Object-Oriented Systems Engineering

By

Anthony Lynn Peterson

A MASTER OF ENGINEERING REPORT

Submitted to the College of Engineering at
Texas Tech University in
Partial Fulfillment of
The Requirements for the
Degree of

MASTER OF ENGINEERING

Approved

_____
Dr. J. Smith

_____
Dr. A. Ertas

_____
Dr. T. Maxwell

_____
Dr. M. Tanik

October 17, 2003

ACKNOWLEDGEMENTS

This report would not have been possible without the support of my wife and family. Without their kind understanding and prodding this report would not have been completed.

I would like to express by thanks to the entire Texas Tech staff and guest speakers. A special thanks go the Dr. Tanik whose suggestions added greatly to this report.

TABLE OF CONTENTS

DISCLAIMER

The opinions expressed in this report are strictly those of the author and are not necessarily those of Raytheon, Texas Tech University, nor any U.S. Government agency.

ABSTRACT

In the defense industry a divide currently exists between the systems engineering and software engineering disciplines. While software engineering has changed methodologies from structured decomposition to object-oriented development, systems engineering has been slow to adopt object-oriented techniques. This report first examines the current practices of systems engineering. Next a new object-oriented systems engineering modeling technique is investigated.

LIST OF FIGURES

LIST OF TABLES

CHAPTER I
MOTIVATION

The inspiration for this paper came from an assignment I received in the summer of 2002. My new assignment was that of chief system engineer for an IPT that had just been formed. Early on, one of the systems engineers on the team told me that she had no background in object-oriented technology and furthermore she had no desire to become versed in object-oriented techniques. That being the case, her responsibilities were restricted to ensuring that the interface documents with other entities were documented so that there was no ambiguity as to the requirements our group was signed up to provide. When it came to verifying that our IPT detailed design would support the interface requirements, we had to rely on other systems engineers, as she was not able to understand the object-oriented design artifacts that the development team had produced.

Looking back over 26 years of experience in software and systems engineering, I recall that development techniques changed over time. At first in the 1970's software design was captured in flowcharts. This was very labor intensive since there were no CASE tools available. The software engineer first sketched each set of flowcharts on paper. Next a technical publications expert would draw the actual diagrams for the design documentation. The intuitive nature of the final documentation meant that systems engineers could grasp the design without any formal training.

By the early 1980's the next system I was involved in used PDL as the vehicle for capturing software design. This represented a giant step forward because a rudimentary CASE tool was used to process the pseudo language generated by the software engineer. The software engineer was able to directly enter the design information into the tool. The result was formatted output generated directly by the CASE tool. This meant that another group was not required to do post processing of the software engineer's inputs. Since the pseudo language was Structured English, other groups could read and understand the software design without any training.

By the late 1980's we began using Structured Analysis / Structured Design (SA/SD) CASE tools. The great advance was that like flowcharts, the SA/SD artifacts were graphical in nature. The software engineer would interact with the CASE tool to draw the diagrams that reflected the analysis and design of the software system. This is done with Data Flow Diagrams (DFDs) that were "a network representation of a system. The system may be automated, manual, or mixed. The Data Flow Diagram portrays the system in terms of its component pieces, with all interfaces among the components indicated. The diagrams are graphic, partitioned, multidimensional, emphasize flow of data, and de-emphasize flow of control." To tie all of the diagrams together was a Data Dictionary (DD) which contained definitions of all of the entities depicted on the diagrams. The DD served as a repository of data about data. With minimal training, the artifacts were understandable by other groups. (DeMarco, 1979)

By the mid 1990's Object-Oriented Analysis and Design (OOA/OOD) begin to be practiced by software engineering. "In the object-oriented approach, the entities of computation are *objects*, which send each other *messages*. These messages result in the invocation of *methods*, which perform the necessary actions. The sender of the message does not need to know how the object organizes its internal state, only that it responds to particular messages in a well-defined way." (Coleman, 1994) CASE tools for OOA/OOD were in their infancy at this time and there were many competing methods for OOA/OOD. More extensive training was required of the software developers because of the paradigm shift that had taken place. Now data encapsulation, inheritance, and polymorphism are the buzzwords that the software engineers are concerned with. Whole new sets of artifacts are generated that are not as self-intuitive as were the artifacts that had previously been generated by software engineering. Without training other groups were now unable to understand much less follow the significance of the artifacts that were now being generated. Now the systems engineers have become unable to verify that the system that was being designed would indeed be able to satisfy all of the system level requirements that they had levied to be satisfied by software.

During this time of software paradigm shift, system level requirements continued to be specified as a set of discrete statements that as a whole described the requirements of the entire system. As was the case with software engineering, systems engineering had over time embraced at first homegrown and then commercial tools for requirement management. Several such tools exist in the marketplace today. With these tools, systems engineers can maintain relationships between requirements and the entity or entities that are responsible for their satisfaction. Software Engineering has found that these artifacts inadequate for use in object-oriented development. Thus, when a software engineering team receives requirements they are to satisfy, they immediately transform them into Use Cases and Sequence Diagrams to elaborate how the software system will behave in satisfaction of their requirements before proceeding with generation of other OOA/OOD artifacts.

CHAPTER II
CURRENT SYSTEMS ENGINEERING PROCESSES

2.1 What is a System?

A system is a collection of different things, which together produce results unachievable by the elements alone. (Maier, 2002) It is a set of functional elements organized to satisfy user needs. These functional elements include hardware, software, people, facilities, data, and services. A system includes the facilities, equipment, special tooling, or processes to establish the manufacturing, test, distribution, training, support, operations, and disposal capabilities. To be useful a system must satisfy a useful purpose at an affordable cost for an acceptable period of time.

Complex systems usually contain subsystems that perform simpler tasks that contribute to the major system goal. These subsystems may be complex themselves, containing their own subsystems. The lowest level in the system hierarchy is the individual components that make up a given subsystem.

Systems do not exist by themselves. They have boundaries that separate and define them from their external environment. Systems normally interact with and are affected by their environment, accepting input and providing output as necessary.

A system has a life cycle from inception to disposal, including identification of a potential customer's perceived needs, addressing development, test, manufacturing, operation, support, and training activities, and continuing through various evaluations and upgrades through system disposal. The focus of this paper is the system development phase.

2.2 Systems Engineering Definition

Systems Engineers define, develop designs, and deploy systems. Systems engineering is a multi-faceted discipline, involving human, organizational, and various technical variables that work together. Systems engineering can be defined as "the definition, design, development, and maintenance of functional, reliable, and trustworthy systems within cost and time constraints." (Sage, 2000)

Systems engineers interact with users, lead teams of experts, and solve technical problems. They adopt a big-picture view of problems and their solutions, encompassing the customer's needs, system constraints, and operating environment into their consideration of possible solutions.

It is the role of systems engineering to form the elements of a system into a whole. The systems engineer does not design or build any one specific part of the system. They are responsible for guiding the evolution of each component throughout the system life cycle to ensure that, when combined, the components will form a system that meets all specified system requirements.

2.3 Major Elements of Systems Engineering During Development

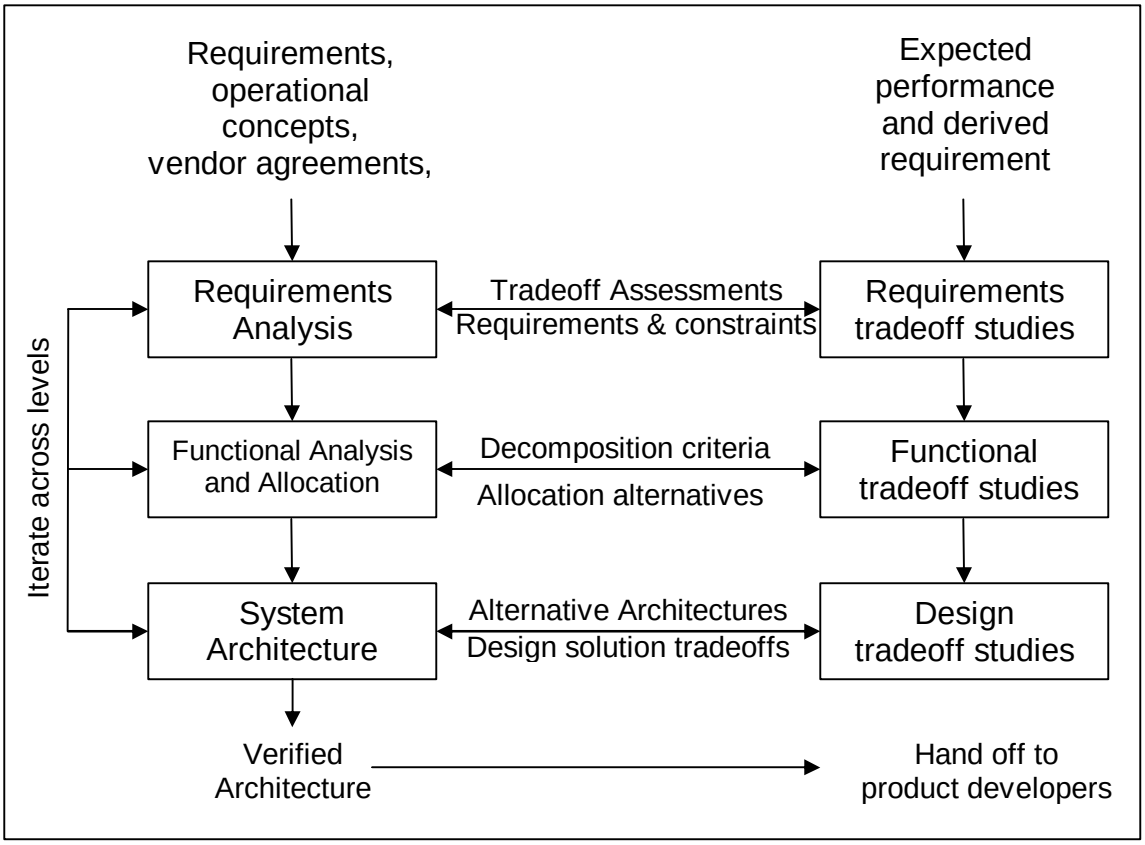
In this paper the systems engineering tasks for system development are considered. The major tasks that are associated with system development are Requirements Development, Functional Analysis, Requirement Allocation, and System Architecture and Design. These tasks are not done in a sequential manner, but are iterated with each other as the system and requirements are decomposed. Figure 1 shows a typical process for systems engineering as currently practiced in industry. This is an iterative process that attempts to evaluate the resulting architecture considered and be able to choose to best architecture. (Krikorian, 2003) This blending of tasks will result in the output products for development. While other tasks such as Trade Studies are also performed they are beyond the scope of this paper.

The classic cruise control system will be used for illustration in this paper of the various elements of systems engineering.

2.3.1 Requirements Development

Requirements Development is the systems engineering activity that examines the customer's requests and defines the needs and expectations for a product. A requirement is a condition, attribute, or capability needed by a customer or user to solve a problem or achieve an objective. A requirement identifies what, how well, and under what stated environment the condition, attribute, or capability is to

be provided or achieved. There are many different types of requirements encountered as you as the requirements for a system are defined.



Source: (Krikorian, 2003)

Figure 1. Typical Systems Engineering Process.

A good requirement from the systems engineer's perspective is much more than the actual text of the requirement statement. There are many requirement elements that need to be created, provided, completed, and documented in order to have a complete set of system, subsystem, or component requirements. Well-written and meaningful requirements share consistent characteristics. Common characteristics include that each requirement is necessary, implementation-free, concise, unambiguous, verifiable, feasible, and complete.

Requirements elements are commonly placed into a requirements tool such as DOORS, RequisitePro, RTM, or Slate. These tools enable the systems engineer to categorize elements into

requirement text, allocation to subsystem/component, verification method(s), and change control information.

Top-level requirements for the cruise control system are depicted in Table 1.

Table 1 Top-Level Requirements for Cruise Control System.

