1

IAPPING AN OBJECT-ORIENTED REQUIREMENT ANALYSIS TO A DESIGN ARCHITECTURE THAT SUPPORTS DESIGN AND COMPONENT REUSE

THESIS

Kelly L. Spicer
Captain, USAF

AFIT/GCS/ENG/90D-13

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/90D-13

MAPPING AN OBJECT-ORIENTED REQUIREMENTS ANALYSIS TO A DESIGN ARCHITECTURE THAT SUPPORTS DESIGN AND COMPONENT REUSE

THESIS

Kelly L. Spicer
Captain, USAF

AFIT/GCS/ENG/90D-13

1

AFIT/GCS/ENG/90D-13

MAPPING AN OBJECT-ORIENTED REQUIREMENTS ANALYSIS TO A DESIGN ARCHITECTURE THAT SUPPORTS DESIGN AND COMPONENT REUSE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

Kelly L. Spicer, B.S., B.S.
Captain, USAF

December, 1990

Acknowledgments

This thesis would not have been possible without the patience, sacrifice, and encouragement of my wife, Sally Jo. My deepest thanks go to her.

I also thank my God for giving me time off from his work to work on this thesis.

Important thanks also goes to the AFIT facility; especially my thesis advisor, Maj David Umphress for his guidance during my research; and my committee members, Maj James Howatt and Maj Patricia Lawlis for their helpful suggestions and critiques.

Important thanks also go to my mother, Laura Spicer, who took the time to make editorial comments and suggestions.

I also thank the Air Force Office of Scientific Research for sponsoring this thesis.

Finally, thanks go to my classmate Paul Hardy. His friendship and encouragement were invaluable to me during our thesis research time.

Kelly L. Spicer

Table of Contents

Page

List of Figures

List of Tables

AFIT/GCS/ENG/90D-13

Abstract

Design reuse has more potential for increasing the productivity of software development and maintenance than do traditional approaches to software reuse that emphasize reuse of smaller components. Current software development methods do not promote design reuse.

The literature contains limited documented research on the subject, but enough that some design reuse principles can be gleaned. Among these principles are that reusable designs should be applicable within some domain of application, have a consistent structure, provide a method for instantiating the design, avoid object nesting, and promote reuse of smaller components as well.

A design mapping method from an object-oriented requirements analysis to a design that follows the principles of design reuse is presented. The mapping method involves two transformation steps and introduces four representation tools for conducting the transformations. These tools are the Object-Mapping Table; the Hierarchical-Structure Diagram, which represents the static structure of the design; the Event-Mapping List; and the Object-Event Interconnection Diagram, a graphical representation of the Event-Mapping list to show the design dynamics. The second step transforms these representations into Ada specifications. Design templates are developed to aid in this transformation.

The design method is applied to two problems to demonstrate the consistent designs it produces. The first problem is then carried through to completion to demonstrate its feasibility and ease of implementation.

MAPPING AN OBJECT-ORIENTED REQUIREMENTS ANALYSIS TO A DESIGN ARCHITECTURE THAT SUPPORTS DESIGN AND COMPONENT REUSE

I. Introduction

Software reuse is a much touted solution to the software crisis. But is it working? Are we using the right methods in our quest for increased productivity through reuse? Are we approaching this problem from the right level? This thesis addresses these questions, suggests some answers, and presents a method for achieving greater reuse potential and benefits through design reuse.

1.1 Background–Problems with Current Reuse Approaches

A common complaint by the computer community is : "Current approaches to software reuse have not lived up to reusability's potential to dramatically improve software productivity and maintainability" [Kaiser and Garlan, 1987:pp17]. There are many reasons software reuse is not more common; they fall into a number of categories: technical, managerial, contracting, etc. The concentration in this thesis will be on technical impediments. In particular, it focuses on the problem of a general emphasis on reuse at the "small" level and that common software design methodologies do not support reuse.

1.1.1 Too Much Emphasis On Reuse in the Small. Biggerstaff and Richter pointed out that reuse of larger software components leads to larger reuse payoffs [Biggerstaff and Richter, 1987:pp42]. Tracz's paper concludes that "most programmers tend to view reusability from the perspective of simply reusing code when reusing other programming artifacts (designs, specifications, and tests) leads to a more productive [development] environment" [Tracz, 1986:pp175].

Biggerstaff and Richter contrasted the differences in code reuse versus design reuse in their paper on reusability directions. They pointed out that the payoff for reuse of software modules quickly reaches a ceiling that is difficult to surpass. Design ideas are often much more reusable than software modules. They complained that no method exists for representing designs, unlike code, which is represented in high-order programming languages. They suggested that design reuse is the only way we can come close to an order-of-magnitude increase in productivity or quality. They listed as "Research Issues" properties that design representations need to exhibit. Among these are what they called "partial specifications." These are "partial architectures" or "partial structures" which they say are "highly reusable, but the details of these typically are not." They must be filled in by the particular implementation. They concluded by saying that "Design Reuse has the greatest potential leverage [to increase payoffs in reuse], but significant representational breakthroughs are needed to realize its full potential." [Biggerstaff and Richter, 1987:pp42-48]

1.1.2 Current Methods Do Not Support Reuse. Tracz pointed out that one of the reasons programmers do not reuse code is that "there are no software development methodologies that stress reusing code, let alone reusing a design, or a specification" [Tracz, 1986:pp172]. The Command and Control Systems Office (CCSO) also reported the lack of a standard design methodology that supports reuse [CCSO, 1988:pp15].

Proponents of Object-Oriented Design (OOD) claimed it to be the methodology able to make reuse practical [Meyer, 1987:pp53]. The principle of OOD is that a system's modular decomposition is based on objects from the problem and solution space, rather than on the functions performed. This allows modules (objects) to be classified and grouped into classes for the purposes of reuse [Kaiser and Garlan, 1987:pp55-63]. OOD is based on good principles and works well for programming in the small. For example data structures, device controllers, devices simulators, etc., can be defined and grouped in a fairly natural way [Booch, 1987:pp45-112].

Apparently, the dilemma is a general lack of a method for applying OOD for larger systems development. Probably the most commonly referenced method for applying OOD is Booch's method of "Developing an Informal Strategy," and then "Formalizing the Strategy" [Booch, 1983:pp38-44,71-79,130-143]. Booch credited Abbott with the idea. The method relies on using natural-language descriptions as a first step toward designing systems. The problem with the strategy, in terms of large scale reuse, is that it will create unique solutions for each problem. Just as different individuals will describe problems and solutions differently, so will the solutions based on these narrative statements be different. The resulting solution is designed for a particular problem, not a class of problems. A similar analysis could be made for other commonly used development methodologies.

Reuse in the large is needed to achieve large gains in software development productivity [Biggerstaff and Richter, 1987:pp46-48] [Tracz, 1986:pp175]. A method of designing software that solves classes of problems is needed to achieve these gains. Such a method would systematically develop similar solutions for similar problems.

1.2 Problem Definition

The objective of this thesis is to develop a method to map from object-oriented requirements definitions to designs that exhibit principles consistent with design reuse. This mapping method includes tools for design representation that demonstrate consistency with these principles, are systematic, and create similar designs for all problems within the application domain. Specific objectives of this thesis are the following:

a. Determine the principles on which design reuse should be based.

b. Define design representation tools needed to represent designs consistent with the above principles.

c. Define steps, using these principles and tools, for developing consistent designs with potential for reuse.

d. Validate the mapping method by applying it to two problems and comparing the results.

1.3 Scope

This thesis discusses problems and solutions for classes of problems mostly within the scope of embedded-Ada applications (for which Ada was initially developed). The principles discussed, however, certainly apply to a much broader range. This scope excludes discussion of approaches to solving classes of problems used by 4th generation languages and code generators.

1.4 Approach and Overview

This research consists of five phases:

1. An investigation into current methods of design reuse under research.
2. A case study of a particular design-reuse approach to glean the principles used.
3. The definition of a mapping method from an object-oriented requirements definition to designs that exhibit these principles.
4. First validation step - apply the mapping method to two problems and compare results.
5. Second validation step - implement one of the resulting designs to demonstrate its usability.

1.5 Maximum Expected Gain

The maximum expected gain from this research is as follows:

- An awareness of the importance and potential of reusing at the design level.
- An awareness of some of the research being done in the area of design reuse.
- An appreciation of the importance of consistent designs, at least within an application domain. This is a first step to design reuse.
- A useful mapping method from an object-oriented requirements definition to designs that are consistent, for embedded event-response applications.
- Further direction in this area through suggestions of follow-on research.

1.6 Sequence of Presentation

The remaining chapters of this thesis follow the phases of research discussed in Section 1.4, Approach and Overview, as described below.

Chapter II lays the foundation of the thesis by reviewing the current literature on design reuse and object-oriented requirements analysis. The chapter starts by discussing design reuse terminology, characteristics and benefits of reusable designs, and how domains for reusable designs are defined. Design structures are then categorized to provide context for further design discussions. Existing research and projects that emphasize reuse of designs are then presented and discussed. Next comes an enumeration and discussion of the ways reusable designs can also support smaller-component reuse. Object-oriented requirements analysis (OORA) is then presented and contrasted with other forms of requirements analysis. OORA concepts and tools are then presented and discussed.

Chapter III develops the method of mapping from an OORA to design. The foundation for the mapping method is first further developed with detailed descriptions of March's OORA method and the OOD Paradigm design method. Benefits and advantages of using an OOD-Paradigm-like reusable design are then listed and described. The mapping method itself is then described. It maps from March's method to a design following OOD-Paradigm principles. Both major steps of the mapping method are described in order: first a mapping from the analysis to design representation tools developed in this thesis, then a mapping from these representations to Ada specifications.

Chapter IV validates the design mapping method by applying it to two different problems. The primary objective is to demonstrate how the method develops similar designs for different problems. This chapter applies the first step of the mapping method to the two problems; the second step, mapping the results of the first step to Ada specifications, is included as appendices A and B. The design results are discussed in an analysis section that compares the results to the list of benefits and advantages of reusable designs developed in Chapter III. Suggestions are then made for the implementor of the designs. As an additional validation step, one of the designs is implemented and the results are discussed; the implementation code is included as appendix C.

Chapter V Summarizes the contribution of this thesis, suggests and discusses follow-on research that needs to be done, and makes some hindsight comments on the use of object-oriented requirements analysis.

II. Literature Survey

2.1 Introduction

This chapter lays a foundation for the idea of designing to solve classes of software problems instead of individual problems by reviewing the literature on this and related topics. Terminology is a problem: what do you call a design solution intended to solve more than one problem in an application domain? Many authors call them "reusable architectures" so this term will be adopted for this chapter. Because the term *design* is more widely understood than *architecture* when describing software, later chapters use the term *reusable design* instead.

