nature of these chunks, DSM prescribes that the best approach to organize responsibility for the handling of the various aspects that are contained in a given chunk, is to consider, manage and define them along with all other aspects of the same chunk.

In order to apply this technique in this process, use cases must be elaborated in order to identify interrelationships between them. Once accomplished, the use-cases can then be grouped into intrinsic sub-system entities which can identify optimal group organization to proceed with transforming the use-cases into the functional and physical domains. Figure 4 illustrates this process. A, B, C and D in the figure are candidates sub systems.

Constraints in control domain

Constraints identify system wide performance limitations on the system. Depending on the functional and physical decomposition of the system, these constraints may be satisfied passively, by never violating the constraint in the solution, or actively, requiring the creation of new FRs or alternative DP selections to keep the system from violating the constraint. Concrete artifacts of constraints in software systems are robustness and conditional handling, etc. Explicitly identifying these exceptional conditions in the constraints allows developers to be more productive and prescribe responses at design time, rather than after the fact.



	U1	U2	U3	U4	U5	U6	U7
U1	*	X			X	X	
U2		*		X			X
U3		X	*	X			X
U4		X	X	*	X		X
U5				X	*	X	
U6	X				X	*	
U7		X	X	X			*

	U1	U6	U5	U4	U2	U3	U7
U1	*	X	X	X			
U6	X	*	X				
U5		X	*	X			
U4			X	*	X	X	X
U2				X	*		X
U3				X	X	*	X
U7				X	X	X	*

Figure 4. Control dynamic requirements.

Design decomposition in the symmetric domains

One of the most important and therefore difficult aspects of functional decomposition is defining the right high level FRs. When high-level FRs are defined well, the decomposition process is straightforward. However, if the high-level FRs are poorly conceived, the rest of the decomposition process will suffer as well.

The utilization of a use case domain in the process described here helps assure the correct high-level FRs, by pre-defining sub-systems and process steps and reducing the scope of the functional decomposition problem. By reducing the scope of work in the functional domain, designers can more easily decompose each subsystem to perform and create object hierarchies as design parameters (DPs) that satisfy corresponding FRs with the zigzagging process between domains.

To allow for evolution of the decomposition over time, the proposed process recommends decomposing each subsystem down to a class level of granularity. This should result in roughly four to five levels of decomposition per subsystem.

While it's possible that designers would choose to decompose beyond the class level all the way down to the method and attribute level (the software equivalents of "nuts and bolts"), this is not necessarily recommended. Decomposition down to too low a level has the potential to balloon into an incredibly large and difficult-to-manage representation of the system. Given that in the software space, the source code for the system not only implements the behavior of the system, but also documents that behavior (albeit in an unconventional form), this level of detail in the decomposition is unnecessary. Another option for capturing this level of detail will be proposed in an optional complementary domain later in this document.

The decision to drive the decomposition to a class level of granularity parallels object oriented development's separation between the object's external behavior (which defines its functions relative to the system) and its internal behavior (which defines its functions relative to itself).

Once the required subsystems are constructed as part of each iteration, the full design matrix should be checked to see that the proposed design is feasible without any coupled interactions. At this point, it is important to identify relationships between the customer domains (the use case domain in this case) and the functional domain such that each process step in every use case should be satisfied by at least one functional requirement. At this point, programmers can start the implementation of classes leveraging the guidance provided by interactions identified in the full design matrix.

Verifying iterations using complementary domains.

The complementary domains have been introduced to verify and bridge between requirements, conceptual design and physical structure. Additional complementary domains can be defined based on the applicable industry or nature of the problem. The following discussion identifies complementary domains particular to software:

Product structure domain: The software class hierarchy is a good example of a product structure. While DPs could in theory represent an appropriate class hierarchy, conceptual DPs, as part of their symmetric nature, are typically organized functionally. The product structure domain in turn maps the functionally organized DPs to a physically organized decomposition. This mapping is common in a variety of problem contexts. For example, a DP may be decomposed to handle the inputs to a circuit and elsewhere another DP may be decomposed to handle the outputs. In the product structure each of these DPs would be mapped to the same physical circuit, a part which implements both the inputs and the outputs.