Requirement ID	Requirement Text	Verification Method
CCS0010	The driver shall be able to turn the cruise control system on.	Test
CCS0020	The driver shall be able to turn the cruise control system off.	Test
CCS0030	The driver shall be able to engage the cruise control system.	Test
CCS0040	The cruise control system shall maintain a constant speed when the cruise control system is on and engaged.	Test
CCS0050	The cruise control system shall disengage when the brake is depressed.	Test
CCS0060	The driver shall be able to request the cruise control system to resume at the previously engaged speed if the cruise control system is on and previously engaged.	Test
CCS0070	When resumed, the cruise control system shall bring the vehicle to the previously engaged speed.	Test
CCS0080	The cruise control system shall not be enabled if the speed of the vehicle is less than 30 (TBR) mph.	Test
CCS0090	The cruise control system shall be disabled when the engine is not on.	Inspection

2.3.2 Functional Analysis

Functional Analysis is the systems engineering activity in which systems engineers determine what functions the system needs to perform in order to meet needs of its missions. What separates Functional Analysis from other activities is that it is performed without regard to and independent of any implementation. The identified functions are refined and analyzed as they are decomposed into lower and lower levels of detail.

The inputs to the Functional Analysis process are the requirements that have been identified and categorized during Requirements Development. The Functional Analysis process typically results in a Context Diagram, Data Flow Diagrams, Control Flow Diagrams, Functional Flow Diagrams, N^2 diagrams, and a Concept of Operations document. The produced Functional Analysis artifacts depict the

hierarchical arrangement of functions and their internal and external interfaces, their design constraints, and their requirements.

2.3.2.1 Context Diagram

The Context Diagram identifies the external entities with which the system must interact. These external entities include other systems and users. An example Context Diagram for the cruise control system is illustrated in Figure 2. On a Context Diagram the dotted lines are control flows and the solid lines are data flows.

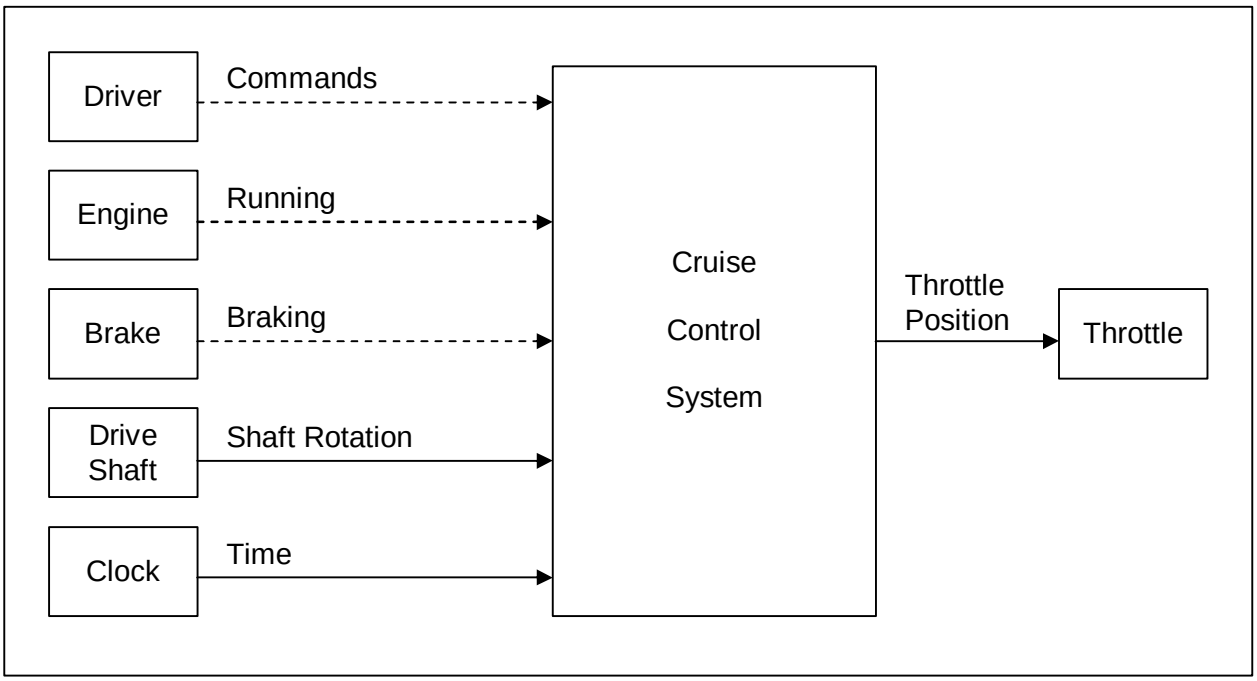


Figure 2. Context Diagram for the cruise control system.

2.3.2.2 Data and Control Flow Diagrams

Data Flow Diagrams illustrate the information flow with in the system. Data flows are shown as arrows. Processes are shown as circles. Data files are shown as two parallel lines. Data sources and external actors are shown by rectangles.

Control Flow Diagrams identify the external the system must interact with. Control flows like data flows are shown as arrows. Processes are still shown with circles. Control sources and external actors are shown by rectangles.

Figure 3 shows a combined Data and Control Flow for the cruise control system. The Data Flows are shown in dark gray while control flows are shown in light gray.

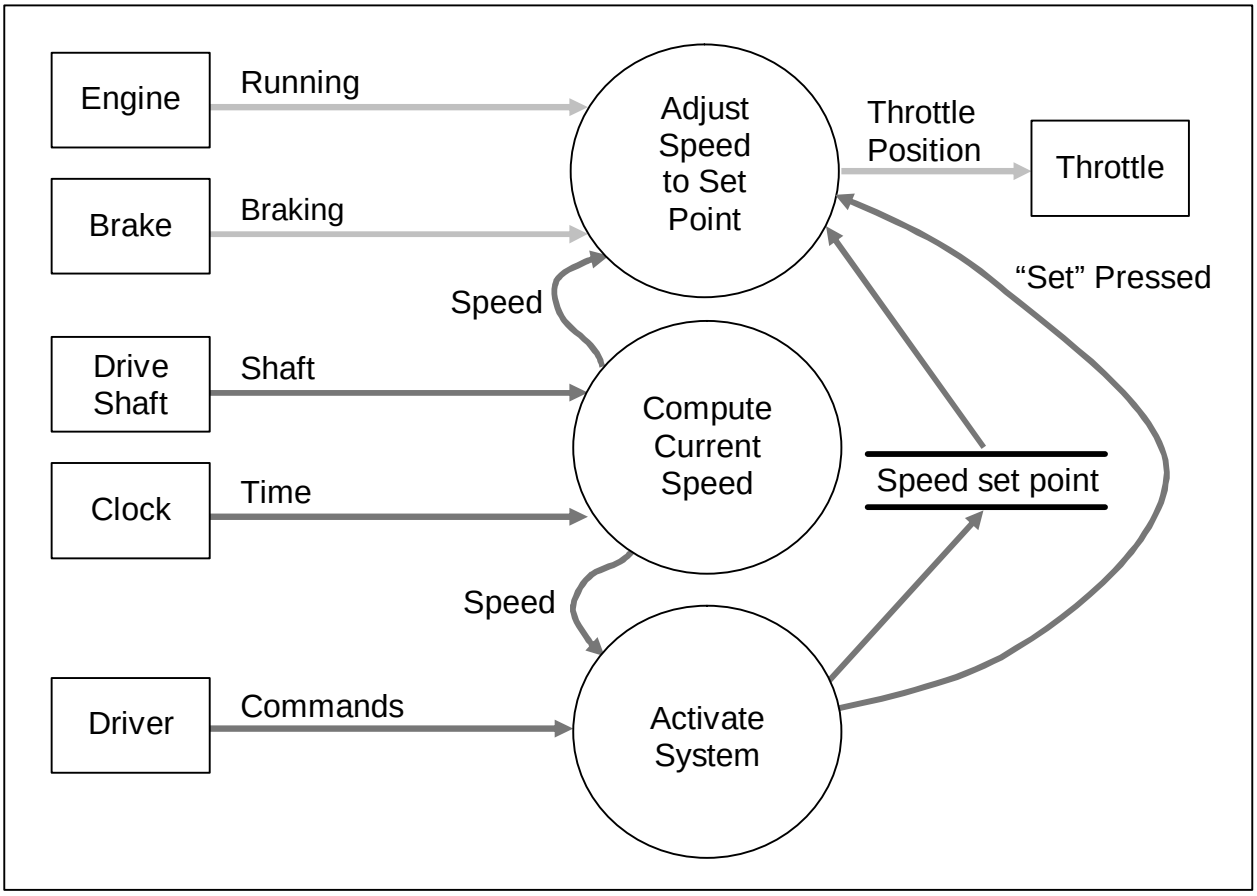


Figure 3. Data and Control Flow Diagram for the cruise control system.

2.3.2.3 Functional Flow Diagrams

Functional Flow Diagrams are designed at each level of functional decomposition. The top-level shows the system broken into system level functions. Successive decompositions detail each function of the previous level using numbered blocks to show traceability throughout the system to the function

origin on the top-level diagram. The diagrams depict what functions are preformed in sequence and which are preformed in parallel as well as which paths are optional.

See Figure 4 for top-level Functional Flow Diagram of the cruise control system. The cruise control system is decomposition into three parts. First the "Activate Cruise Control" function receives the system on/off requests from the driver. It also receives the engine on/off signal. Another input received is the engage request, which will request the current speed and activate the system to maintain a constant speed. The "Activate Cruise Control" function also receives the brake pressed signal and disengages the cruse control system. The "Compute Current Speed" function receives the pulses from wheel and clock inputs and computes the current speed. The "Adjust Speed to Set Point" function maintains the target speed by frequently obtaining current speed and requesting the engine to accelerate or decelerate as needed.

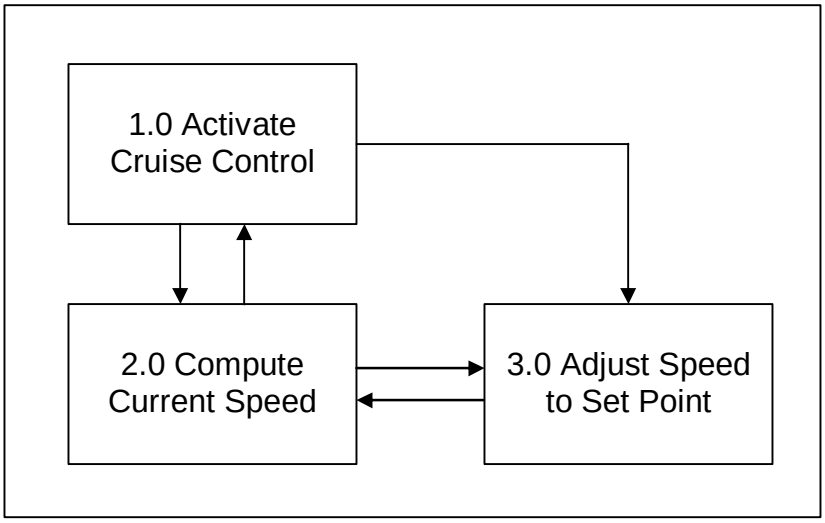


Figure 4. Top-Level Functional Flow Diagram for the cruise control system.

Figure 5 shows a decomposition of "3.0 Adjust to Set Point" on the top-level Functional Flow Diagram of the cruise control system. This has been decomposition into six parts. First the Receive Set Point for Speed receives the engage request from the "1.0 Activate Cruise Control" function. Next the "3.2 Maintain Speed at Set Point" function will request the current speed from "2.0 Compute Current

Speed", remember that speed set point, and activate "3.2 Maintain Speed at Set Point" function. The "3.2 Maintain Speed at Set Point" function maintains the target speed by frequently obtaining current speed from "2.0 Compute Current Speed" and requesting the engine to accelerate or decelerate as needed. The "3.3 Accelerate to Set Point" will be invoked to raise the speed to the speed set point. The "3.4 Decelerate to Set Point" will be invoked to lower the speed to the speed set point. The "3.5 Receive Brake Depressed" function receives the brake pressed signal and disengages the cruise control system by notifying the "3.2 Maintain Speed at Set Point". The "3.6 Receive Resume Request" function requests the "3.2 Maintain Speed at Set Point" to start again maintaining speed at the set point.