Some definitions of reusable designs, as presented in the literature, are the lead topic of the chapter. Domain analysis is then briefly discussed since it deals with analyzing classes of problems instead of individual problems. Design structures are categorized to provide context for discussing different kinds of reusable designs. Some actual reusable designs and designs that apply reusable design principles are then presented. Because component reuse is thought to be an important side benefit of reusable designs, characteristics of a design that would support component reuse are discussed. Object-oriented requirements analysis is then discussed to lay a foundation for the mapping method presented in Chapter III.

2.2 Definition of a Reusable Architecture

Parnas discussed the idea of reusable architectures in his 1976 paper on program families. His context was multiple release software systems where significant differences exist from one release to another. He said that multiple releases are generally conducted by first making one complete and working release, then following releases are made by modifying the first release. He proposed the idea of forming a baseline at some stage of partial design. This is the point where all releases share a common design but will now diverge to satisfy their differing requirements. Having a baseline point of partial design is a similar idea to that of reusable architectures. The common partial design is reused for each different release. [Parnas, 1976:pp1-3]

No formal definition of a reusable software architecture can be found in a dictionary or textbook. It is an approach to software building called by different names by different people. For instance, Brown and Quanrud called then "Generic Architectures" [Brown and Quanrud, 1988:pp390], Richard D'Ippolito called them "Models" [D'Ippolito, 1989:pp256] but his research team (the Software Architectures Engineering Project (SAE) at the Software Engineering Institute (SEI)) used to call them "Paradigms" [Rissman and others, 1988:pp1]. D'Ippolito explains in his paper that they changed their terminology from "paradigm" to "model" and more recently to "structural models." Ted Ruegsegger called them "generic functional architectures" [Ruegsegger, 1988:pp16]. Batory, Barnett, Roy, Twichell, and Garza called them generic architectures and defined them as follows: "An architecture is a template in which building-blocks can be plugged. Interfaces are standardized to make blocks interchangeable" [Batory and others, 1988:pp1].

2.3 Characteristics and Benefits of Reusable Architectures

Based on the literature cited later in this chapter, reusable architectures can be characterized as follows:

1. Applicable within a problem/application domain.
2. Has a consistent structure.
3. Provides a general solution to a class of problems.
4. Provides a method for instantiating specific solutions.
5. Designed to promote reuse of components.

A reusable architecture is a template for solving problems within an application domain. A reusable architecture includes a design structure as a minimum. It may also include a set of templates for instantiating the design as in the SEI's OOD-Paradigm [Rissman and others, 1988:pp53], or a set of reusable subcomponents applicable within the domain and designed to be used in the architecture as with the RAPID architecture [Brown and Quanrud, 1988:pp390].

Richard D'Ippolito, of the SEI's SAE project, who called reusable architectures "Models," defined them as follows:

> Models are general solutions to problems and provide reuse at the design level because they are expressed in a reusable, adaptable form. The adaptability is provided by parameterizing the services so that they may be scaled to fit a particular application of the model. This requires the model to present to the designer a clear sense of the general problem the model solves and to present to the implementor the means to create an instance of the model that solves the specific problem. In theory, the model represents captured science, and the parameterization is what allows the engineer to apply the science to a specific problem. A good model will be capable of being applied to the expected range of applications with no change to its working structure and with predictable performance. [D'Ippolito, 1989:pp258]

D'Ippolito also quoted Baber as saying the following:

> Especially noteworthy is that the engineer employs a scientific, theoretical foundation to verify – by systematic calculation before the object is actually built – that a proposed design will satisfy the specifications. [D'Ippolito, 1989:pp257]

D'Ippolito was saying that a model solution is the tool the software developer needs to accomplish this – he would then be functioning as an "Engineer."

2.4 Domain Analysis

The source of information needed to create a standard architecture is certainly some type of domain analysis. Like reusable software architectures, domain analysis has a variety of interpretations. Berard said the reason for conducting an analysis is to identify reusable items within the domain, that is, items which may be reused to build multiple applications within the domain. He defined "domain" in domain analysis to mean "application domain," for example, graphical user interfaces, embedded missile applications, decision support systems, etc. [Berard, 1990a:pp4].

Prieto-Dĭaz made the following statement on the goal of domain analysis:

> ...we try to generalize all systems in an application domain by means of a domain model that transcends specific applications. Domain analysis is thus at a higher level of abstraction than systems analysis. In domain analysis, common characteristics from similar systems are generalized, objects and operations common to all systems within the same domain are identified, and a model is defined to describe their relationships. [Prieto-Dĭaz, 1987:pp347]

Brown and Quanrud stated that a domain analysis is conducted to establish the scope of the domain. "It [the domain analysis] should identify the requirements that are common to the applications of the domain ... " [Brown and Quanrud, 1988:pp391].

In terms of what should be included in a domain analysis, Brown and Quanrud also stated the following:

> The domain analysis must also include the development of the preliminary design for the architecture. The design must be specified to some level of detail in order to know whether it can be shared by all of the applications within the scope of the domain. Applications that cannot share a common design cannot be included in the same domain. Thus, the design plays a critical role in determining the scope of the domain itself. [Brown and Quanrud, 1988:pp391]

Thus, the expected products of domain analysis vary depending on the writer. Brown and Quanrud were looking for a high-level design; Prieto-Dĭaz was looking for reusable library components, domain standards, and reuse guidelines. Prieto-Dĩaz conceded that no methodology or formalization is currently available for the domain analysis process or products, and that most current interest is on the products more than on the process of obtaining them [Prieto-Dĩaz, 1987:pp347].

2.5 Categories of Architecture Structures

Most design hierarchies represent some type of layered architecture. The difference between types is in how the layers are abstracted. We will see that the layers may be viewed as successive "virtual machine" layers (the seniority hierarchy), or successive decompositions of, or compositions to, a high-level context diagram (the composition hierarchy).

For object-oriented systems, Seidewitz referenced Rajlich in defining two orthogonal hierarchies: the composition hierarchy and the seniority hierarchy. Seidewitz pointed out: "The composition hierarchy deals with the composition of larger objects from smaller component objects. The seniority hierarchy deals with the organization of a set of objects into 'layers.' Each layer defines a virtual machine that provides services to senior layers [Seidewitz, 1989:pp97]." Rajlich said a common misconception is to equate these two hierarchies. Rajlich actually called the composition hierarchy the "parent-child hierarchy," since he was looking at them from a top-down perspective. He said that this hierarchy deals with the decomposition of larger packages into smaller packages. [Seidewitz, 1989:pp97-102] [Rajlich, 1985:pp719]

Sidewitz pointed out that the composition (parent-child) hierarchy can be directly expressed by leveling diagrams (much like with data-flow diagrams as Pressman explained the principle originated by Yourdon, Constantine, Riggs, Gane, and Demarco [Pressman, 1987:pp166-172]). The top level is like a context diagram in the sense that it can show an entire system interacting with external objects. Successive diagrams are then produced decomposing the components into children at each

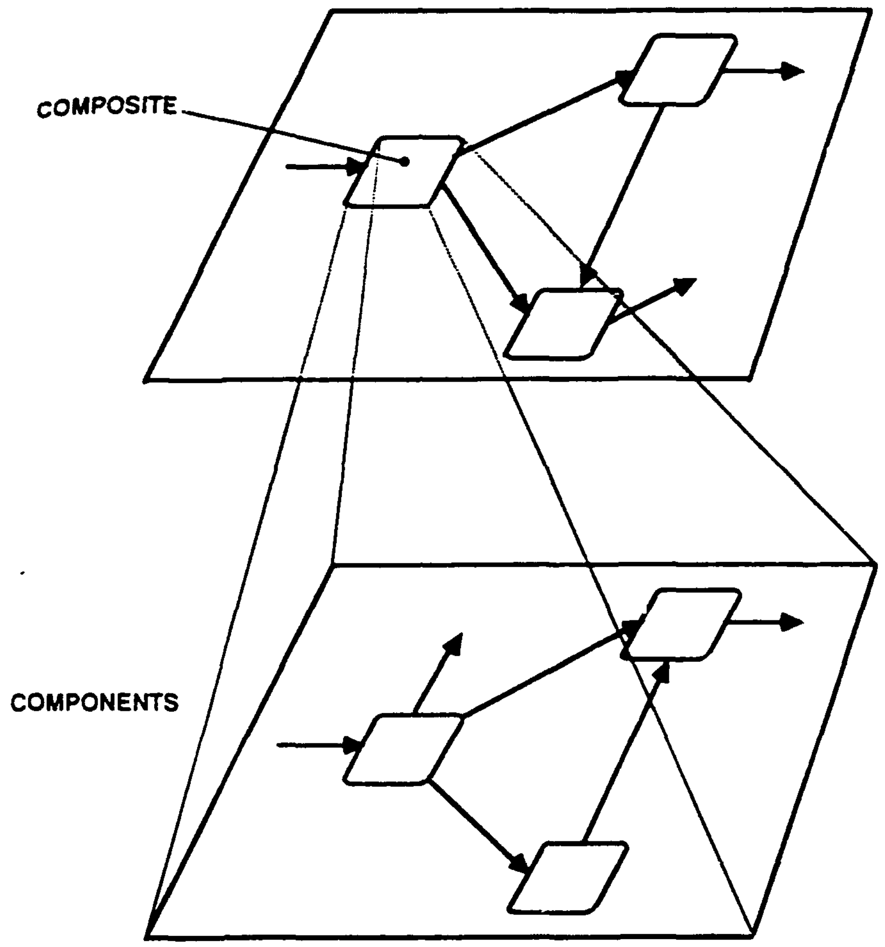

Figure 2.1. Composition Hierarchy
[Seidewitz, 1989:pp98]

level (Figure 2.1). Finally, at the lowest level, objects are completely decomposed into primitive objects such as small subprograms and state/data stores. [Seidewitz, 1989:pp97-98]

Conceptually the composition architecture is a tree. In another paper, Rajlich called the composition hierarchy a "system tree" [Rajlich, 1984:pp192]. The seniority hierarchy is not necessarily a tree; it can be expressed by a single diagram with the different virtual layers separated by horizontal lines (Figure 2.2). Using the convention of Rajlich, senior layers call junior layers, but not vice-versa (Ada's exception propagation is an exception to this rule) [Rajlich, 1985:pp719]. In Ada senior layers "with" junior layers. Senior layers treat the junior layers as a set of primitive operations in an extended language. Each junior (virtual) layer is designed using the principles of abstraction and information hiding. That this architecture need not be a tree is seen by recognizing that two modules in a senior layer can call the same operation in a junior layer.

An advantage of the seniority hierarchy is the reduced coupling it encourages between layers. Junior layers know nothing about the senior layers calling them, and senior layers do not need to know how junior layers accomplish their work, they see only their interfaces.

Designs resulting from a Structured Analysis and Design Technique (SADT) approach generally would be a composition hierarchy. The concept of top-down leveling used by SADT results in the parent-child structure.