In software, the product structure domain can be used to verify code by automatically deriving the structure from the existing code base and examining the relationships of the derived domain against the DPs in the system. The same could also be done with mechanical modeling tools or other tools which maintain structurally-oriented product data.

This mapping allows project teams to verify that the set of conceptual DPs are appropriately allocated to physical parts. Moreover, the product structure itself can be further analyzed via DSM techniques for additional optimization of the structure and maintenance of the system. In theory, clustering analysis of both the use case domain and the physical structure domain should produce highly similar subsystem clusters. Testing this hypothesis is outside the scope of this document and remains to be proven.

Copyright © 2004 by ICAD2004

Product activity domain: Most software today is "event" driven. In an event driven system behavior is initiated in response to the receipt of an event. Conceptually, this is the equivalent of the mail man ringing your doorbell to alert you that your mail is present, rather than requiring you to continuously check your mailbox to test whether mail has arrived. In software, events take the form of mouse clicks, file system alerts, and other notifications which alert the software that a response may be required (at the software's discretion). These activity events represent the dynamic nature of the software and serve as starting points for interactions between objects. Software failures can occur if the software does not respond appropriately to an event based on its internal state which is rarely deterministically defined over time. From a management perspective, these activities should be categorized by a set of process iteration steps which are identified in the use case domain for every development cycle. Accumulating these activities represents the actual status of software project schedule.

Test case domain: Use cases are the primary source of test cases. Every relationship between the use case domain and the functional domain is subject to be a single test case. These test cases should also be mindful of testing the limitations of the system as defined in the constraints. Additional tests can be listed in this domain. Maintaining tests this way allows teams to visualize the impact (in terms of system behavior) of the test failures. Because of these linkages, managers can travel among use cases, FRs, DPs, product structures and product activities to figure out whether each customer requirement can be satisfied. Another benefit of the links is ability to generate test case scenarios for each iteration or build cycle and, if the test case fails, use the results as inputs to the quality assurance domain.

Quality assurance (QA or Risk Assessment) domain: Every software organization has a QA division to quantify failures in the software. Most of the QA team maintains a bug reporting system as part of the QA process of keeping track of the lifecycle of the failures. The QA domain records the failures coordinated by elements of the case domain and categorizes them by creating hierarchies depending on the origin of the failure. Failures can be ranked by risk strength. Probability of occurrence and potential severity of consequence for each failure item could be captured from test case execution. These particular custom attributes then are used to rank risks. Figure 5 shows a graphical depiction of risk ranks for a series of failures. In this representation, failures appearing in the upper right portion of the graph are considered most critical [Haimes, 1998]. By maintaining traceability throughout the various domains, managers can trace high risk failures back through the test case domain, to the product activity domain to the product structure domain, to the FR/DP domain, to the use case, and ultimately to the customer requirement that is in jeopardy.

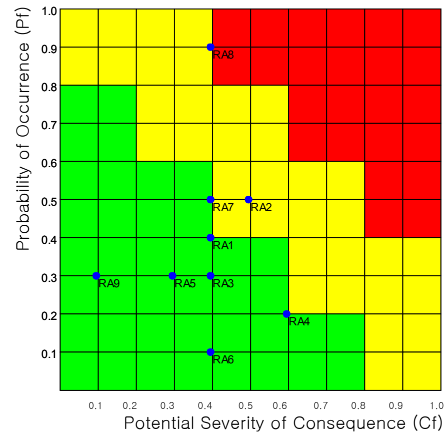
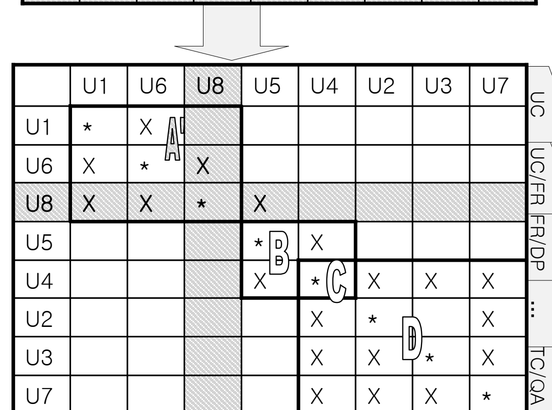


Figure 5. Example of risk profiles.