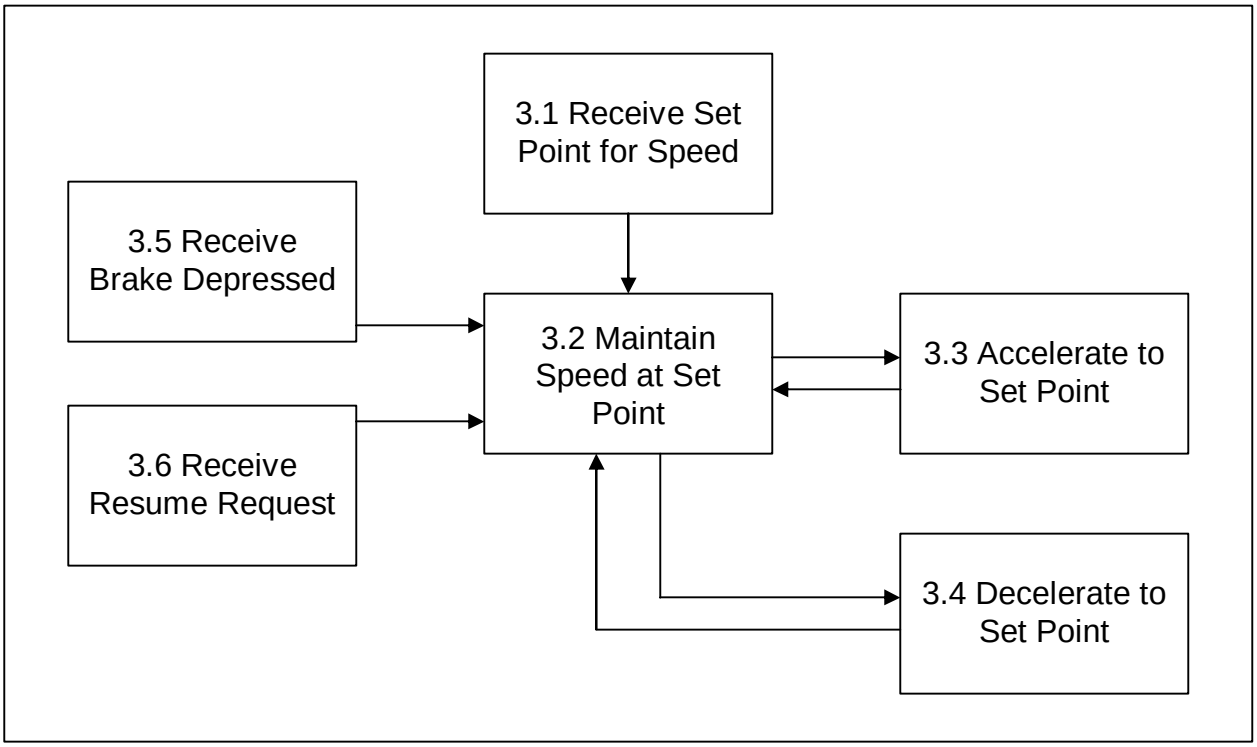


Figure 5. Second Level Functional Flow Diagram for the cruise control system.

2.3.2.4 N^2 Diagrams

N^2 Diagrams show relationships between different functions of the system. The result is an enumeration of the interfaces between functions. Table 2 depicts a N^2 Diagram for the cruise control system for the functions in Figure 2.

Table 2 N^2 Diagram for Top-Level Functions of the Cruise Control System.

1.0 Activate Cruise Control	1.0 ? 2.0 Request Current Speed	1.0 ? 3.0 New Speed Set Point
2.0 ? 1.0 Current speed	2.0 Compute Current Speed	2.0 ? 3.0 Current speed
	3.0 ? 2.0 Request Current Speed	3.0 Adjust Speed to Set Point

2.3.2.5 Concept of Operations

The Concept of Operations document portrays the system from the point of view of the end-user. This document describes the operation of the system and its use in the intended environment. The Concept of Operations describes what the system is supposed to do, the order in which it is done, and within what time constraints. Thus the Concept of Operations describes what the system is supposed to provide the customer in a multitude of situations. See Appendix A for the Concept of Operations for the cruise control system.

2.3.3 Requirement Allocation

While decomposing in the Functional Analysis process the system engineer also allocates requirements to the system level functions and their subsequent sub-functions. The goal of system level mapping requirements to functions is to ensure that all requirements are satisfied by the identified functions and sub-functions. These relationships are captured in the requirements tool. Table 3 contains the addition of requirement allocation to the top-level functions of the cruise control system. Table 4 contains the decomposed requirements for the "3.0 Adjust to Set Point" function and their traceability to parent requirements and allocation to the second-level functions.

Table 3 Top-Level Requirement Allocations to the Cruise Control System.

Requirement ID	Requirement Text	Allocation	Verification Method
CCS0010	The driver shall be able to turn the cruise control system on.	Activate Cruise Control	Test

Requirement ID	Requirement Text	Allocation	Verification Method
CCS0020	The driver shall be able to turn the cruise control system off.	Activate Cruise Control	Test
CCS0030	The driver shall be able to engage the cruise control system.	Activate Cruise Control Adjust Speed to Set Point	Test
CCS0040	The cruise control system shall maintain a constant speed when the cruise control system is on and engaged.	Compute Current Speed Adjust Speed to Set Point	Test
CCS0050	The cruise control system shall disengage when the brake is depressed.	Adjust Speed to Set Point	Test
CCS0060	The driver shall be able to request the cruise control system to resume at the previously engaged speed if the cruise control system is on and previously engaged.	Activate Cruise Control Adjust Speed to Set Point	Test
CCS0070	When resumed, the cruise control system shall bring the vehicle to the previously engaged speed.	Compute Current Speed Adjust Speed to Set Point	Test
CCS0080	The cruise control system shall be disabled when the engine is not on.	Activate Cruise Control	Inspection

Table 4 Second-Level Requirement Allocation to the Adjust to Set Point function.

Requirement ID	Parent Requirement ID	Requirement Text	Allocation	Verification Method
CCS0031	CCS0030	The cruise control system shall retain current speed as a set point whenever engaged.	Receive Set Point for Speed	Test
CCS0041	CCS0040	If speed is below the speed set point the cruise control system shall accelerate to the speed set point when the cruise control system is active.	Maintain Speed at Set Point Accelerate to Set Point	Test
CCS0042	CCS0040	If speed is above the speed set point the cruise control system shall decelerate to the speed set point when the cruise control system is active.	Maintain Speed at Set Point Decelerate to Set Point	Test
CCS0051	CCS0050	The cruise control system shall receive a signal when the brake is depressed.	Receive Brake Depressed	Test

Requirement ID	Parent Requirement ID	Requirement Text	Allocation	Verification Method
CCS0052	CCS0050	The cruise control system shall stop controlling speed of the vehicle when the brake depressed signal is received.	Maintain Speed at Set Point	Test
CCS0061	CCS0060	When requested, the cruise control system shall resume if the cruise control system is active.	Receive Resume Request	Test
CCS0071	CCS0070	When resumed, the cruise control system shall bring the vehicle to the previous speed set point.	Maintain Speed at Set Point	Test

2.3.4 System Architecture and Design

System Architecture is the major physical properties, style, structure, interactions, and purpose of a system. This is sometimes referred to as the physical architecture. It is the association of the system functional and performance requirements with physical entities. The development of the System Architecture uses the artifacts from the Requirement Analysis, Functional Analysis, and Requirement Allocation phases. On-going with construction of the System Architecture is development of the System Design.

During the System Architecture phase the systems engineer must know and concentrate on the critical details and interfaces that really matter so that he does not become overloaded with all the details of the system. This is important for effective relationships with the client and the others on the systems engineering team. To the extent in this phase that the systems engineer becomes concerned with the Functional Analysis and System Design is with those specific details that critically affect the system as a whole. (Maier, 2002)

Besides the critical details, the greatest concerns during the System Architecture phase are "with the systems' components and interfaces because: (1) they distinguish a system from its components; (2) their addition produces unique system-level functions; (3) subsystem specialists are likely to concentrate most on the core and least on the periphery of their subsystems, viewing the latter as (generally welcomed) external constraints on their internal design" sense their concern is less for the system as a

whole. This stresses the need for viewing the system as a whole because if not managed well, the system's functions can be in jeopardy when integrated together. (Maier, 2002)

Quality systems demand that quality time be dedicated to the architecture of the enterprise prior to detail design and implementation. There are three generally accepted frameworks for describing the architecture of a system. They are the Zachman Framework, the Department of Defense Architecture Framework (DoDAF) formally known as C4ISR, and the Rational 4+1 Views. Experience shows the benefits of multiple views of the system along with a structured, consistent approach for describing and developing complex systems. For defense work with the DoD, the DoDAF is the preferred architecture, but that does not preclude developing views from the Zachman Framework or the Rational 4+1 Views where the systems engineer finds those of benefit.

2.3.4.1 Zachman Framework

The Zachman Framework is a comprehensive, logical, and systematic classification scheme for organizing the primitive architectural artifacts of an enterprise. It depicts how everything in the enterprise fits together and enables analysis of specific aspects of the enterprise. It is a matrix whose columns are the six interrogatives: what (data), how (functions), where (network), who (people), when (timing), and why (motivation). The rows represent different perspectives of the enterprise: contextual (planner), conceptual (owner), logical (designer), physical (builder), and out-of-context (sub-contractor). Table 5 shows the associations of the rows and columns of the Zachman Framework.

Table 5 Zackman Framework.

	Data (What)	Function (How)	Network (Where)	People (Who)	Time (When)	Motivation (Why)
Objectives/ Scope (Contextual) *Planner*	List of things important to the enterprise	List of processes the enterprise performs	List of locations where the enterprise operates	List of organizational units	List of business events / cycles	List of business goals / strategies
Enterprise Model (Conceptual) *Owner*	Entity relationship diagram	Business process model (physical date flow diagram)	Logistics network (nodes and links)	Organization chart, with roles; skill sets; security	Business master schedule	Business Rules

	Data (What)	Function (How)	Network (Where)	People (Who)	Time (When)	Motivation (Why)
System Model (Logical) *Designer*	Data model (converged entities, fully normalized)	Essential data flow diagram; application architecture	Distributed system architecture	Human interface architecture (roles, data, access)	Dependency diagram, entity life history (process structure)	Business rule model
Technology Model (Physical) *Builder*	Data architecture (tables and columns); map to legacy data	System design; structure chart, pseudo-code	System architecture (hardware, software types)	User interface (how the system will behave); security design	Control flow diagram (control structure)	Business rule design
Detailed Representation (out-of-context) *Sub-Contractor*	Data design (denormalized) physical storage design	Detailed program design	Network architecture	Screens, security architecture (who can see what?)	Timing definitions	Rule specification in program logic
Functioning System	Converted data	Executable programs	Communica-tions facilities	Trained people	Business events	Enforced rules

Source: Baake (2003)

2.3.4.2 Department of Defense Architecture Framework (DoDAF)

Each view within the DoDAF is made up of required (essential) and optional elements. In use the systems engineer must complete all essential elements along with selected optional elements as seen appropriate. Each of these elements is a modeling method. (Maier, 2002) The four major views of the DoDAF are the all views, operational, system, and technical.