2.6 Survey of Existing Reusable Architectures

2.6.1 SEI's "Structural Model" Solutions Overview. Much work in the area of standard architectures has been done by the the SEI's SAE Project led by Richard D'Ippolito. Their foundational effort appears to be their 1988 Report: "An OOD Paradigm for Flight Simulators, 2nd Edition," [Rissman and others, 1988:pp1-120]. It described the design for a reusable architecture for the flight simulator domain, and applied it to the engine "system" of a flight simulator. They have also released a follow-on report that applied the architecture to the electrical system of a flight simulator at the "system" level of the architecture [Rissman and others, 1989a:pp1-166].

The flight-simulator architecture has also been applied to the Millimeter-Wave Seeker Missile Under the Ada Shadow program by Hercules Defense Electronics, [D'Ippolito, 1989:pp261,264]. The SAE project team has also helped Space Command develop a "structural model solution" (See Section 2.6.3).

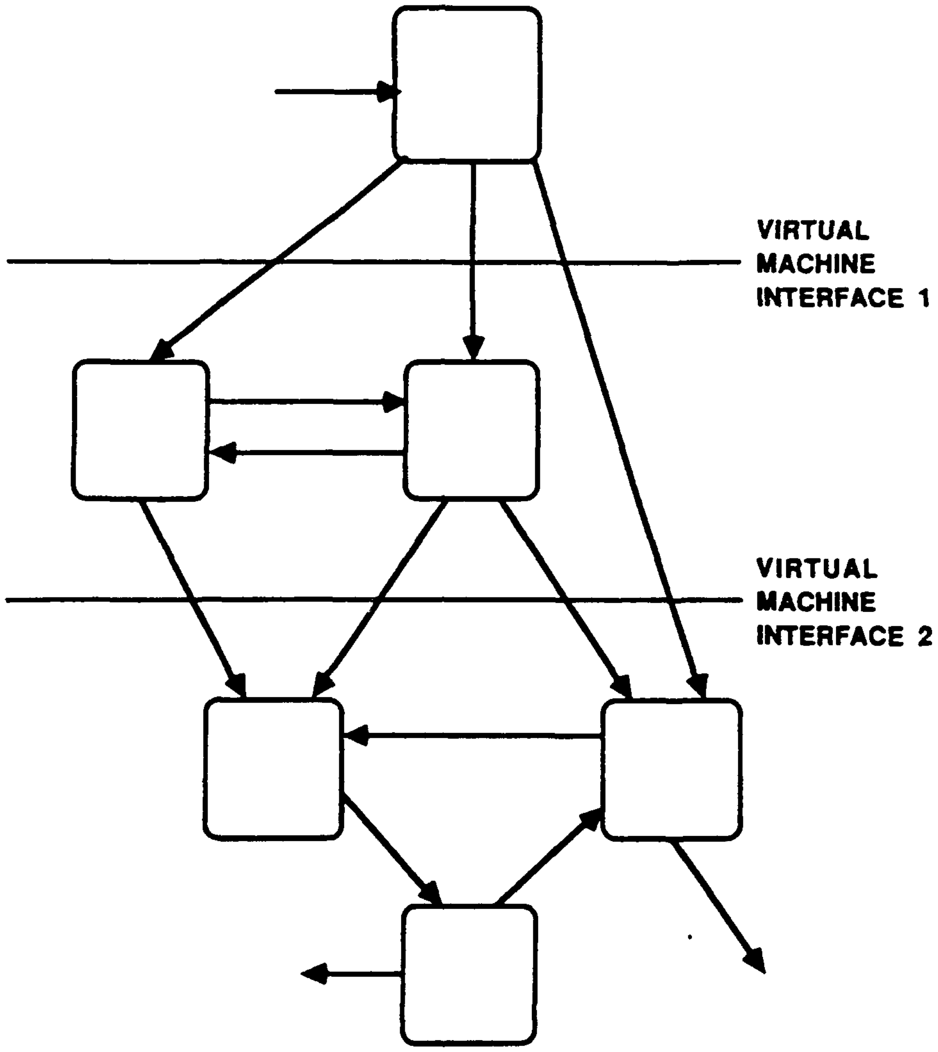

Figure 2.2. Seniority Hierarchy
[Seidewitz, 1989:pp99]

2.6.2 SEI's OOD-Paradigm for Flight Simulators.

The OOD-Paradigm report presented a unique architecture. This architecture has characteristics of both the composition hierarchy and the seniority hierarchy, with some unique characteristics of its own. It is similar to the composition hierarchy in the sense that each of the major software units ("executives") is decomposed through two sublevels. It resembles the seniority hierarchy in the sense that all the decompositions follow a parallel layering pattern (Figure 3.1). In the implementation the decompositions are articulated as compositions: each higher level is actually an aggregate of the lower-level components (that it was decomposed to during analysis). Higher levels are implemented as data structures containing instances of the lower-level components. [Rissman and others, 1988:pp7-10]

The OOD-Paradigm's design team started the project with two basic goals: eliminate nested implementations of objects (rationale discussed in Section 3.4.4) and simplify dependencies among objects. Notable characteristics of the architecture are the following:

1. The architecture consists of logical layers replacing the usually nested objects found in a composition hierarchy. Though the control flow follows the hierarchy, data generally flows across the hierarchy, that is, data may pass directly between different major software units without going up and down the hierarchy.
2. Templates are used for instantiating the architecture for new applications within the flight-simulator domain.
3. Connectors are used to connect objects. This reduces coupling and renders the objects themselves more reusable since there are no direct dependencies (Ada "with"ing) between objects.

2.6.3 Granite Sentry Command and Control System.

The Air Force Space Command's Granite Sentry Program at Peterson AFB is using the SEI's "model solution" approach for a C^3I application. They have consulted with The SAE Project team at the SEI for help in developing their model solution. The application is a multi-phase upgrade and replacement of the NORAD Computer System (NCS) and the Modular Display System (MDS) in the Cheyenne Mountain Complex of the North American Aerospace Defense Command (NORAD). Granite Sentry is implemented using the Ada language. [Goyden, 1989:pp40]

The message translation and validation part of the system was implemented using the SEI's model solution for message handling. Many types of messages must be handled (about 60 kinds, for example, air-related messages, missile-related messages, sensor messages, etc.). The recurring problems were identified and templates were developed for implementing instances of the message types. The reusable architecture is represented by the hierarchy of components needed to be combined to instantiate a message type (Figure 2.3). This same solution has been used for other C^3I applications [Rissman and others, 1989b:pp1] [Goyden, 1989:pp43-50].

Granite Sentry's design methodology follows the principles recommended by the SAE Project at the SEI [Rissman and others, 1989b:pp56-67]. The project team utilized the structural-model concept to employ an incremental, depth-first software development process. First they were able to speed the design phase by abstracting the details of how different messages types are handled. Then they developed a working prototype that fully implemented the handling for one message type. This was then used as a model to develop the templates so software for the other message types could be instantiated. They found that this approach resulted in higher productivity (34 Ada lines per programmer-day), less documentation, less test effort, and more efficient reviews than traditional methods. [Goyden, 1989:pp49]

2.6.4 RAPID. Another research project using reusable designs is the Army Information Systems Engineering Command's RAPID project (Reusable Ada Packages for Information System Development). The application is a management infor-

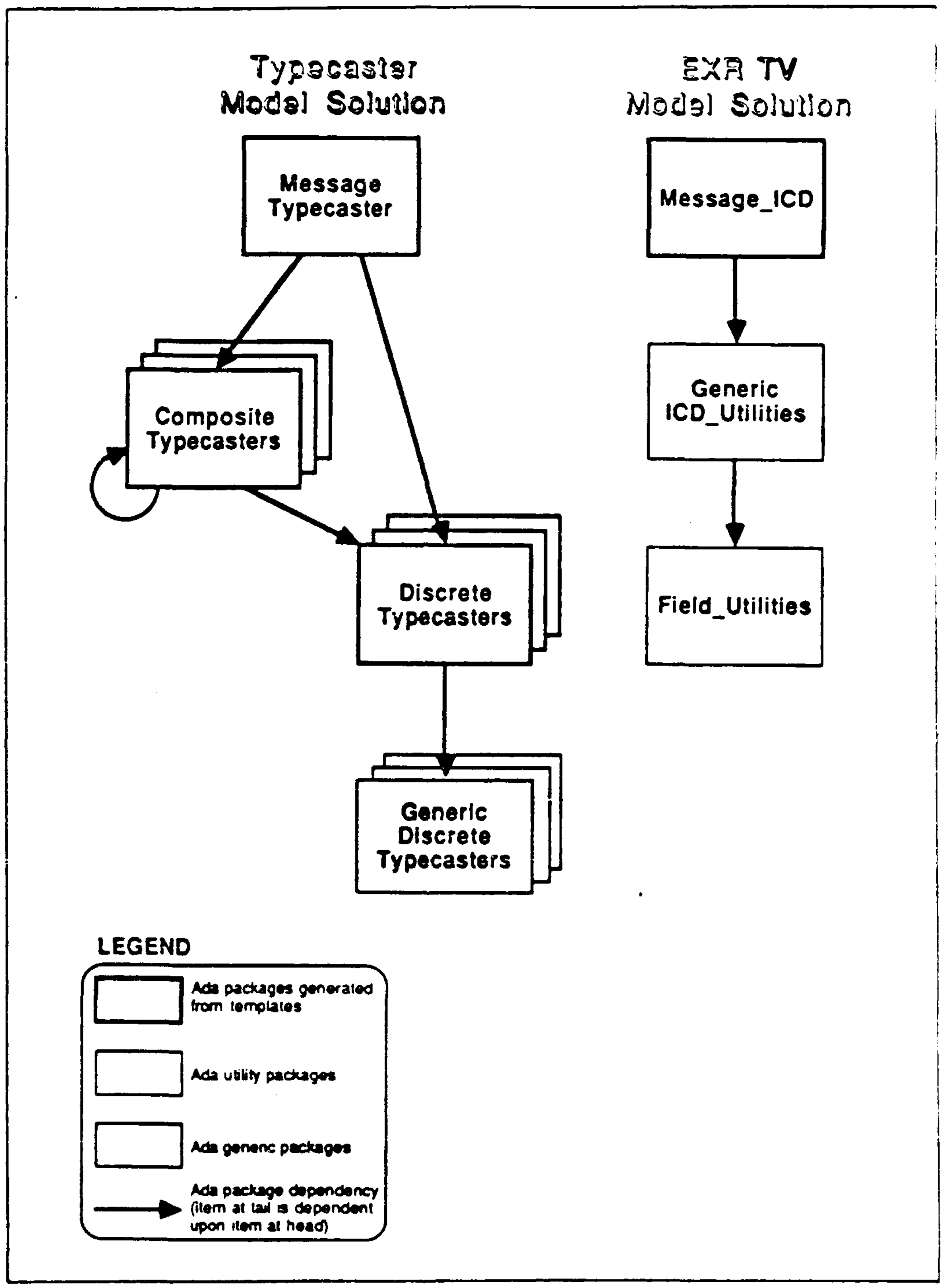

Figure 2.3. Granite Sentry Model Solution Architecture [Rissman and others, 1989b:pp45]

mation system. The commonality basis for the architecture is resource management. [Ruegsegger, 1988:pp16]

The RAPID architecture is a classic functional decomposition and forms a tree (Figure 2.4). Reuse of the architecture is achieved through the use of Ada generics in the leaf modules. These modules are instantiated for the "discrete resource of interest" (like ammunition, repair parts, personnel, blood products, etc). Among their findings Ruegsegger reported the following:

> The generic architecture is a concrete implementation of the concept of design reuse. The method devised for transforming SADT models to Ada PDL provides a clear link between the two disciplines and preserves all design decisions embodied in the original architecture model.
>
> The development and use of generic architectures fosters the establishment of standard architectures and promotes the habit of designing for reusability. The generic architecture includes commonly applicable types and operations, and it identifies for the developer those that need to be defined. This has long-term benefits in the form of consistent, more easily maintainable software architectures. [Ruegsegger, 1988:pp22]

2.6.5 University of Texas DBMS. [Batory and others, 1988:pp1-12]

Reusable architecture research has also been done at the University of Texas on a DBMS. Using E-R (Entity-Relationship) modeling, they have developed an architecture and a set of building blocks for interfacing the architecture components using standardized interfaces. They claimed to be able to assemble a file-management system in minutes that otherwise would take man-years of effort and hundreds of thousands of dollars using traditional methods. Their basis of commonality is file-management systems. Their method of customization is "include commands" in their C compiler; needed modules are included and unneeded modules are left out.