4 VISUALIZING THE AXIOMATIC ITERATIVE PROCESS MODEL

Suppose that a design group has executed the initial iteration that contains every element in Section 3 starting from requirements and culminating in risk analysis. As requirements are added to the process, the next process iteration unfolds. First, the new requirements should be translated into use cases. Next, the DSM analysis with a newly added use results in impact change analysis. Two modes for impact change assessment are shown in Figure 6, which extends from Figure 4 and indicates the "U8" as change: one is a matrix and the other is a diagram that shows chain links across each domain. This visual representation is consistent with the past decisions that the new change should consider during the lifecycle of the iteration and provides a high degree of implementation reliability.

	U1	U2	U3	U4	U5	U6	U7	U8
U1	*	X			X	X		
U2		*		X			X	
U3		X	*	X			X	
U4		X	X	*	X		X	
U5				X	*	X		
U6	X				X	*		X
U7		X	X	X			*	
U8	X				X	X		*



	U1	U6	U8	U5	U4	U2	U3	U7
U1	*	X						
U6	X	*	X					
U8	X	X	*	X				
U5				*	X			
U4				X	*	X	X	X
U2					X	*		X
U3					X	X	*	X
U7					X	X	X	*

Copyright © 2004 by ICAD2004

Figure 6. Process visualization.

5 DISCUSSION

Product lifecycle management requires coordination among a variety of different disciplines and organizational responsibilities. By adopting a systematic approach, many of the pitfalls inherent in this type of coordination can be avoided. The axiomatic design based framework illustrated here defines a repeatable, yet flexible solution to this problem.

Such as process is highly compatible with the Software Engineering Institute's Capability Maturity Model, which characterizes organizational processes based on how well defined and practiced they are. In theory, the features of this framework could support CMM certification all the way through to level 5. However, additional research is required to characterize the specific role of this framework in a CMM context.

This type of process requires the support of collaborative software which can organize and capture information from a variety of team members simultaneously. This is one of the fundamental design points for ADSI's Acclaro Designer.

6 REFERENCES

[1] ADSI, Axiomatic Design Solutions, Inc. http://www.axiomaticdesign.com

[2] Haimes, Y. Yacov, *Risk Modeling, Assessment, and Management*, Wiley-Interscience, 1998, ISBN 0471240052

[3] Kroll, Per and Kruchten, Philippe, *The Rational Unified Process Made Easy: A Practitioner's Guide to Rational Unified Process*, Addison-Wesley, 2003, ISBN 0321166094

[4] Pimmler, Thomas U. and Eppinger, Steven D., "Integration Analysis of Product Decompositions", *Proceedings of the ASME Sixth International Conference on Design Theory and Methodology*, Minneapolis, MN, Sept., 1994.

[5] Pressman, Roger S., *Software Engineering: A practitioner's Approach*, McGraw-Hill, New York, 1997. ISBN 0-07-052182-4

[6] Suh N.P., *The Principles of Design*, New York: Oxford University Press, 1990. ISBN 0-19-504345-6

[7] Suh N.P., *Axiomatic Design, Advances and Applications*, New York: Oxford University Press, 2001. ISBN 0-19-513466-4

[8] Ulrich, K.T. and Eppinger, Steven D., *Product Design and Development*, McGraw-Hill, New York, 2003. ISBN 0072471468

Copyright © 2004 by ICAD2004