There are three all views. The first two are required. They are denoted as Overview and Summary (AV-1), Information and Integrated Dictionary (AV-2), and Capability Maturity Profile (AV-3). They are simple, textual objects. The AV-1 serves two purposes. In the initial phases of architecture development it serves as a planning guide. Upon completion of an architecture project it provides summary textual information concerning the six interrogatives: who, what, when, why, and how. The AV-2 is a typical data dictionary providing definitions of all terms used in the framework. The AV-3 discusses the maturity of the system over time. (Baake, 2003)

"The operational view shows how military operations are carried out through the exchange of information. It is defined as a description of task and activities, operational elements, and information

flows integrated to accomplish support military operations." (Maier, 2002) It contains descriptions (often graphical) of the operational elements, assigned tasks and activities, and information flows required to support the warfighter. It defines the types of information exchanged, the frequency of exchange, which tasks and activities are supported by the information exchanges, and the nature of information exchanges in detail sufficient to ascertain specific interoperability requirements. The three essential products are: High-Level Operational Concept Graphic (OV-1), Operational Node Connectivity Description (OV-2), and Operational Information Exchange Matrix (OV-3). The six optional products are: Command Relationships Model (OV-4), Activity Model (OV-5), Operational Rules Model (OV-6a), Operational State Transition Model (OV-6b), Operational Event/Trace Description (OV-6c), and Logical Data Model (OV-7). (Baake, 2003)

The systems architecture view is a description, including graphics, of systems and interconnections providing for, or supporting, warfighting functions. (Maier, 2002) For a domain, the systems architecture views shows how multiple systems link and interoperate, and may describe the internal construction and operations of particular systems within the architecture. For the individual system, the systems architecture view includes the physical connection, location, and identification of key nodes (including materiel item nodes), circuits, networks, warfighting platforms, etc., and specifies system and component performance parameters (e.g., mean time between failure, maintainability, availability). The systems architecture view associates physical resources and their performance attributes to the operational view and its requirements per standards defined in the technical architecture. The one essential product is the System Interface Description (SV-1). The twelve optional products are: Systems Communications Description (SV-2), Systems^2 Matrix (SV-3), Systems Functional Description (SV-4), Operational Activity to System Traceability Matrix (SV-5), System Information Exchange Matrix (SV-6), Systems Performance Parameters Matrix (SV-7), System Evolution Description Matrix (SV-8), System Technology Forecast (SV-9), System Rules Model (SV-10a), System State Transcription Description (SV-10b), Systems Event Trace Description (SV-10c), Physical Data Model (SV-11). (Baake, 2003)

The technical architecture view is the minimal set of rules governing the arrangement, interaction, and interdependence of system parts or elements, whose purpose is to ensure that a conformant system satisfies a specified set of requirements. (Maier, 2002) The technical architecture view provides the technical systems-implementation guidelines upon which engineering specifications are based, common building blocks are established, and product lines are developed. The technical architecture view includes a collection of the technical standards, conventions, rules and criteria organized into profile(s) that govern system services, interfaces, and relationships for particular systems architecture views and that relate to particular operational views. The one essential product is the Technical Architecture Profile (TV-1). The one optional product is the Standards Technology Forecast (TV-2). (Baake, 2003)

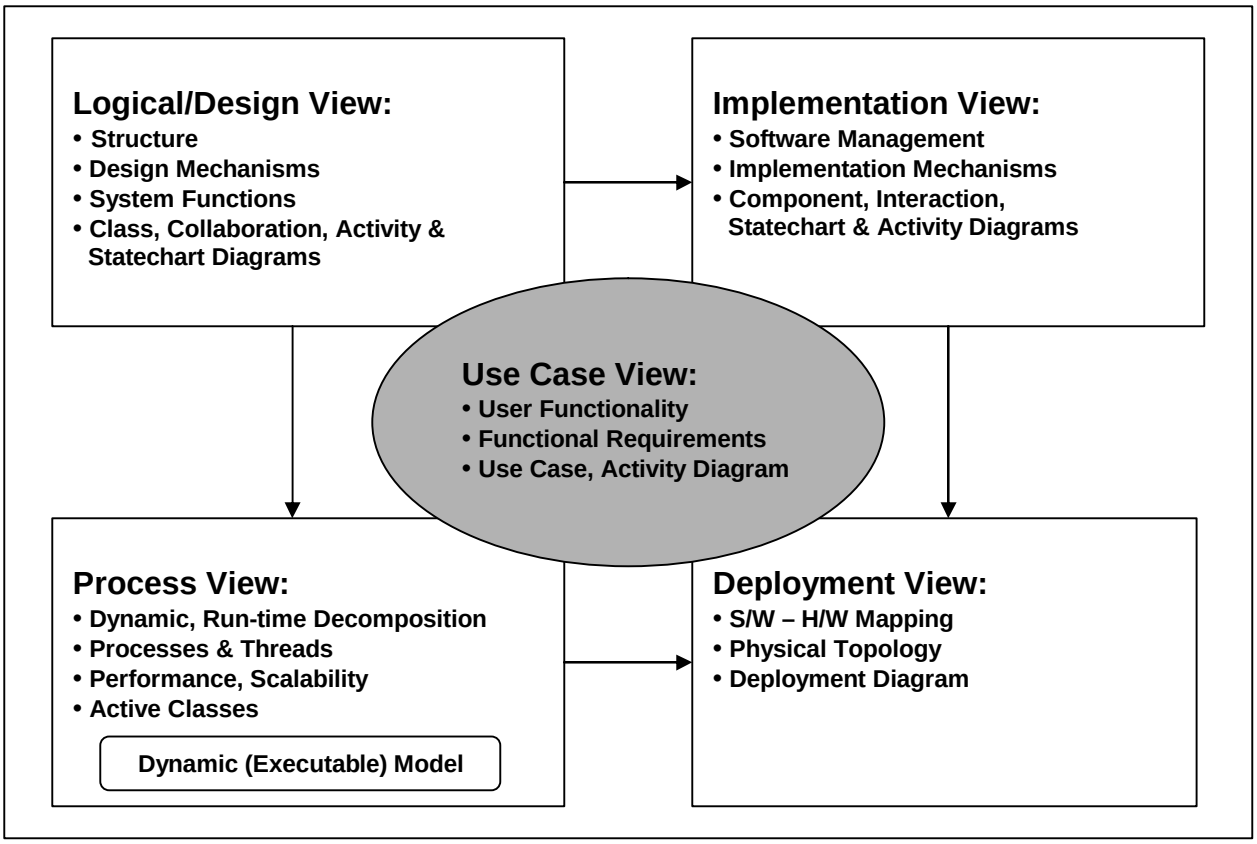
2.3.4.3 Rational 4+1 Views

The Rational 4+1 Views architecture consists of the Logical/Design View, Implementation (Development) View, Process View, and Deployment (Physical) View all surrounding the Use Case (Scenario) View. Figure 6 depicts the relationships of the five views.

The 4+1 View model allows the various stakeholders to find what they want to know about the architecture. Systems engineers approach this from the Process and Deployment views. Data specialists, customers, and end-users approach from the Logical/Design view. Project managers and software configuration management see it from the Deployment view. (Kruchten, 1995) Table 6 shows a summery of the Rational 4+1 Views.

2.3.4.4 System Design

The purpose of System Design is to further describe and refine the subsystem elements in order to provide a well-defined system to be developed. This involves translating the System Architecture into products, components, and process solutions. Thus the System Architecture is converted into a realizable design involving hardware, software, and operations. This effort is iterative, as more of the System Architecture is known more full featured becomes the System Design.



Source: Baake (2003)

Figure 6. Rational 4+1 Views.

Table 6 Summary of the Rational 4+1 View Model.

View	Logical	Process	Development	Physical	Use Case
Components	Class	Task	Module	Subsystem, Node	Step, Scripts
Connectors	Association, Inheritance, Containment	Rendezvous, Message, Broadcast, RPC, JMS, etc.	Compilation dependency, "with" clause, "include"	Communication medium, LAN, WAN, bus, etc.	
Containers	Class category	Process	Subsystem (library)	Physical subsystem	Web
Stakeholders	End-user	System designer, integrator	Developer, Manager	System designer	End-user, Developer
Concerns	Functionality	Performance, Availability, Software fault-tolerance, Integrity	Organization, Portability, Reuse, Line-of-product	Scalability, Performance, Availability	Understand-ability

Source: (Kruchten, 1995)

Inputs to System Design are the artifacts from the other systems engineering efforts. System Design artifacts include Interface Specifications which document the external and internal interfaces between systems, products, subsystems, and components. Typical Interface Specifications include requirement specifications for the interface, definition of the exchange over the interface, and design of the interface.

During this phase, system requirements are further mapped to ensure that the finished solution will meet all requirements by identifying where each portion of a requirement is satisfied. This provides another means to monitor and track progress against the requirements form the user. Remember that requirements were initially allocation from the Functional Analysis of the system. This mapping is a continuation of the requirements allocation process down to the design level of detail. The relationship of requirements, functions, and products is captured in the requirements tool. From this tool, reports can be pulled that will show interrelationships of requirements to the various work products.

CHAPTER III
OBJECT-ORIENTED SYSTEMS ENGINEERING

3.1 The Issue with Current Systems Engineering Processes

Despite all of the efforts of systems engineers, many times products fail to meet the expectations of the customer. Often, these types of failures are attributed to gaps in the interactions between the various engineering disciplines (systems, software, hardware, etc.). Contributing to this gap is the fact that these many different disciplines use different techniques for requirement analysis, functional analysis, architecture partitioning, and design development. This gap results from incomplete communications resulting from an incomplete set of requirements used to develop the system, misunderstanding the architecture description, and/or some other collection of missing attributes of the system. (Krikorian, 2003)

To foster added communication, these different disciplines need to establish a common vocabulary. For software intensive systems, methods used in both disciplines need to be evaluated. Since software engineering has switched from Structured Analysis and Design (SA/SD) to Object-Oriented Analysis and Design (OOA/OOD), can object-oriented techniques be also used by systems engineering? That is the focus of this chapter.

3.2 Object-Oriented Systems Engineering (OOSE)

Since the Unified Modeling Language (UML) is the de facto standard for software OOA/OOD, UML is a reasonable starting place for Object-Oriented Systems Engineering (OOSE). Today UML supports a language for incremental and iterative software requirements analysis and detailed design through OOA/OOD. The UML standard, currently at version 1.3 permits specification of the software product independent of programming language or development process. "This independent product representation has raised interest in the systems engineering community: OO methods might be a mechanism to unite product development disciplines and remove the gap between the specified and as-built work products." (Krikorian, 2003) Version 2.0 is being targeted to have extensions to enable

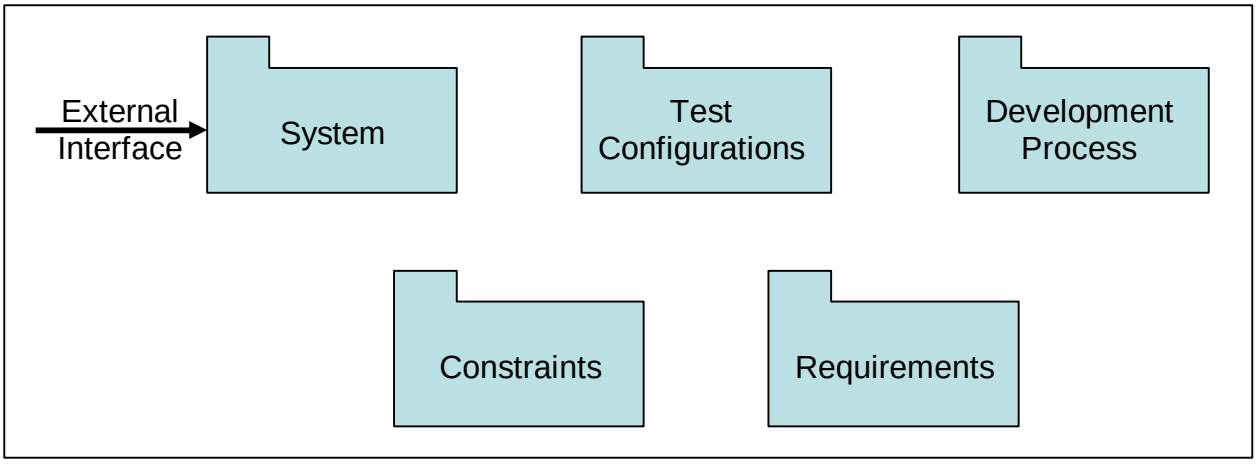
systems engineers to generate additional artifacts to support their needs for Requirements Development, Functional Analysis, Requirement Allocation, and System Architecture and Design, as well as support for Subsystem and System Testing.

OOSE proposes two additional diagrams to supplement the 4+1 architecture views. The first is the Metacontext Diagram. The other is the System Assembly view. All of these views together help ensure product consistency at all levels of system decomposition. (Krikorian, 2003)

3.2.1 Metacontext Diagram

The top-level Metacontext Diagram, as depicted in Figure 7, "captures the system, external influences, constraints performance requirements, test configuration, and development requirements and constraints on the elements. … These views also shed light on element development issues, requirements, and constraints; they also provide a context for the other views.

"This diagram also provides a forum for capturing the hierarchy of product elements and the impacts on system development. It captures these items as packages to support decomposition and groups these influences on the system. This layering of packages ensures that the context for the element under investigation is available to the analyst and developer for product definition." (Krikorian, 2003)



Source: (Krikorian, 2003)

Figure 7. Metacontext Diagram.

3.2.2 System Assembly View

A System Assembly view provides an extension on the analysis techniques used for software-based systems. This additional view attempts to reconcile the mismatch among the various physical, hardware, and software architectures. By capturing the expected product hierarchy, the System Assembly view "provides a context for guiding the development of the 4+1 architecture view." (Krikorian, 2003) While the cruise control example used until now was for only the software to be developed, note that this view also includes the hardware and other disciplines that drive the overall system. An example System Assembly view for the cruise control system with hardware elements is shown in Figure 8.