2.6.6 Kiem's Keystone Methodology. [Kiem, 1989]

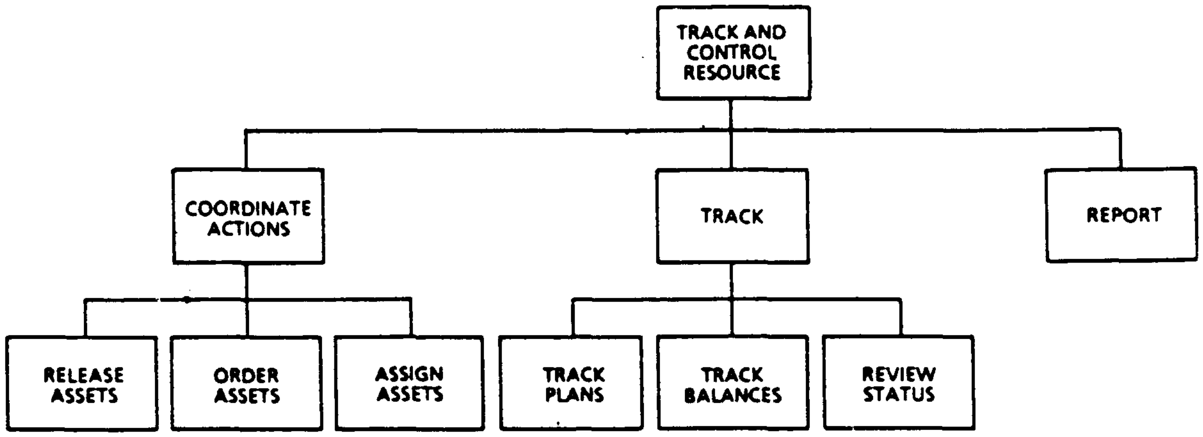

Figure 2.4. RAPID Hierarchy
[Ruegsegger, 1988:pp18]

Kiem's Keystone Methodology is included not because it is a reusable design but because it develops designs that follow some of the principles supported by this thesis: use of intermediary components to eliminate direct "with"ing between problem-space objects (to enhance component reuse), and higher-level components that are actually aggregates of lower-level objects to flatten the architecture and increase efficiency. Here is the abstract of Kiem's paper:

> The Keystone Methodology uses Entity-Relationship modeling to determine an optimum object-oriented packaging structure, which will exhibit minimum coupling and inter-dependencies between elements of a system and therefore maximum reusability potential. Furthermore, the resulting organization of the data dimension permits extensive use of a limited range of generics to provide complete data manipulation through the use of relational operations. The form and disposition of concurrent elements of a system can also be determined directly from the E-R model. The modeling process is proven and the implementation of the resulting design is systematic.

The Keystone methodology develops a design structure that is a composition

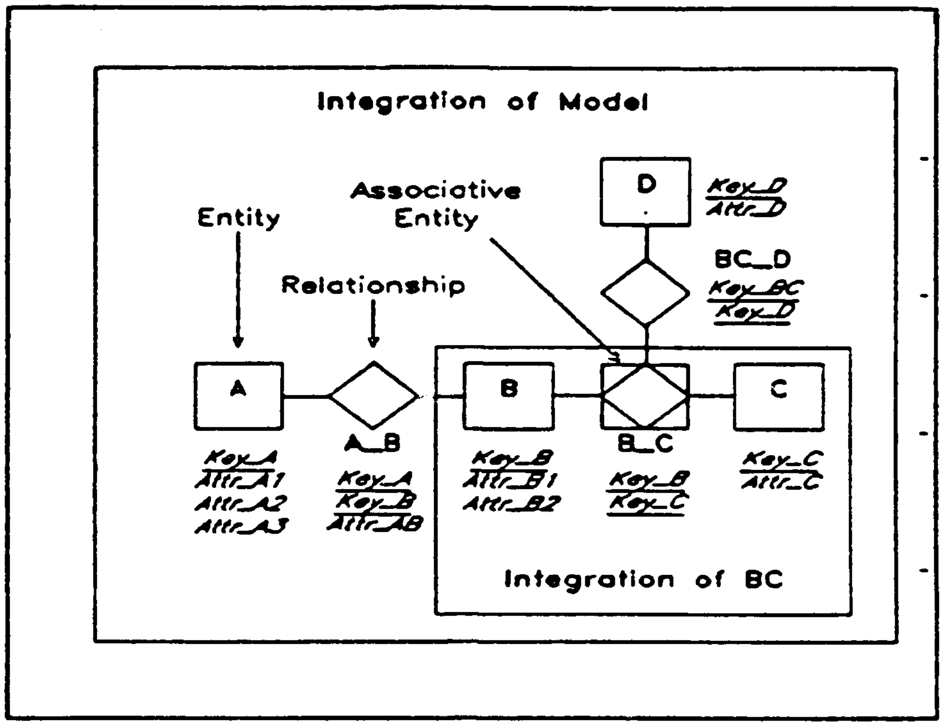

Figure 2.5. Example of an Entity-Relationship Model
[Kiem, 1989:pp101]

hierarchy. The designs are similar to the OOD-Paradigm in the way that higher-level components are actually aggregates of instances of lower-level components. The Keystone methodology uses E-R modeling to identify the components and aggregates. Components are from the problem space, aggregates are the relationships between the components as identified by the E-R modeling (See Figures 2.5 and 2.6).

The method is proven in the sense that it has been applied to the development of a now-fielded Air Force system: SARAH-lite. SARAH-lite was developed at the Command and Control Systems Center at Tinker AFB, using a Rational^{TM} environment. The implementation runs on Zenith personal computers. The application is a message preparation workstation.

Kiem claimed the method is systematic, which he said is contrary to traditional OOD (informal strategy - formal strategy described in Section 1.1.2).

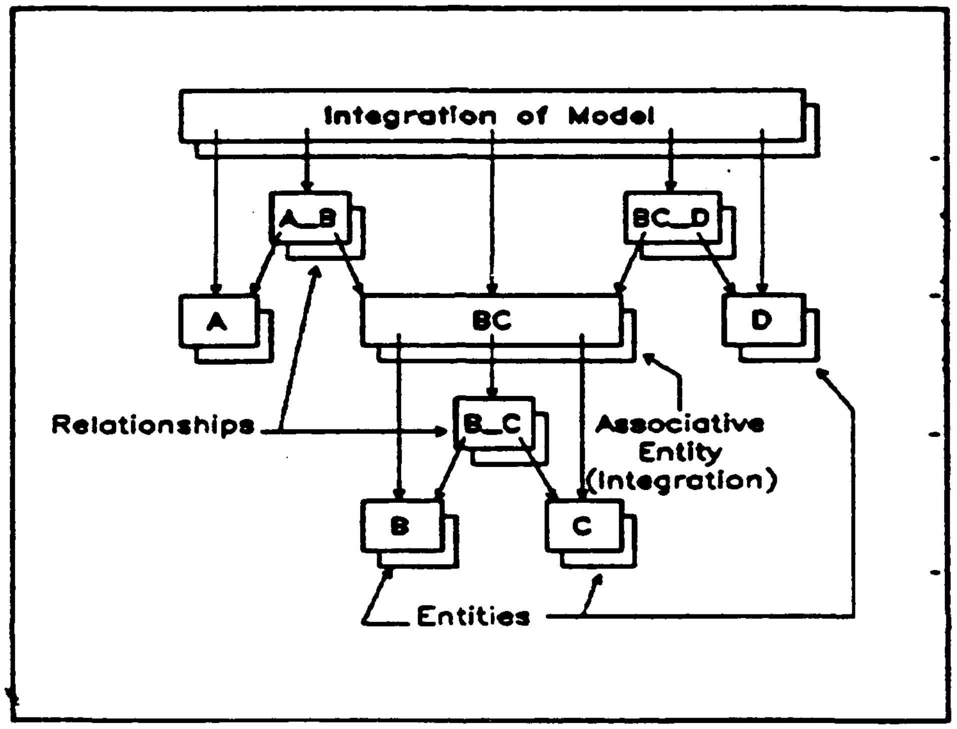

Figure 2.6. The Keystone Packaging Schema for the Previous Figure [Kiem, 1989:pp102]

2.7 Characteristics Needed to Support Component Reuse

In addition to supporting design reuse, the design of a reusable architecture should also consider supporting reuse at the component level. Actually, many of the concepts of reusable design support component reuse as a side benefit.

Two kinds of component reuse are at issue:

1. "Swapping out" components in an implementation.
2. Reusing components between implementations.

The first kind would be useful for trying different components that perform the same function, but with somewhat different characteristics. For example, an electronic combat model may have several candidate radar units that model corresponding alternate choices for radar components in the actual radar; the software objects that model these radar components could be swapped out to compare effects on performance.

The second kind is the conventional type of reuse we usually think of. This would normally be intra-domain reuse. The SAE team made the following comment on intra-domain reuse:

> Flight simulators provide natural opportunities for reusing software. First, different aircraft have the same kinds of components, e.g. engines, fuel systems, electrical systems, etc. [Rissman and others, 1988:pp4]

These design characteristics are discussed in the following sections.

2.7.1 Object Oriented.

This discussion brings to light an important characteristic needed for the software components to be reusable: they should model their real-world counterparts when possible. To do this the architecture should represent an object-oriented design. Richard St. Dennis at Honeywell pointed out that for a component to be reusable it should represent an object-oriented mapping of the problem to the solution; that is, the software solution represents the human view of the original problem [St. Dennis, 1987:pp515].

St. Dennis continues with the following:

> Reusable software should act on objects explicitly. What we are advocating here is a clear definition and method of 'acting' on objects. All actions or operations on objects should be defined as subprograms (or their equivalent) with the objects as parameters. Furthermore, the objects, or at least their types, should be 'packaged' as close to the definition of the operations on them as possible. It is better not to use global data that is changed implicitly by routines to which it is visible but to pass the data to routines as parameters making it explicit that these routines are actors/operators on the data and this is just how this data will be treated (e.g. as input only, as a constant, and so forth). [St. Dennis, 1987:pp515]

2.7.2 Explicitly Defined Purpose/Function. Each object must have an explicitly defined purpose. SofTech pointed out that each component intended for reuse should implement a single well-defined function. The scope within which the component is to be used and the degree of generality should be clearly stated. [SofTech, 1985:pp28]

2.7.3 Independent Objects. Low coupling and high cohesion is the key here. Coupling must be kept to a minimum for components to be moved or exchanged.

Examples of design methods which lead to low coupling are the OOD-Paradigm and Keystone methods (both presented earlier). Both use intermediary modules to entirely remove direct coupling between software objects.

The CCSO pointed out the compromise that often must be made between coupling and the use of system-wide tools for standardization. Although they used an object-oriented design, their components were not reusable due to dependencies on global tool packages. They standardized on the use of tools for buffer management, linked-list manipulation, I/O, etc. Because of this, components were not reusable since they depended on all these tools and associated types [CCSO, 1988:pp11-12].

Hardware, operating system, and compiler dependencies also must be kept to a minimum. Of course, some software components require these kinds of dependencies due to the nature of their function. Both Brown and the CCSO pointed out the need to isolate these necessary dependencies. The amount of components with these kinds of dependencies should be minimized; they should be clearly labeled and perhaps grouped into a kernel layer. [Brown and Quanrud, 1988:pp391] [CCSO, 1988:pp10]

2.7.4 Layered Architecture. The principle of designing at the higher levels while abstracting the details of lower levels is fundamental in software design. D'Ippolito at the SEI reminded us that software designers quickly get bogged down in too much detail if they do not follow this principle [D'Ippolito, 1989:pp258]. This is an important principle for reuse as well; the objective is to reuse at a high level without the necessity of concern about how the low-level components accomplish their tasks.

SofTech also pointed out that a layered architecture contributes to reusability by giving us reuse levels. Different concerns can be separated into discrete layers that can be separately replaced or tailored without affecting other layers [SofTech, 1985:pp22]. Ideally, the layers should decompose just as the problem space or real-world object would. In the SEI's OOD-Paradigm report, a middle layer is an engine mapping and the components on the next level down represent real-world engine components [Rissman and others, 1988:pp19].

2.7.5 Standard Interfaces. The University of Texas applied this principle of standard interfaces in their approach to generic architectures. They said the following:

> Every object of an architecture is associated with a distinct class of modules. All modules are plug-compatible (for interchangeability), and each module is a different implementation of the object.
>
> Declaring an ad hoc interface to be a standard is the worst of all possibilities. A better way is to 1) identify the class of implementations that are to be supported, and 2) design the simplest interface that supports all implementations of the class. The greater the number of implementations, the more likely it is that the interface *captures fundamental properties of the object*. Such an interface is no longer ad hoc because it is justified by its demonstratable generality. We call this the simplest common interface (SCI) method of standardized interface design. [Batory and others, 1988:pp3]

2.8 Object-Oriented Requirements Analysis

The foundation of the design phase, where the reusable designs are constructed, is the requirements analysis phase which precedes it in the software development lifecycle. The requirements analysis phase is very important as the design is derived from products of the analysis. In preparation for the mapping method presentation in the next chapter, which maps an object-oriented analysis to design, this section overviews object-oriented requirements analysis. In addition to this, March's analysis method is discussed in more depth in Section 3.2, and some hindsight comments on the use of object-oriented analysis are given in Section 5.3.2.

Object-Oriented Requirements Analysis (OORA) is an approach to requirements analysis that moves the introduction of object-oriented techniques to an earlier phase in the software lifecycle. An object-oriented approach is thought to map more naturally to an object-oriented design than do other analysis methods [March, 1989:pp1-2] [AFIT, 1990] [EVB, 1989]. Object-oriented approaches were pioneered by Booch and Abbott as a method of exploiting the features and constructs provided by the newly developed Ada programming language [March, 1989:pp1-1]. This method of using an informal strategy as the basis for object-oriented analysis is more recently rejected by many. This is because it inherently lacks rigor due to the impreciseness of the English language [Ladden, 1989:pp86]. In his thesis, March cited Pressman, EVB, and Ladden in saying the following:

> Recent research suggests the use of object-oriented techniques in the earlier phase of requirements analysis provides a more coherent approach to object-oi :nted development. A complete life cycle object-oriented methodology provides a stronger framework for the application of Ada in he management of software complexity. [March, 1989:pp1-2]

OORA can be contrasted to other more traditional methods of requirements analysis such as data-flow oriented analysis and data-structure oriented analysis. Data-flow analysis uses data-flow diagrams, a data dictionary to describe each "flow," and functional descriptions to describe each function (the nodes) in the data-flow diagrams [Pressman, 1987:pp164-175]. DeMarco's Structured Analysis, and Yourdon and Constantine's Structured Methods and Structured Design are examples of methods that are based on data-flow analysis (Do not be mislead by the word "structured" in the names of these methods, these methods are not formal, they are informal methods that use standard notations and embodiments of good practice [Sommerville, 1989:pp179]). Data-structure oriented analysis methods focus on data structure rather than data flow to represent software requirements. Key information *objects* (also called *entities* or *items*) and *operations* are identified and a hierarchy is formed to represent the requirements. Warnier-Orr and Jackson System Development are examples of methods that are based on data structure oriented analysis. [Pressman, 1987:pp293-333]

OORA, on the other hand, describes a system as a series of interacting objects (or classes, since an object is an instance of a class) [EVB, 1989]. The interactions can be seen as messages or operations. Objects generally are named using nouns; messages are generally named using action verbs [AFIT, 1990]. The objects and messages are derived from a description of the problem.

Objects/Classes can be identified using a top-down decomposition of the problem using the following steps as presented by [EVB, 1989]:

- View the system as an object, produce a precise and concise high-level description of the system.
- Graphically represent the object-oriented composition of the system using semantic networks.
- Define the operations suffered by and required of the system.
- Describe the state information of the system.
- Verify the object-oriented representation of the system.

AFIT described another method of identifying objects using concept maps. Concept maps are similar to entity relationship diagrams but simpler. They consist of *nodes (ovals) and directed arcs.* Node *names represent* important entities or concepts about a topic, and the directed arcs represent relationships among them (Figure 2.7). Concept maps are used as a tool to model the problem space. Concept maps can be leveled by developing lower-level maps to model lower levels of abstraction. Solution-space concept maps are obtained by pruning the problem-space concept maps. Node names can be mapped to objects and arc names to operations. [AFIT, 1990]

Both EVB's and AFIT's methods use object/class specifications to completely describe each object identified. Object/Class specifications include the following:

- A narrative description.
- A graphical representation, both static and dynamic. The static representation should be a semantic network (Figure 2.8). The dynamic representation may be a state diagram, petri-net, or both (Figure 2.9).
- Operations, both suffered operations and required operations.
- Possible states.

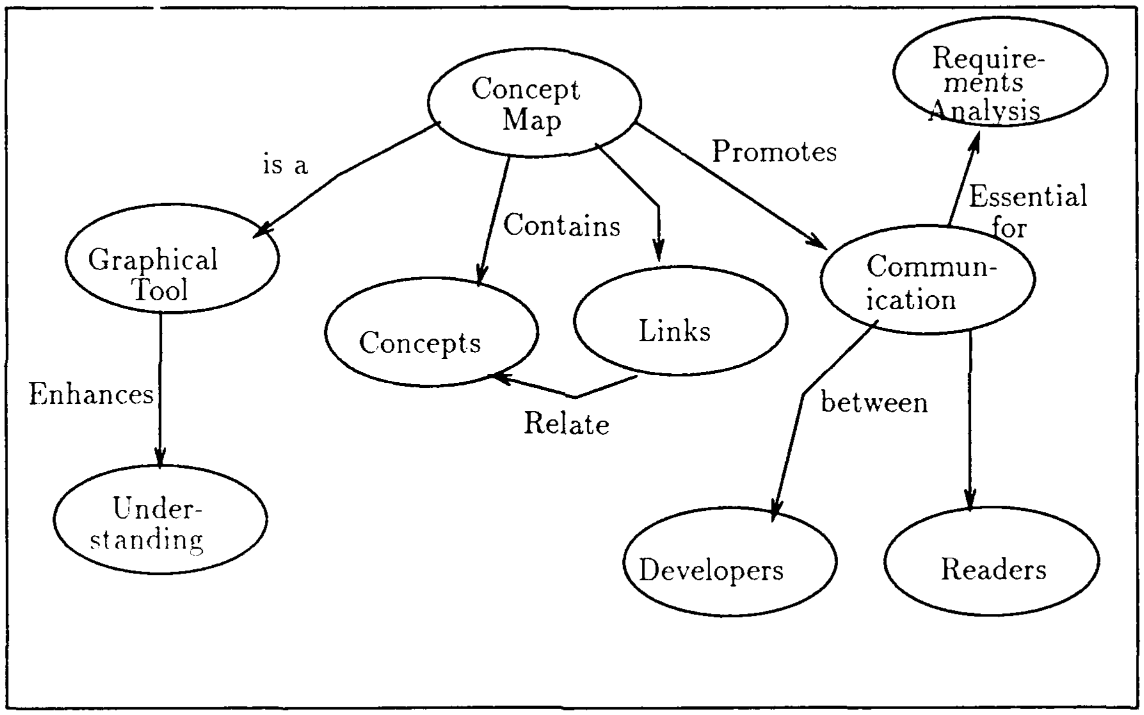

Figure 2.7. Concept Map of "Concept Maps"
[March, 1989:pp2-16]

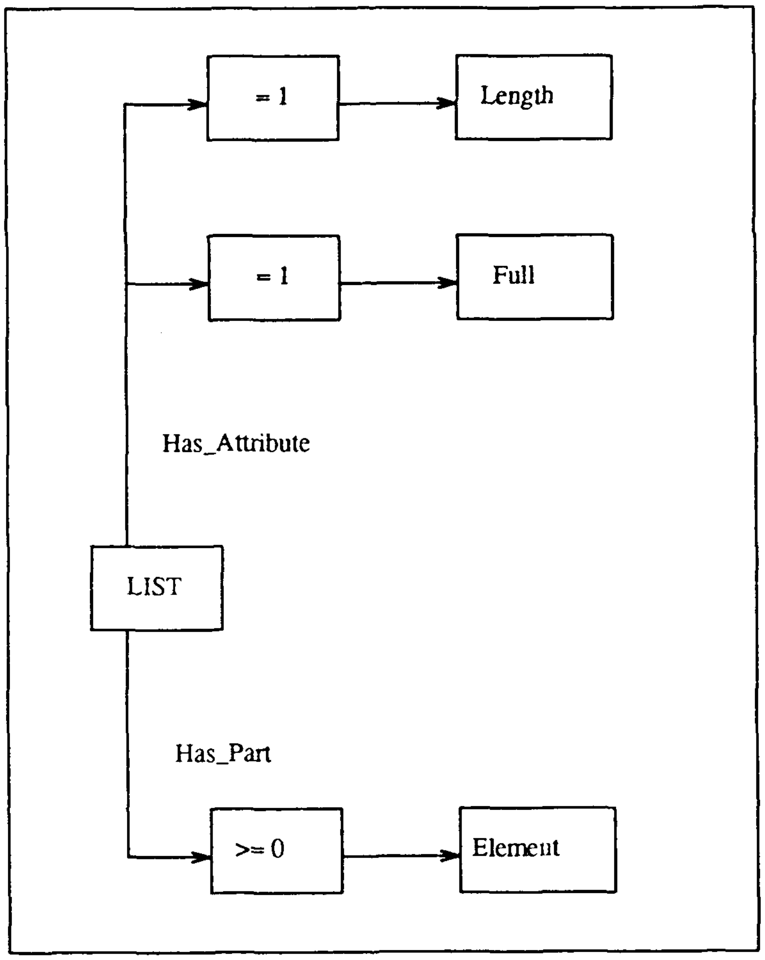

Figure 2.8. Static Relationship Diagram for "List" Object
[EVB, 1989]

- Possible exceptions.