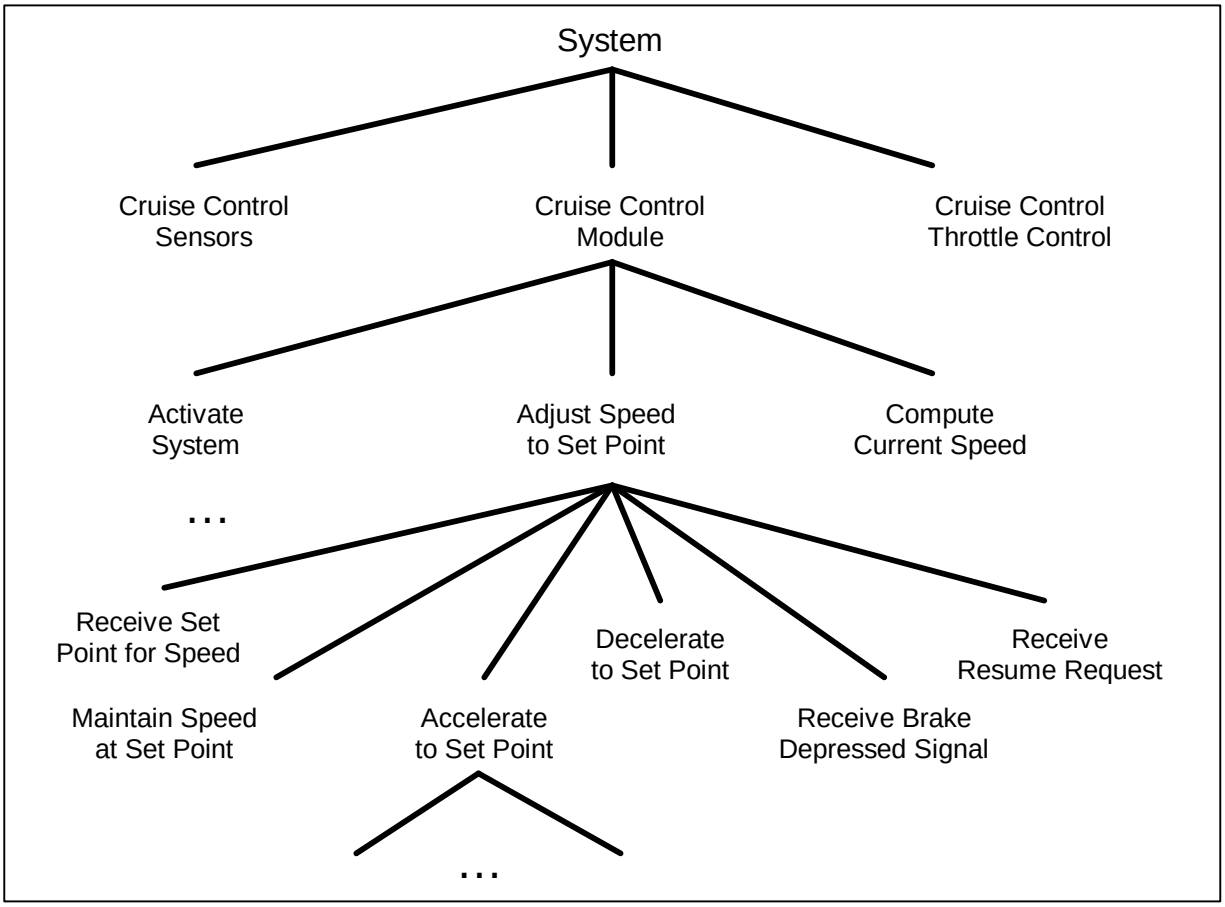


Figure 8. System Assembly View for Cruise Control System.

From domain dependent and independent analysis, performance and non-functional requirements are allocated to the physical attributes of lower-level elements in the System Assembly view. This analysis considers such things as algorithm accuracies, error margins and budgets, performance budgets, and manufacturing issues to develop criteria for allocating parameters to the product elements. (Krikorian, 2003)

Facets of issues that impact Requirements Development, Functional Analysis, Requirement Allocation, and System Architecture and Design processes are incorporated in the System Assembly view. Its layered tree of expected system elements and their conceived behavior guides the system engineer using the OOSE process in describing how the product elements work together so that the system can perform its intended functions. This view also emphasizes roles, behavior, development items, and issues that are critical to the system. (Krikorian, 2003)

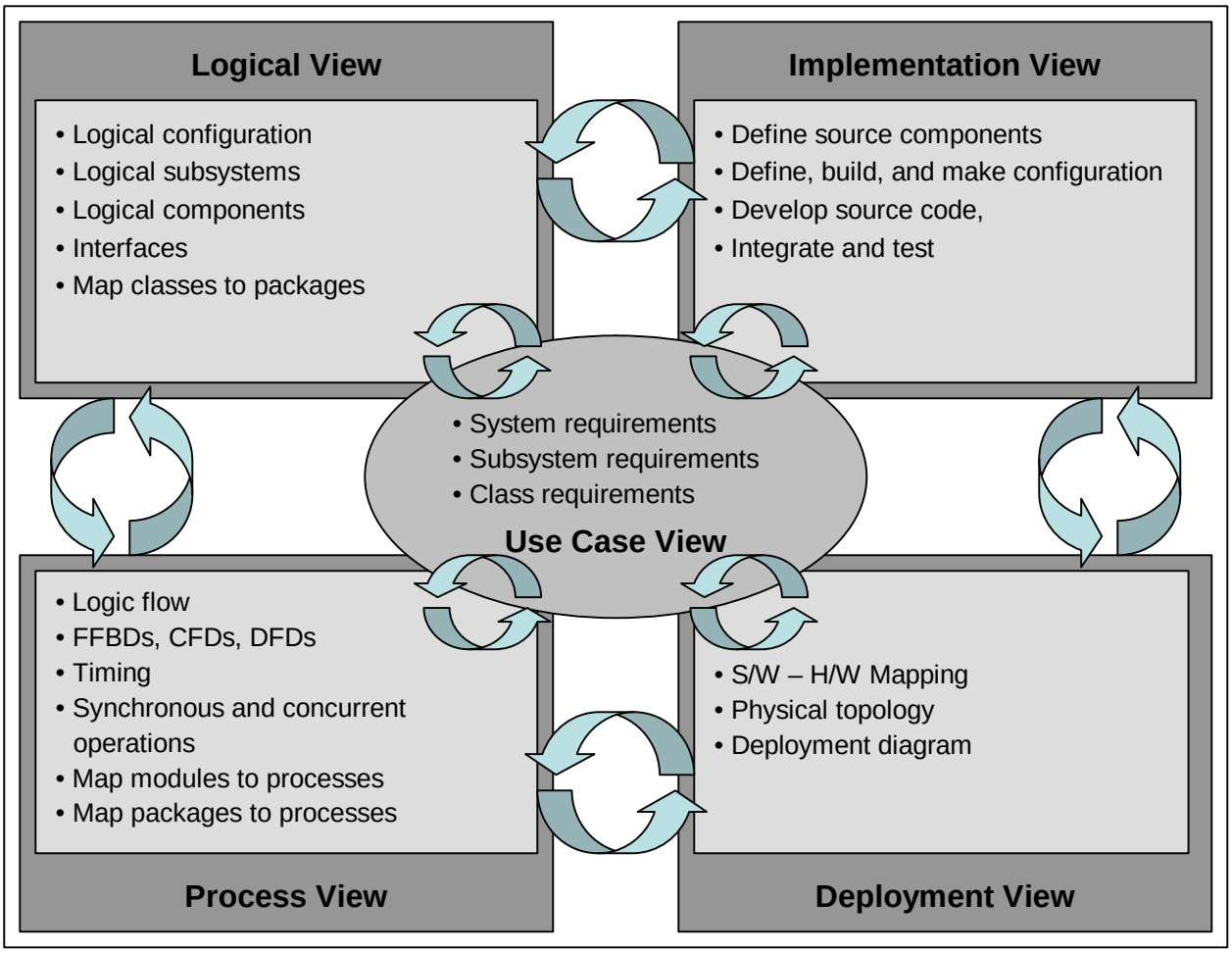
3.2.3 Augmented 4+1 View

Recall the Rational 4+1 view was discussed in section 2.3.4.3. It consists of 4 views (Logical, Process, Deployment, and Implementation) with a central Use Case view in the middle. For OOSE this has been extended as shown in Figure 9.

3.2.3.1 Augmented Use Case View

In the 4+1 View, the Use Case View is in the center, stressing the central role that use cases play. In addition to the use case, a textual description of a expected system behavior, sequence diagrams are drawn. A sequence diagram provides a ordered series diagram that exhibits the interactions of elements in the System Assembly view with various external actors (entities).

Cantor proposed extending the use case descriptions and associated sequence diagrams to include expected internal interactions among lower-level elements that are necessary to realize the behavior requested by external sources (Cantor, 1998) This extension to use cases provides the foundation used in OOSE to identify, group, refine, and allocate behavior to lower-level elements.



Source: (Krikorian, 2003)

Figure 9. Augmented 4+1 Views.

In OOSE, performance requirements are attached to the appropriate use case and associated sequence diagram, down to a specific use case step. Developers will then build collaboration diagrams in conjunction with a sequence diagram of a use case to allocate and discuss functional allocation, interface, performance, and communications issues. (Krikorian, 2003)

3.2.3.2 Augmented Logical View

The relationships of dependencies among system elements are captured in the Logical View. Partitioning of the system into computational and physical element services to collaborate to support system behavior is shown on the Logical View. (Krikorian, 2003)

3.2.3.3 Augmented Process View

States, modes, concurrency, and synchronization relationships are captured in the Process View. The use of UML stereotyping is exploited to express traditional functional flow, data flow, and control flow information, though not to complete satisfaction. The activity diagram portion of UML is used to provide the primary view which captures relationships and activities that prove difficult using just use cases to capture and express. Following this, developers will generate activity diagrams directly from these artifacts. (Krikorian, 2003)

3.2.3.4 Augmented Deployment/Physical View

For OOSE, the deployment view is extended to include more physical elements. If the system under development includes preexisting (legacy) systems, this view will also capture any preexisting processing resources and components and their physical relationships. These will flow into the System Architecture and Design across the system decomposition hierarchy. In conjunction with the logical view and the System Assembly view, the deployment/physical view captures the resulting element-level architecture. Also captured are any architecture impacts "from the computational-task allocation policies, computational-file allocation policies, and the selected design patterns." (Krikorian, 2003)

3.2.3.5 Augmented Implementation View

Captured in the implementation view are any realized functional components of the system. Hardware-only items are captured as components. By including mechanical structures, developers are actually aided in identifying additional performance and response requirements on software-based controls. (Krikorian, 2003)

3.3 OOSE Process Execution

Object-oriented systems engineering (OOSE) promises to allow discovery of functionally needed by a system latter in a program than the traditional "do-it-all-at-the-beginning" process prior to any development that is used today. All of the views will be developed at the same time as the system

definition evolves. The development team can begin without the system first being fully defined. Learning can occur during all activities as the system comes together. Insertion of late functionality can occur without large sections of the system being affected.