AFIT also used event-response lists and story boards to supplement the concept maps and an overall object/class network diagram to summarize and show visibility among all the objects [AFIT, 1990]. Section 4.3.1.1 contains an example of an event-response list. See March's thesis for an example of story boards.

March developed his version of an object-oriented analysis method in his thesis. His method is intended mainly for embedded systems to be implemented using Ada. March's method is similar to AFIT's in that he used concept maps, story boards, and

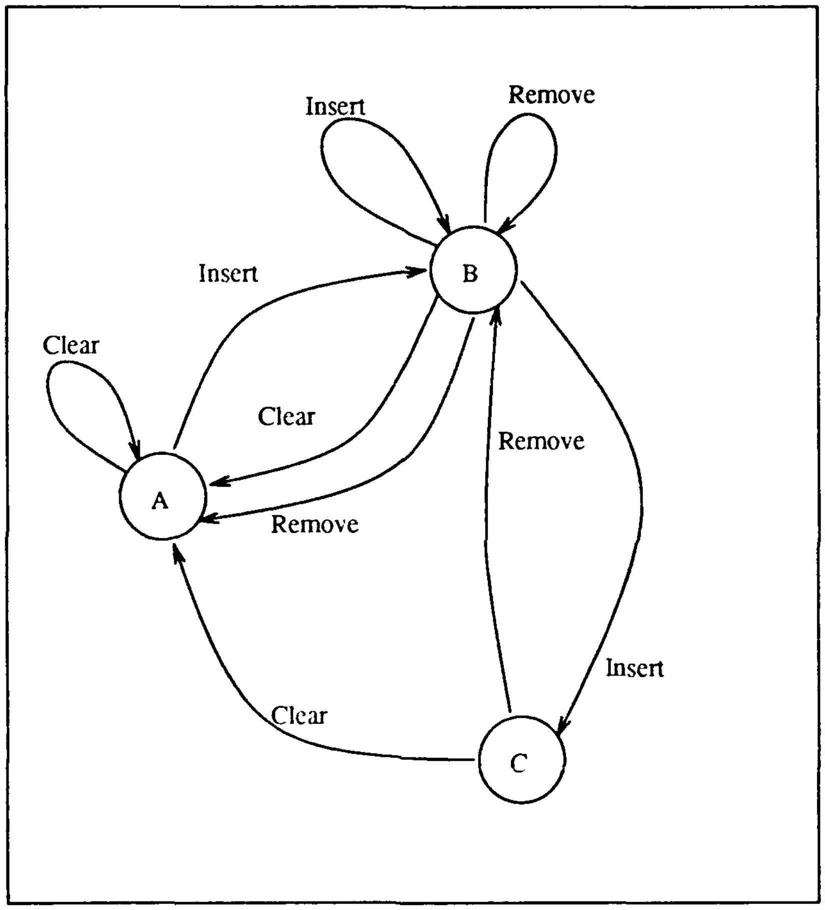

Figure 2.9. Dynamic Relationship - State Diagram for "List" Object [EVB, 1989]

event-response lists to communicate with the customer and identify objects/classes. He also developed the idea of dividing requirements analysis into two steps: one step for communicating with the customer/domain expert to obtain accurate and complete requirements in a form the customer can verify and a second step for structuring the requirements into a form the designer can more easily use to transform into a design. He also introduced the idea of creating an *object encyclopedia* for documenting the rough equivalent of a set of object/class specifications.

A more detailed presentation of March's method is presented in section 3.2.

2.9 Conclusion

The science of reusing designs is in its infancy at best. This chapter contains some principles that can be applied toward design reuse and some approaches that are currently being tried. A fundamental principle needed for design reuse is design structure consistency. If we can develop a method of representing designs and mapping to designs that produces consistent solutions for problems in the same domain, then we are definitely on the right road to design reuse.

The OOD-Paradigm exhibits many of the principles needed for design reuse. Its principles are presented and used further in the next chapter.

III. A Method of Mapping from an OORA to a Design That Supports Reuse

3.1 Introduction

This chapter describes a method of mapping from the Object-Oriented Analysis method proposed by Steve March in his thesis [March, 1989:pp1-1 ... A-89], to a design following closely the principles of "An OOD-Paradigm for Flight Simulators, 2nd Edition" report written by the SAE Project team at the Software Engineering Institute [Rissman and others, 1988:pp1-120]. The chapter starts by first summarizing the products available from March's analysis method. It then gives a detailed description of the SEI design. This provides the domain and range of the mapping method. Benefits and characteristics of the OOD-Paradigm architecture are summarized and discussed. The mapping method itself follows.

We make the following assumptions about March's analysis method and the OOD-Paradigm:

1. March's method provides a good representation of the requirements.
2. The SEI's OOD-Paradigm architecture represents a good design. A design following the OOD-Paradigm architecture will exhibit the benefits outlined in section 3.4. We are choosing to take a case study approach to this report since it is assumed to be exemplary of a reusable architecture approach to design.

These assumptions will be supported and discussed.

3.2 Overview of the Products of March's Analysis Method

[March, 1989:pp1-1 ... A-89]

March's method involves two major steps, each producing a set of products. These products are discussed in the following sections.

3.2.1 Step-One Products.

Step One is intended to document the requirements of the customer and domain expert. The tools used and products developed reflect this goal. The products of Step One are the following:

- **Define the overall purpose of the software.** This is the starting point for understanding the software to be developed. This description may vary in length from one sentence to one page.
- **Concept maps.** These provide a general understanding of the *elements* of the overall problem and their inter-relationships.
- **Story boards.** This is a sequence of paper drawings depicting a user-view scenario of the system as it runs. It is an early paper prototype of the proposed system. It portrays a sequence of actions and can be used to portray the physical layout of screen displays, though it is not limited to this.
- **Event-response lists.** These complement the concept maps by listing the sequence of actions (responses) to be taken in the event of a particular stimulus. Both external and internal stimuli are listed in the event-response list. Maximum response time is part of the event-response list.
- **Known software restrictions.** These can be non-functional requirements (size and timing constraints), regulatory restrictions, security, etc.
- **Metarequirements.** These are design decisions made by the customer apriori. An example is the use of an internal data base format to ensure compatibility with other existing or planned software.

3.2.2 Step-Two Products.

Step-one products are transformed into step-two products, in a value added manner, resulting in products intended to be more suitable for use by the designer. March says that step two adds structure to the products of step one. The products of step two are the following:

- **External interface diagram.** Puts the software system in context with its external environment.
- **High-level actor object identification.** This is the analog of the main driver in a program. It controls/coordinates the action of part, or all, of the rest of the the software.
- **Organized preliminary object list.** Objects are listed, grouped by class, and formed into hierarchies when appropriate.
- **Message senders and receivers.** This is a transformation of the event-response list created in Step One. The main change is that the events and responses are viewed as messages and an attempt is made to identify the senders/receivers of these messages. Message may be forwarded from object to object.
- **Object encyclopedia.** - Each object/class is documented with an entry in the *object encyclopedia.* Entries contain the following information:
  - **Textual description.** This states the purpose of the class and miscellaneous information not included anywhere else.
  - **Structure diagram.** This shows attributes/subobjects. It looks like a concept map but is limited to relations that are structural relations of the class being discussed.
  - **Interface diagram.** This is the communication of this object to other objects in the system. This one also looks like a concept map, but it shows the external view of the object/class. There should be a correspondence here between the interfaces to other classes and the messages received/sent list.
  - **State transition diagram.** This may help in identifying messages that an object receives. It may also indicate that a certain message must be

received to transition the object into a different state. It is included if appropriate.

- **Message received/sent.** Two lists naming the message and who it's received/sent from/to.
- **Description of state limitations.** Some messages may be received/sent only when the object is in a particular state.
- **List of exported exceptions.** Included if appropriate.
- **List of exported constants.** Included if appropriate.
- **Reuse considerations.** Explains if the object/class is application specific or if it may be generalized for use elsewhere.

3.3 Description of the OOD-Paradigm Architecture

[Rissman and others, 1988:pp1-120]

This description continues from the outline of the OOD-Paradigm presented in Section 2.6.2.

The design team started the project with two basic goals: eliminate nested implementations of objects (rationale discussed in Section 3.4.4) and simplify dependencies among objects. Notable characteristics of the architecture that will be discussed are the following:

1. Logical layers replacing the usually nested objects found in a composition hierarchy.
2. Connectors for moving data between objects.
3. Templates for instantiating the architecture.
4. Object managers as the lowest level templates.

3.3.1 Structure Overview.

The architecture consists of three logical levels (layers): the executive level, the system level, and the object level. An executive controls the update of a set of systems, a system controls the update of a set of objects. In the context of the earlier discussion on the composition hierarchy, a set of systems can be seen as the result of a decomposition of an executive; and a set of objects is the result of a decomposition of a system (Figure 3.1).

Starting at the bottom, the fundamental units of the architecture are objects and connections, which constitute the object level. Objects use mathematical models to represent real-world entities. An object's only interface to its environment is through its connection object(s) (except possibly the global types package, called Standard_Engineering_Types). In this way, objects operate in general ignorance of the rest of the system. They map their inputs to their outputs and maintain their state. The mathematical models, themselves, are provided by the manufacturer of the aircraft components for the actual aircraft.

A connection is a mechanism for transferring state information between objects. Invoking a connection results in reading the state of some objects on the connection and broadcasting it to others. The two levels of connections are: executive level and system level. Executive-level connections transfer state information between objects in different systems, and system-level objects transfer state information between objects in the same system. The object on one side of a connection may be a hardware object. Connections perform data-type conversions when necessary (Figure 3.2).

Connecting procedures provide a consistent means of updating systems and objects. Thus, connecting procedures provide a means for implicitly specifying control flow. No extraneous concepts or operations are required. They provide a locus of control since all connections at an abstraction level are handled in one place.

A system, the middle level in the hierarchy, provides two abstractions. First, it logically groups a set of objects and their connections. Second, it provides an update abstraction to update the objects as a unit in order to maintain system

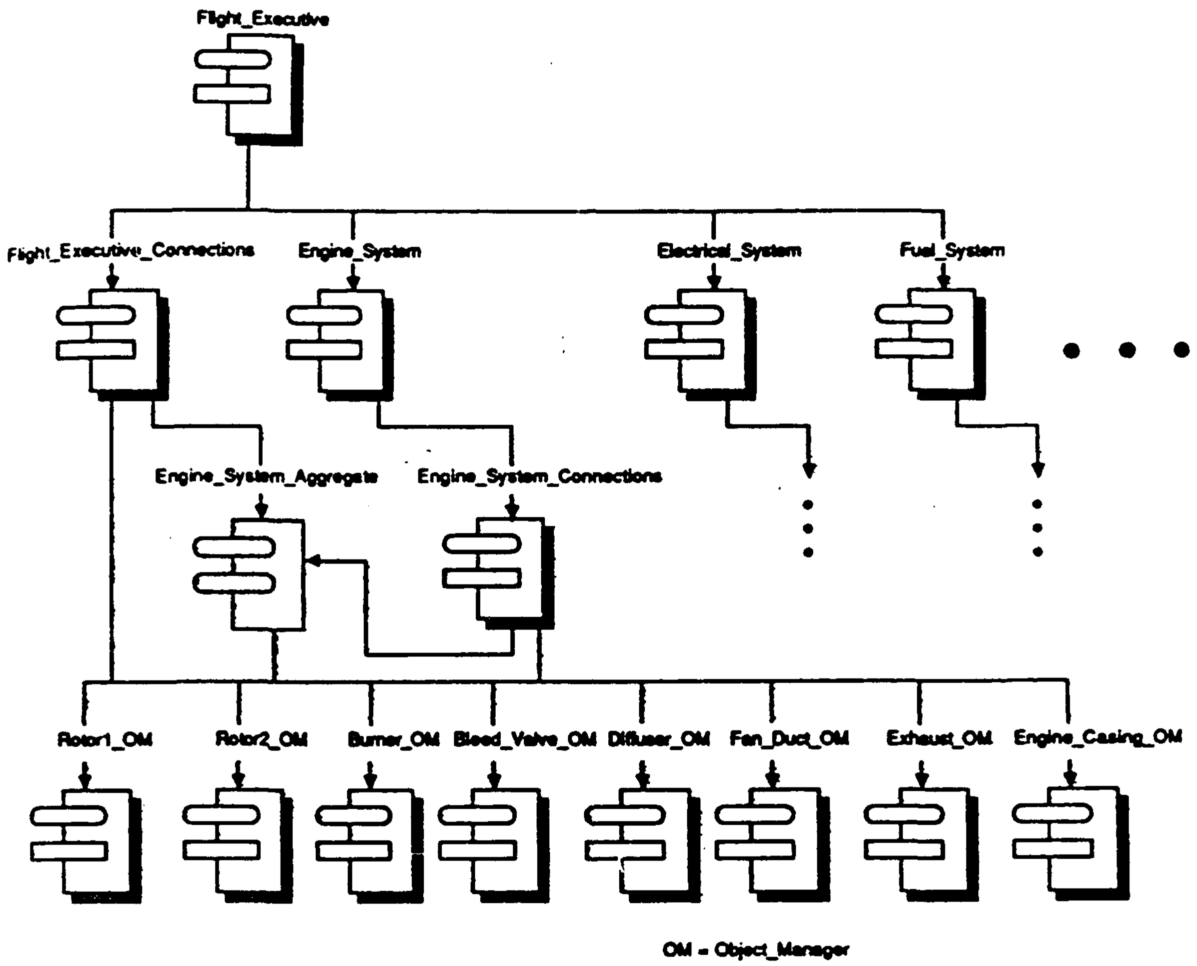

Figure 3.1. SEI Overall Software Architecture
[Rissman and others, 1988:pp15]

```
with Standard_Engineering_Types;
with Engine_System_Aggregate;
with Ignition_System_Aggregate;

with Flight_System_Names;

with Burner_Object_Manager;
with Ignition_Object_Manager;

separate (Flight_Executive_Connection_Manager)

procedure Process_External_Connections_To_Engine_System is

  Integrated_Drive_Energy : Generator_Object_Manager.Energy;

  Some_Spark : Ignition_Object_Manager.Spark;
  The_Burner_Spark : Burner_Object_Manager.Spark;

  function Spark_Conversion (In_Spark : in Ignition_Object_Manager.Spark)
                return Burner_Object_Manager.Spark is
  begin
    case In_Spark is
      when 0 .. 2 =>
        RETURN Burner_Object_Manager.None;
      when 3 .. 9 =>
        RETURN Burner_Object_Manager.Low;
      when 10 .. 20 =>
        RETURN Burner_Object_Manager.High;
    end case ;
  end Spark_Conversion;

begin  -- Process_External_Connections_To_Engine_System

  for An_Engine in Flight_Systems_Names.Aircraft_Engines loop

        Some_Spark := Ignition_Object_Manager.Get_Spark_From
             (A_Ignition => Ignition_System_Aggregate.Ignitions
                                    (Engines_To_Ignition_Map (An_Engine)));

        The_Burner_Spark := Spark_Conversion (Some_Spark);

        Burner_Object_Manager.Give_Spark_To
             (A_Burner   => Engine_System_Aggregate.Engines
                                    (An_Engine).The_Burner,
              Given_Spark => The_Burner_Spark);
   end loop ;
end Process_External_Connections_To_Engine_System;
```

Figure 3.2. Executive-Level Connection–Spark Conversion Routine [Rissman and others, 1988:pp29]

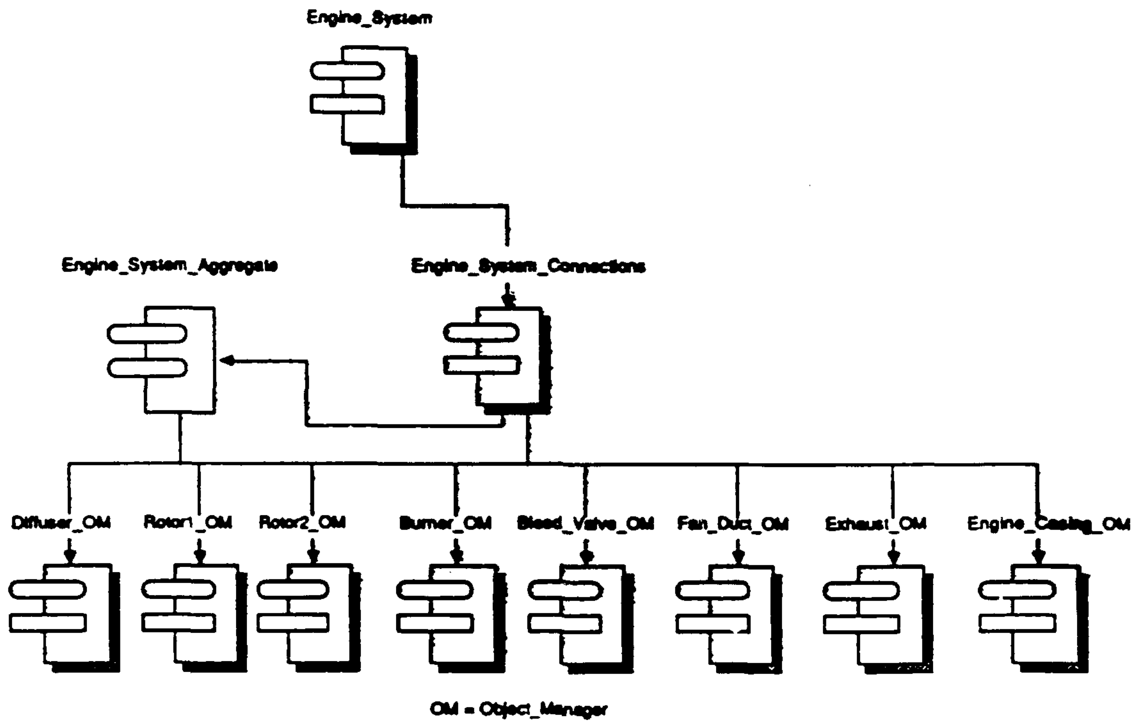

Figure 3.3. SEI System-Level Architecture
[Rissman and others, 1988:pp14]

state consistency. The system performs the update by gating, i.e. invoking, all of its system-level connections to transfer the states of the connected objects. (Figure 3.3).

At the upper-level, an executive performs a similar abstraction as a system. At update time it gates all the executive-level connections and then calls its systems to do their updates as described above (Figure 3.4). Distributed processing could be achieved by distributing each executive to run on its own processor.

3.3.2 The System Abstraction. A system is an aggregation of objects, and the connections between the objects, with a common goal. For example, the

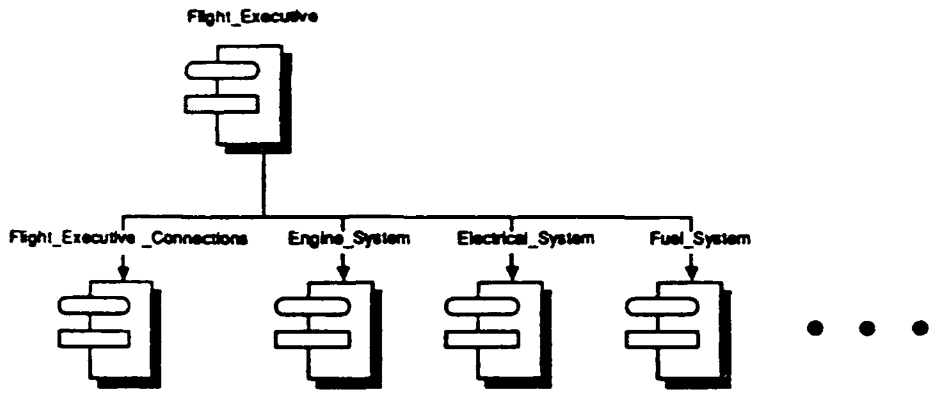

Figure 3.4. SEI Executive Level Architecture
[Rissman and others, 1988:pp13]

objects making up the engine system provide thrust; the objects of an electrical system provide power. A system is updated as a unit.