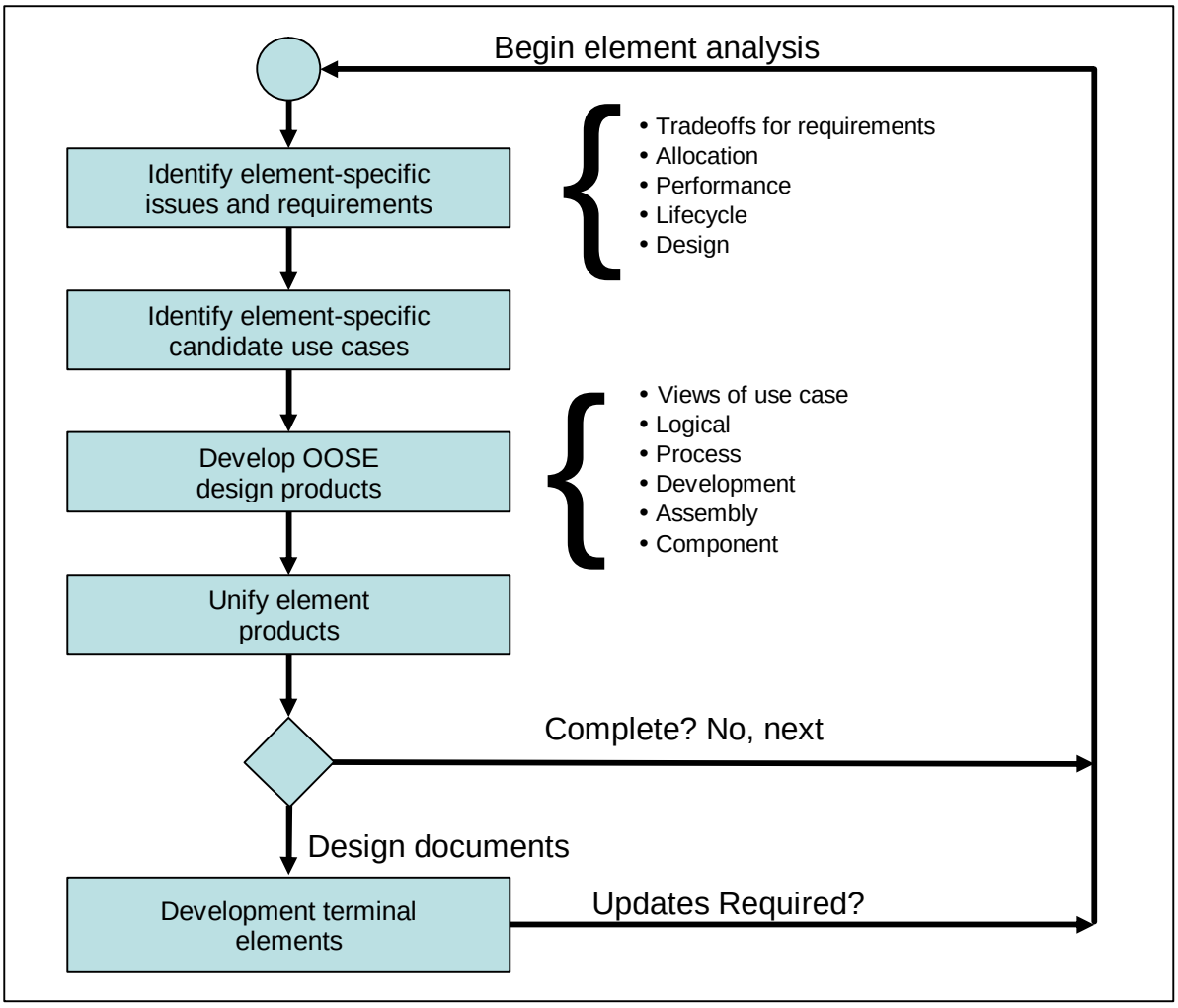
Figure 10 depicts OOSE product development, from top-level systems analysis to the definitions of all components. The systems engineering team applies this process to all elements in the system hierarchy. For each level of decomposition the products produced are a refinement of the Systems Analysis view, the supported 4+1 views, and the description of all allocation of the behavior throughout the system. Allocations of functional requirements to immediately lower elements are captured in these descriptions. Other architecturally significant nonfunctional requirements and constraints such as those affecting interfaces, operations, performance, physical layout, testing, or manufacturing are also captured. (Krikorian, 2003)

3.3.1 System Assembly View and Requirements

Typically the customer provides his requirements in one or more requirements document. These documents usually cover both function and non-functional requirements. They are the basis for the systems engineering team to generate first a system-level, then a subsystem level, and finally a component level set of requirements. The systems engineering team not only generates the requirements, but adds additional information to aid the development team in its understanding of the requirements. Employing Use Cases is a fundamental part of object-oriented software engineering. Use Cases and their associated Sequence Diagrams are a way of describing the behavior of the system from the perspective of the actors (users and other entities) that interact with the system to accomplish its objectives. (Jacobson, 1992)

From the System Assembly view a layered-tree hierarchy of relationships is established. Some system behaviors and supporting interfaces for the requirements may not be immediately visible. (“A good example of this problem is the requirement for anomaly detection or support for legacy manufacturing processes.”) Elements are added to the System Assembly view represent related, grouped

behavior. These added elements supplement the hierarchy and capture the roles of lower-level Assembly views at the Subsystem and finally Component level. (Krikorian, 2003)



Source: (Krikorian, 2003)

Figure 10. Update of Typical Systems Engineering Process for OOSE.

3.3.2 Analyze the Requirements

Leffingwell has identified five steps to perform while doing problem analysis so that the systems engineering team can understand the needs of all of the stakeholders of the new system. Since most systems have a specific set of problems to be solved, using problem analysis techniques will ensure that the team has an understanding of the problem domain. (Leffingwell, 2000)

The first step is to gain agreement among the team on the definition of the problem that is to be solved. Understanding the needs and wants of the customer and end user is necessary. Be sure to include involvement of as many stakeholders as possible. Inclusion of management often provides a different viewpoint that is valuable so that the team stays on target will all contract ional agreements. (Leffingwell, 2000)

Next, examine the problem statements and perform a root cause analysis. This is a systematic way of uncovering the underlying (or root) origin of the problems being examined. Drawing a fishbone diagram often aids in ascertaining root causes. Brainstorming is another approach that could be employed. Be careful that solutions selected are of value to the customer and end user, as many roots causes are simply not worth fixing. The updated problem statement should be distributed among as many as practical for feedback. When completed, all project members should be working toward the same goal. (Leffingwell, 2000)

Thirdly to effectively solve any complex problem all of the stakeholders and users must have their needs satisfied. Leffingwell defines a stakeholder as "*anyone who could be materially affected by the implementation of a new system or application*." Some stakeholders are not directly users of the system. Do not overlook the political environment the system will reside. It is crucially important that all stakeholder's needs are addressed adequately so that the solution is not only effective, but accepted. (Leffingwell, 2000)

Now that the problem statement has been agreed to and the users and stakeholders are defined, the boundaries of the solution to be developed must be established. Understanding all of the entities that interact with the system is vital. In object-oriented terminology these entities are referred to as actors. Leffingwell gives the following questions to aid in finding all of the actors:

- Who will supply, use, of remove information from the system?
- Who will operate the system?
- Who will perform any system maintenance?

? Where will the system be used?

? Where does the system get its information?

? What other external systems will interact with the system? (Leffingwell, 2000)

The systems engineer now can create a "system perspective" that describes the system boundary interactions with the users and other interfaces. Figure 11 depicts a simplified system perspective for the cruise control system. The dotted line shows the cruise control system boundary.

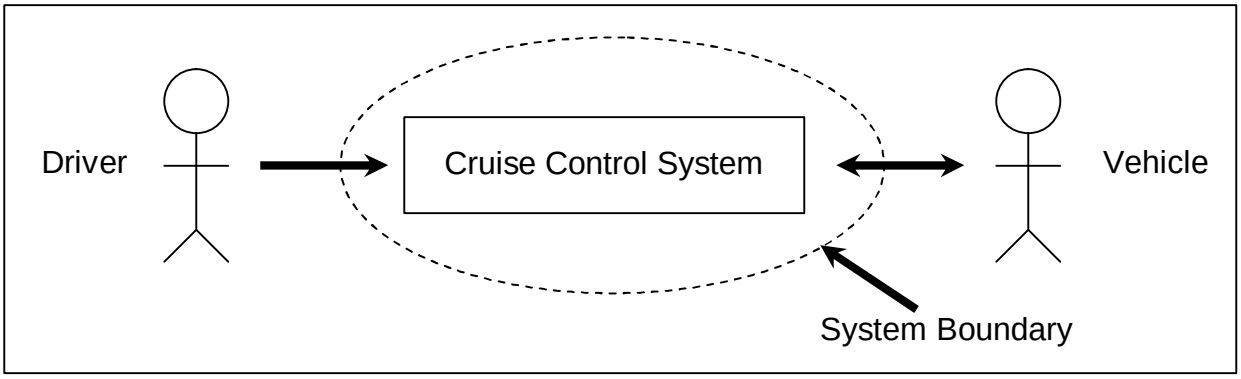


Figure 11. Simplified System Perspective of Cruise Control System.

Finally, the fifth step is identifying any constraints that need to be imposed on the system. Leffingwell defines a constraint as "*a restriction on the degree of freedom we have in providing a solution.*" Each constraint must be carefully considered as each might restrict the ability to deliver a particular selected solution. Constraints can come from many different sources including economic, political, technical, system, environmental, operational, resource, and schedule. (Leffingwell, 2000)

With these steps completed, a good understanding of the problem to be solved is accomplished. The stakeholders who will decide whether or not the project is successful are identified. An understanding of what is part of the system and what is outside of the system has been accomplished. The limits imposed by the various constraints on the system are known. (Leffingwell, 2000)

3.3.3 Identify Candidate Use Cases

In OOSE, this is one of the most important activities. Review the Concept of Operations and high-level requirements to gain an understanding of the system and its logical components and

subsystems. Now with the various stakeholders conduct a series of workshops to gain insight into product development and delivery concerns. In these workshops, using the System Assembly view as the principal framework to capture issues and drivers, identify how to best identify the top-level Use Cases. In the workshop, the Use Cases should allow for the discovery of subsystem element design drivers. The Use Cases identified must address deployment, operational, non-operational, test, and maintenance activities. (Krikorian, 2003)

For the cruise control example, several use cases can be constructed. Five candidate use cases are: Activate/Deactivate cruise control, Compute speed, Store speed, Keep speed constant, and Resume speed. Figure 12 depicts these in a Use Case Diagram.

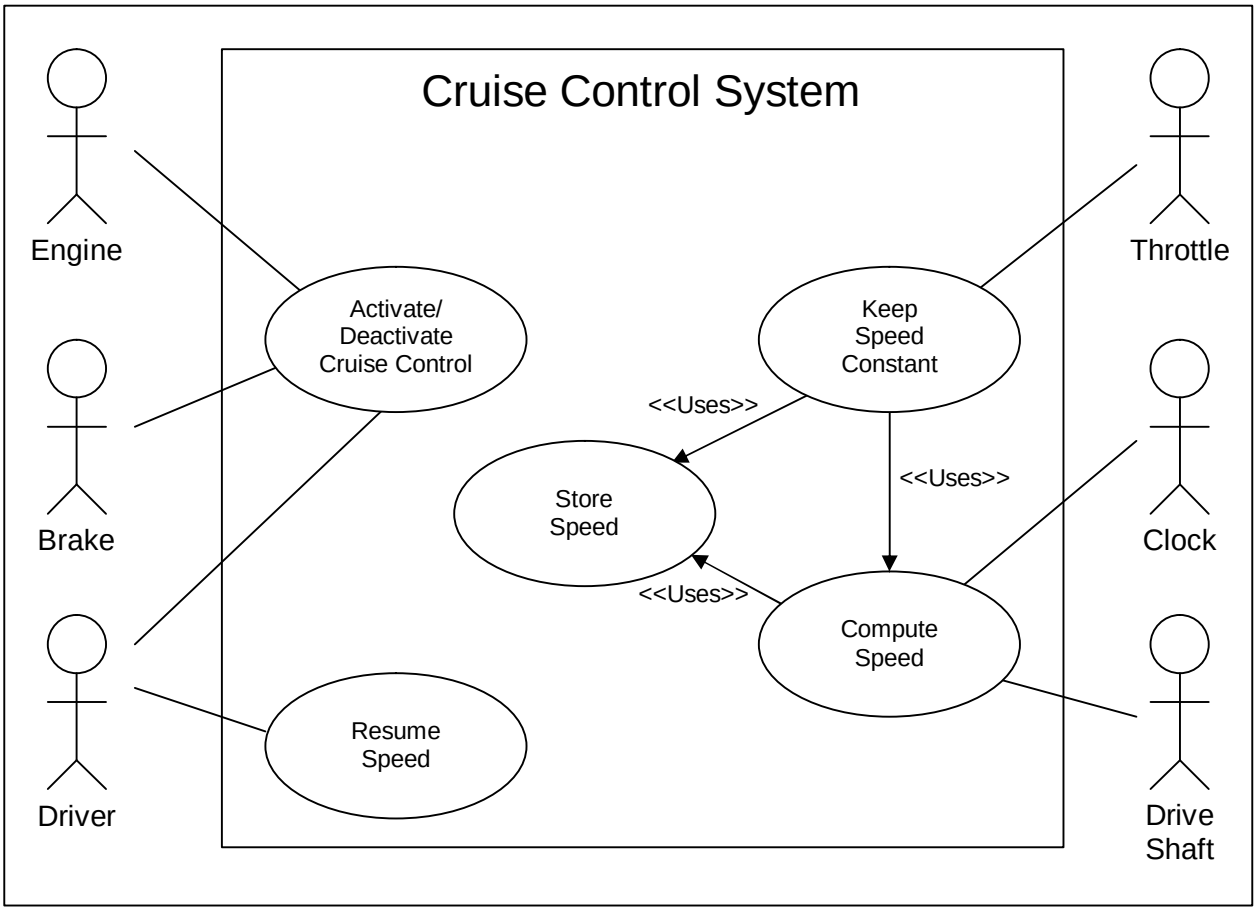


Figure 12. Use Case Diagram of Cruise Control System.

Large-scale systems that are typical in defense systems will have multiple interfaces and limited to no user interfaces often have behaviors that are not visible at the upper-level of decomposition. Consequently, it is not easy to fully describe the requirements, operational sequences, of operational states and modes imposed on subsidiary elements. Kirkorian gives two examples to illustrate this point. "For example, the systems ability to

- Conduct routine operations and maintenance
- Detect and resolve anomalies in line-replaceable units

are examples of system level requirements that do not easily map to interfaces." Even though there is not yet visibility at the system level, developers can have insight into how the lower level components would handle these types of requirements. (Krikorian, 2003)

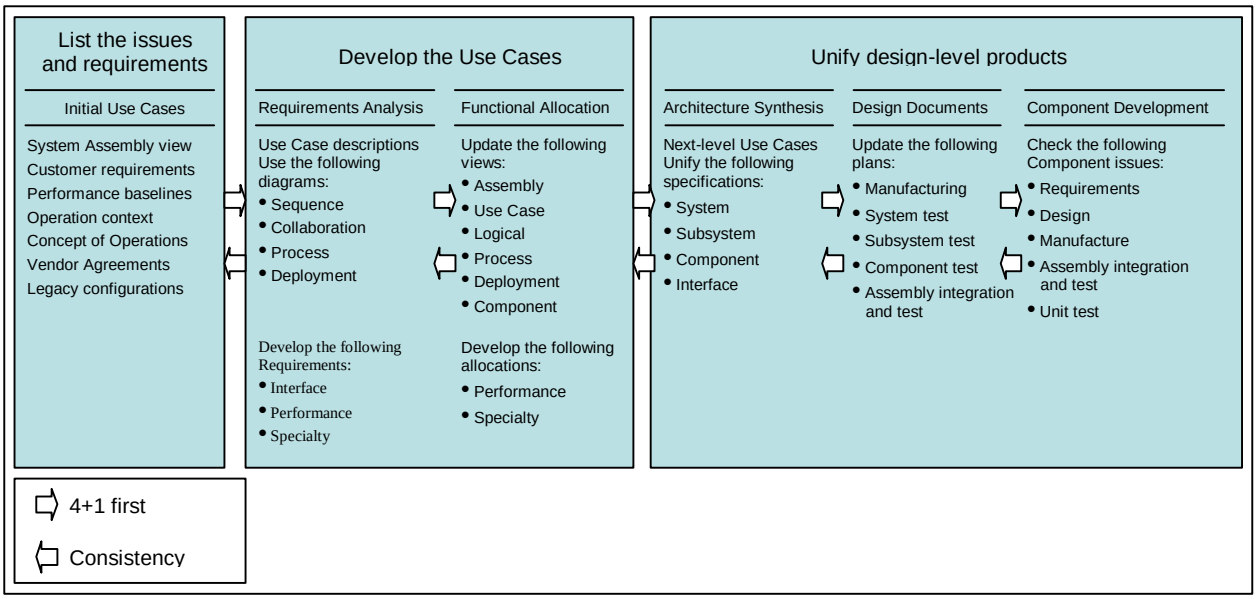
In summary, Krikorian states that the use case workshops are "therefore critical to capturing the overall system behavior, yet provides the freedom to later discover expected, lower level behavior and not expose the lower behavior at the current system-level of system decomposition, Therefore, use case descriptions have a broader variety in there descriptions at higher levels. They begin to incorporate more specific details as the analysis approaches terminal components." (Krikorian, 2003)

3.4 Using OOSE to Design Products

Once the top-level use cases are defined, systems engineers can turn their attention to use case requirements, architecture, and design products. Figure 13 illustrates this process.

3.4.1 Developing Use Cases

In use case development, the systems engineering team develops functional, interface, and operational requirements. These requirements are then allocated to the elements of the next level of the System Assembly view hierarchy. (Krikorian, 2003)



Source: (Krikorian, 2003)

Figure 13. Systems Engineering Phases for each System Assembly Element.

From Larman, use case descriptions identify the external influences on the elements and the intended response. The text description of use cases capture this response by containing

? An external view of the element's behavior in response to these interfaces

? And internal view that suggests how dependent subelements might interact to satisfy the use case. (Larman, 1997)

The system engineering teams discover, reinforce, and refine the partitioning of the subselements of the System Assembly views through their discussion of the internal view. (Krikorian, 2003)

3.4.1.1 Use Case Descriptions

As use case descriptions are developed for a level of elements in the System Assembly view, consideration must also be given to the internal behavior of the subelements of that level of decomposition. For example, in developing the use case descriptions for "Adjust to Set Point" for the cruise control system in Figure 8 in section 3.2.2, the internal behavior of the six elements in the level below must also be considered. Note that while focusing on "Adjust to Set Point" the subordinate elements to "Activate System" and "Compute Current Speed" are not considered.

Care must be taken by the systems engineering team to not drop too low too quickly. Even if functionality is suspected at a lower level of the System Assembly view than that under consideration, intentionally focus on only the interactions at the next lower level. Doing so will ensure that each level has a consistency of detail in the artifacts of that level. At this time the team also develops alternative flows for the use cases as well as expected error conditions. (Krikorian, 2003)

3.4.1.2 Design Artifacts

For each element there are design analysis artifacts which describe the operations of the logical or physical components at the next level of System Assembly decomposition. Included in these artifacts are use case descriptions and their supporting sequence diagrams. These artifacts capture the expected interaction among subelements. The downward flow of allocated behavior to the supporting subsystems and components is traced in the use case anatomy diagrams. Descriptions of the subelements capture the global constraints that are placed on lower-level elements and their allocated requirements. (Krikorian, 2003)

Activity and class diagrams make up the process views and capture critical timing relationships. Note that the process also generates deployment views that conform to these process views. (Krikorian, 2003)

At this point, a systems engineering team might discover that the originally proposed hardware configuration is inadequate. The System Assembly view is updated by the systems engineering team to reflect items that it has been added, removed or collapsed into other physical elements. The updated System Assembly view also captures the collaborations of the new roles of the lower-level components. The design thus gains cohesion, lowering the total cost of ownership. (Krikorian, 2003)

Krikorian notes that "the design team does not maintain all its design products throughout the analysis process. It maintains products only to the level required to support design decisions, the confirmation of use case operations, the allocation of requirements, and design development." (Krikorian, 2003)

For systems with components that are merely artificial elements in the decomposition hierarchy it can be impractical to develop class diagrams. Such components might just purposefully pass through broad and undefined system requirements, such as “deploy the system,” “detect and resolve anomalies,” or “ maintain the system.” Component functionality developed in support of these requirements will only be visible at the lowest level components. So, it is of no value to arbitrarily force the design and maintenance of classes for all system levels. (Krikorian, 2003)

3.4.1.3 Supplemental Requirements

Krikorian writes that “use cases also identify, capture, and assign requirements related to weight, temperature, power, size, throughput, response time, capacity, and algorithm accuracy. These requirements typically come from and are allocated by a performance flow-down analysis that exists outside the process yet works in concert with it. Having this additional context often drives the internal view’s description of behavior and, subsequently, the allocation of requirements to lower-tier elements. This context also constrains the selection of physical-component options.” (Krikorian, 2003)

3.4.1.4 Analysis Issues

In practice, OOSE has revealed similar issues as other methods have had with systems that have cooperative, asynchronous component behavior that is independent. OOSE does specifically attempt to simplify use case descriptions when these techniques are introduced to personnel unfamiliar with systems or software engineering. By using structured English to the greatest extent possible, OOSE ensures that use case descriptions are easily understood by new practitioners. Asynchronous behavior is captured in activity diagrams which supplement use case descriptions. Today these techniques are still primarily accepted by software engineering. When disciplines gain further acceptance of OOSE outside of software, it will be possible to introduce more expressive capabilities. (Krikorian, 2003)

Krikorian states that “because of the often asynchronous and mechanical nature of systems, use case descriptions must capture both sides of the interactions. That is, an internal step might state that X ? Y(A); that is, element X of the assembly architecture commands a service or function A of element Y.

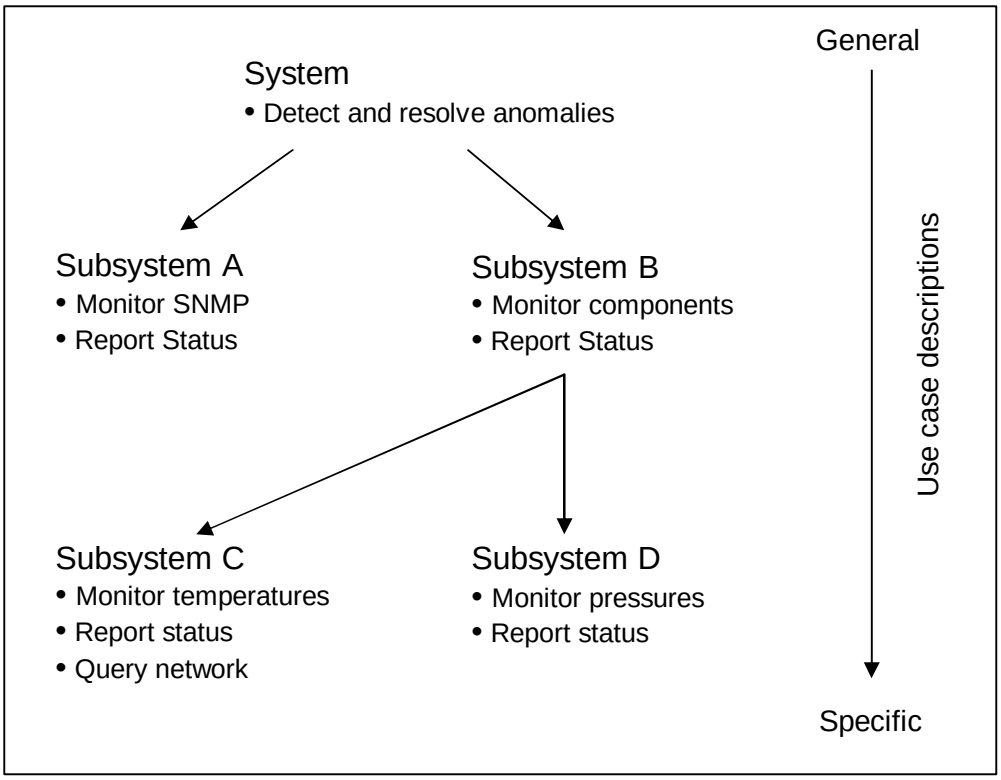
However, since many systems are asynchronous, and the requesting protocol – typically a UDP (user diagram protocol) broadcast or a force applied to a structure – is often very different than the protocol returning status, usually FTP (file transfer protocol) of mechanical movement. To address this, OOSE uses case descriptions capture extra internal steps specifically mentioning that element Y performs function A. This extra care in describing element behavior ensures a forum for discussing Y(A)-unique issues, protocols, responses, safety, and test requirements, when necessary to gain further insight into the development requirements, constraints, and component roles." (Krikorian, 2003)

3.4.2 Unifying Design-level Products

OOSE unifies the use case internal-view steps to develop the candidate set of use cases for the elements at the next level of system decomposition. During requirements analysis, an integrated understanding of the requirements for the interacting hierarchical elements might not exist. (Krikorian, 2003)

System-level sequence diagrams, unlike sequence diagrams that directly relate to classes, provide a means to express the interaction of the system hierarchy. Often, lower-level subsystem details become known later in the process; they can impact the initial understanding and consequently force a reallocation of functionality. (Krikorian, 2003)

Krikorian, in Figure 14, "depicts an example of use case decomposition from the general system requirements to the lower-level elements. This activity expands, collapses, and updates all views (use case, logical, process, deployment, implementation, and assembly) for consistency. It then distills the resulting descriptions to identify the intended subelement behavior. (Krikorian, 2003)



Source: (Krikorian, 2003)

Figure 14. System Assembly View Unification Results.

"The expansion step addresses the potential oversight by analysts in their understanding of the requirements and the subsequent allocation of behavior to subtiered elements. At this time in the process, analysts ask, "Is element Y really required to do A?" They also examine whether Y's expected behavior is exactly the same each and every time the use case sequences describe Y(A). This review uncovers issues and new requirements, and therefore addresses the problems caused by inadequate initial requirements. (Krikorian, 2003)

"The collapse step makes up for the specification of similar capabilities. At this time in the process, analysts ask, "Is all the functionality necessary? Are there really this many variants? Is there a common thread throughout the discussions that really highlights the intended functions?" This review removes similar functionality and therefore prevents over-specifying requirements." (Krikorian, 2003)

3.4.2.1 Element Descriptions

Now systems engineers review the output of the previous two steps (expansion and collapse) to determine the use cases for the next level of system decomposition. At this point in the process, the system engineering team has "discovered" known or expected behavior for lower-level elements. This lower-level behavior constrains the current element's internal-view descriptions. (Krikorian, 2003)

However, unless the analysis has reached a lowest-level component, the systems engineering team must still allocate and decompose the requirements and behavior across the next level of underling elements. This step thus determines and allocates the set of use cases that best describe the behavior of elements in the next level subsequent to that of the current element. (Krikorian, 2003)

The systems engineering team carries forward the resultant expanded and collapsed internal-view steps from the previous steps into the next-level subelements. These steps become the external view steps for the next-level subelement's use cases. (Krikorian, 2003)

3.4.2.2 Unification Benefits

These products provide a forum for capturing requirements and constraints that apply globally to an element. This step also provides a forum for capturing and resolving design decisions and issues. (Potts, 1998) It also forms the work package for element development and identifies functional, performance, timing, throughput, sizing, and other requirements that apply globally to the element. This unification and subsequent lower-level use case development continues for each level of system decomposition until the process reaches the lowest-level components. (Krikorian, 2003)

3.5 Benefits and Lessons Learned

Krikorian concludes that "up-front planning and buy-in to this disciplined approach is critical to the success of multidisciplined teams that engineer solutions. The disciplined approach described here has netted untold rewards in system understanding and insight far earlier in product development than with previous approaches. Using this approach, developers can distribute and allocate functions, interfaces,

operations, states, and modes of operation consistently onto the supporting subsystems." (Krikorian, 2003)

CHAPTER IV
SUMMARY AND CONCLUSIONS

4.1 Cultural Barriers to Implementing UML for Systems Engineering

The preeminence of the Unified Modeling Language (UML) as the prevailing standard for object oriented software specification has been has been noted by systems engineers and engineering managers for several years now. While progress toward extending UML for the systems engineering domain has been made, it is not clear that widespread systems engineering acceptance of UML is going to happen any time soon. (Steiner, 2003)

4.1.1 Barrier #1: Lack of Model-Driven Experience

UML, being a modeling language, imposes a model-driven approach to implementation. Regrettably, resistance can be overwhelming, particularly to an unsophisticated organization or customer. A pure model driven approach is rarely implemented by systems engineering, and few organizations can point to concrete successes. While bidding model driven systems engineering sounds great in the proposal phase, it is rarely bid properly. The learning curve is rather steep for an organization's first use by systems engineering (The same was true for initial use of OOA/OOD by software engineering a decade ago). As a result, the required front-loading of effort to successfully implement a model driven systems engineering effort is seldom anticipated or funded. (Steiner, 2003)

Steiner states that "a sophisticated customer will certainly appreciate the validity, adaptability, and integrity of a model driven systems engineering approach. A less sophisticated customer will probably become impatient with the level of rigor required up front "just to get the specification out." The power and elegance of a fully traceable, integrated and allocated system behavioral model will be lost on a customer who sees only potential delays in the start of coding. Managers forced to live with a meager budget and tight schedule (remember the cost proposal?) will gladly sacrifice the promises of a system model for the short term gain of meeting a delivery date, and getting this uneasy customer off his/her back. This is particularly true if the Systems Architect has not provided a constant, reassuring status of the

system modeling activity to management, or has failed to deliver on short-term milestones. Any delay in the early phase of a tight program will be projected into a big problem, and will be fatal to the system modeling effort!" (Steiner, 2003)

Our current systems engineering metrics that work well for managing document generation in current systems engineering processes do not translate well to model driven approaches. Specification page counts or requirement counts are not satisfactory for assessing the progress of system modeling. Instead, Steiner suggests than "new measures like requirement-behavior-component linkage (horizontal completeness), or enterprise-system-subsystem decomposition (vertical consistency) may need to be used to assess progress. A lack of historical trend date makes interpreting these new measures problematic, and will not inspire the necessary confidence when time and budget get tight." (Steiner, 2003)

Overcoming this first barrier must involve a resolute effort to characterize and measure model driven approaches, collect lessons learned, and educate program managers, systems engineers, and customer communities on their application and management. (Steiner, 2003) Unfortunately, these are all unknown until OOSE has been deployed.

4.1.2 Barrier #2: Unrealistic Expectations

The emergence of UML as the indisputable modeling language for object-oriented software development has caused some observers to consider it the answer for a wide variety of development process issues. To these observers, the use of UML as a language for systems engineering holds the promise of combining the systems and software engineering processes, resulting in what they predict "dramatic cost reduction and productivity gains!" This expectation is unrealistic, especially in light of the different objectives of systems and software engineering. (Steiner, 2003)

Even when the differing roles of systems and software engineers are clearly understood, the expected "smooth transition" associated with dual use UML remains. It is clear that a common language will certainly facilitate communication, but it is not clear that systems and software engineering will perform their tasks using the same model. (Steiner, 2003)

Since software engineering has seen productivity gains associated with using object-oriented techniques which uses UML as the modeling language, similar expectations of productivity gains will be expected when OOSE, which utilizes UML with extensions, is applied to systems engineering. Again, this is not realistic – the UML modeling language is not the root cause of software productivity gains. These are caused by mature processes and more efficient languages to develop code in such as Java. Nevertheless, the potential benefits of a model driven, UML facilitated OOSE process seem to capture the imagination of management. Steiner points to expected panaceas as "quick response to requirements changes, turnkey specifications from an instrumented system model, and the potential of "error free" systems engineering! While this hype provides good marketing and rationale for a lower cost bid, it sets up a very real barrier to realistic implementation!" (Steiner, 2003)

Steiner believes that "overcoming this second barrier requires a level of maturity that only comes with time and experience. OOSE offers a promise of real improvement in the allocation of requirements, and in bridging systems and software models. To date, this has been realized only through anecdotal evidence. Potential users must carefully consider the specifics of their application, and details of similar applications, before any claims can be made." (Steiner, 2003)

4.1.3 Barrier #3: Naïve Familiarity with UML

Steiner notes that "the OMG has consistently emphasized flexibility over elegance in the development of UML. The combination of notations in the original UML concept was never intended to represent a streamlined, minimalist language. UML was meant to be extended to embrace new methods and approaches, and relied on the tools (rather than the language) to enforce subtleties of method. The resulting language has notational constructs both dangerously obvious and deceivingly subtle! What could be simpler than sequence diagrams, state diagrams, activity diagrams, and swimlane diagrams? Every systems engineer can easily understand these concepts, and has probably used them already! Even class diagrams seem rather familiar to the systems engineer, once past the general notion of 'generalization' and 'aggregation.' But then things get more complicated – what does the 'black' diamond

mean? Use case diagrams can be difficult for an ‘old school’ systems engineer, and the distinction between ‘extends’ and ‘includes’ can be baffling!” (Steiner, 2003)

Since UML was designed to be the unifying modeling standard for OOA/OOD, it was not designed to support what most systems engineers need to produce. Steiner observes that “in theory, UML classes can be used to represent anything, even systems and subsystems that are not necessarily software. The expression of a system model can, in theory, be treated as an imperative, which the system software must meet. That is not what the notation was built for, however. The subtleties of class notation directly support expressions of software structures, like inheritance. Bending the use of the notation to express something as mundane as a parts tree or bill of materials looses much of the richness of the language, and is downright confusing to an experienced software engineer. UML ‘components’ do not easily map to system physical architecture and UML ‘interfaces’ do not easily map to system physical and functional interfaces. Building an interface specification from a UML compliant model can be a frustrating exercise for a systems engineer, and will undoubtedly require the use of some ‘custom’ stereotypes. Using the ‘includes’ relationship between Use Cases to express a functional hierarchy is inconsistent with the original intent of that relationship.” (Steiner, 2003)

4.2 Wrap up

In moving UML from supporting OOA/OOD to OOSE, we must note that systems engineering and software engineering are different disciplines with different objectives. Software engineers seek inventive ways to meet and refine requirements, most of which they perceive as reassuringly unchanging. Systems engineers seek innovative ways to validate and bound requirements, most of which they understand as very changeable. To software engineering requirements are the starting point while to systems engineering requirements are the product. While OOSE does not provide the definitive modeling language for systems engineering development, it is soon likely to become the standard. (Steiner, 2003)

REFERENCES

Baake, M., Architecture Lecture, July 2003, Texas Tech University

Cantor, M., Objected Oriented Program Management with UML, 1998, New York, John Wiley and Sons

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P., Object-Oriented Development – The Fusion Method, 1994, Englewood Cliffs, NJ, Prentice-Hall

DeMarco, T., Structured Analysis and System Specification, 1979, Englewood Cliffs, NJ, Prentice-Hall

Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., Object-Oriented Software Engineering: A Use Case Driven Approach, 1992, Boston, Addison-Wesley

Krikorian, H., "Introduction to Object-Oriented Systems Engineering, Part 1 and Part 2", March-April and May-June 2003, IT Professional periodical

Kruchten, P., "Architectural Blueprints – The '4+1' View Model of Software Architecture", November 1995, http://www.rational.com/media/whitepapers/Pbk4p1.pdf

Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design, 1997, Englewood Cliffs, NJ, Prentice-Hall

Leffingwell, D., and Widreg, D., Managing Software Requirements, 2000, Boston, Addison-Wesley

Maier, M.W. and Rechtin, E., The Art of Systems Architecting, 2002, Boca Raton, FL, CRC Press LLC, 2nd Ed.

Potts, C. and Burns, G., Recording the Reasons for Design Decisions, Proceedings International Conference on Software Engineering, 1998, IEEE SC Press

Sage, A.P., Introduction to Systems Engineering, 2000, New York, John Wiley and Sons

Steiner, R., "Cultural Barriers to Implementing UML for Systems Engineering", 2003, Proceedings of the 13th Annual International INCOSE Symposium

APPENDIX A
CONCEPT OF OPERATIONS FOR CRUISE CONTROL SYSTEM

When the vehicle is initially started, the cruise control system is off. The driver may at anytime later turn the cruise control system on by pressing the "on" button. Following turning the cruise control system on, the driver may activate the cruise control system by pressing the "set" speed button providing that the vehicle is going at least 30 (TBR) mph. When activated, the cruise control system will maintain a constant speed by accelerating and decelerating the engine. If the driver presses the brake pedal, the cruise control system will become deactivated thus no longer controlling the engine. The driver can later reactivate the cruise control system by pressing the "resume" button and the cruise control system will resume controlling the engine's speed and bring the vehicle back to the previously set speed. If the vehicle is going less than the previous set speed, the vehicle will accelerate at a high rate to resume to the set point. The driver may wish to accelerate to near the speed of the set point prior to pressing the "resume" button to avoid rapid acceleration. If the driver desires the cruise control system to be set at a higher speed, the driver must accelerate to the new speed and then press the "set" button. If the driver desires the cruise control system to be set at a lower speed, the driver must first slow the car to the new desired speed and then press the "set" button. The driver can turn off the cruise control system by pressing the "off" button and the cruise control system will not retain the latest set speed setting. The cruise control system will also be turned off when the engine is turned off.