Each system includes an "aggregate" package. This package contains the actual instances of the objects for that system (Figure 3.5). This is possible because the objects are abstract data types (private types) declared in each object manager. This allows multiple instances of each object to exist. Operations on the objects (implemented by the mathematical models) are of course part of the object managers themselves. There is one object manager for each kind of object.

The developers employ an "object diagram" as a tool to map the real-world objects to the architecture. Figure 3.6 represents the set of real-world objects being mapped and Figure 3.7 is the object diagram mapping. Instantiation information for the components of the architecture come from the object diagram once it is developed. The object diagram has icons that represent objects (rectangles), connections (arrows), systems (roundtangles), and executives (shaded areas).

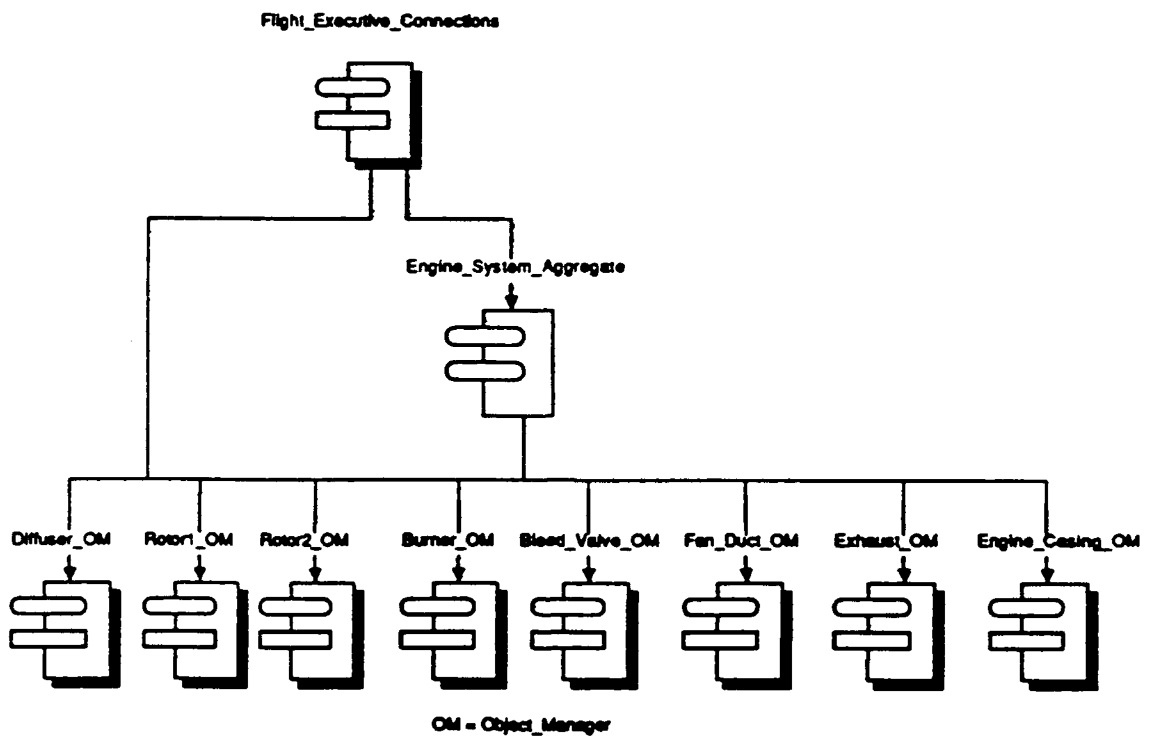

Figure 3.5. Connection Manager Software Architecture [Rissman and others, 1988:pp13]

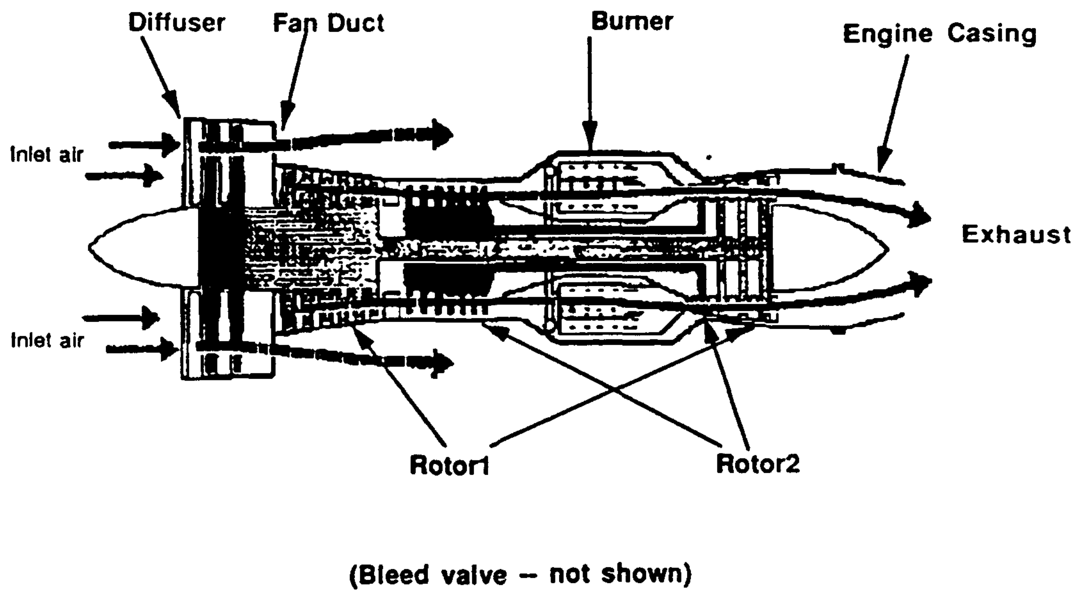

Figure 3.6. Turbofan Engine Description
[Rissman and others, 1988:pp18]

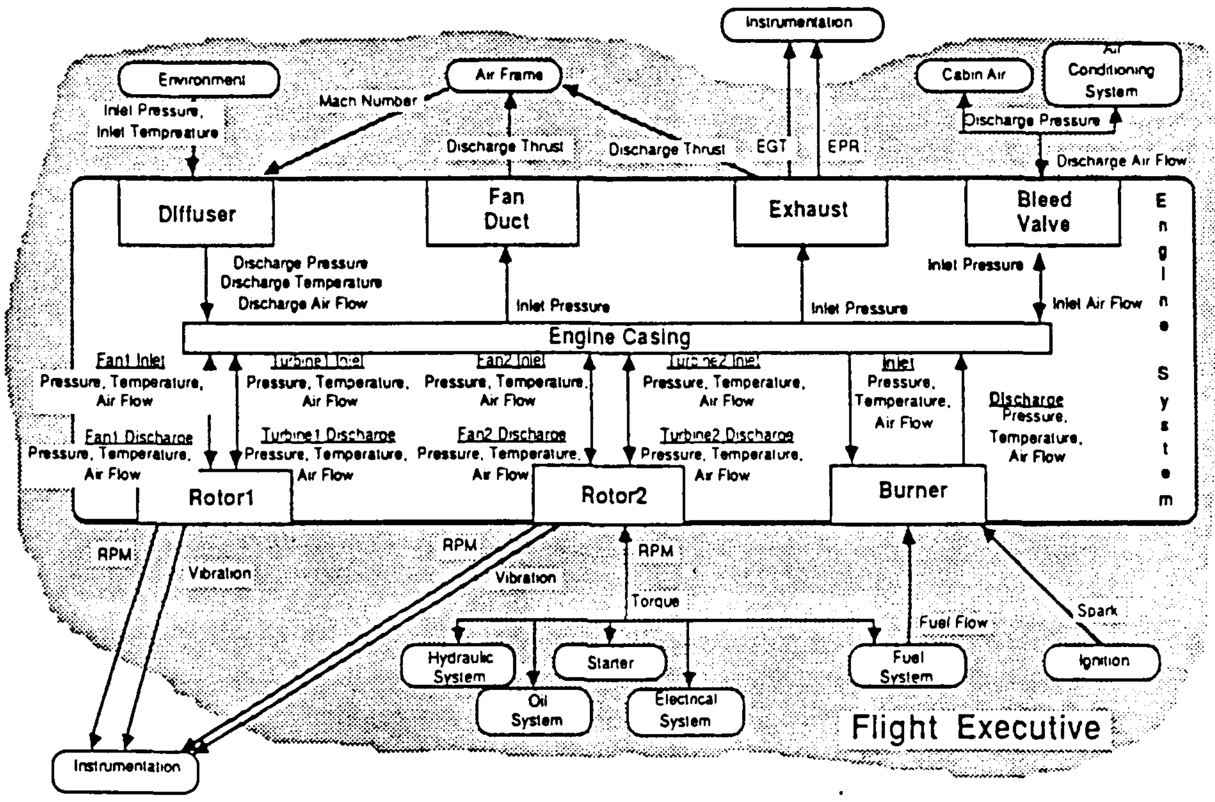

Figure 3.7. Turbofan Engine Object Diagram [Rissman and others, 1988:pp19]

3.3.3 Templates For Recreating Architecture Parts.

Templates are a kind of generic used for creating instances of most of the components of the architecture. Standard naming conventions describe the package and subprogram specifications, the implementor supplies the implementation details. An example of the notation is "$< Object >$ *_Object_Manager*," for the package name of each object manager, where "$< Object >$" is replaced by the actual object name. Also each object manager has exactly three types of suffered operations, which also have standard names: "*New_* $< Object >$" to create new instances of the object, "*Give_* $< external\_effect >$ *_To*" for writing external effects to an object, and "*Get_* $< object\_output >$ *_From*" for reading the state from an object. Figure 3.8 shows a template ready to be used, in this case the object manager template, and Figure 3.9 shows the object manager template after it has been instantiated for the burner object.

Two steps are required to instantiate the architecture. First, an object diagram must be created, which is then mapped to the components of the architecture. To create an object diagram, the implementor first reviews the domain/requirements and selects a set of real-world objects and connections. These are grouped into systems following real-world analogies. This information is mapped into an object diagram. Once the object diagram is created, the architecture dictates the implementation.

An object diagram is created for each system. The software architecture can then be derived mechanically (instantiating templates) from the set of object diagrams. The implementor fills in the details of the templates and provides the appropriate mathematical models for the bodies of the object managers. There is potential here for an automated tool to parse the object diagrams and generate filled-in templates.

```
with Standard_Engineering_Types;
package <Object>_Object_Manager is

  package Set renames Standard_Engineering_Types;
  type <Object> is private ;
  type <Attribute_2> is ??;
  type <Attribute_1> is ??;

  function New_<Object> return <Object>;
  --|************************************************************
  --| Description:
  --|   This function returns a pointer to a new <object> object
  --|   representation. This pointer will be used to identify
  --|   the object for state update and state reporting purposes.
  --|
  --| Parameter Description:
  --|   return <object> which is an access to a <object> object.
  --|************************************************************

  procedure Give_<State_1>_To (A_<Object> : in <Object>;
                  Given_<Input>_<Type_1> : in Set.<Type_1>;
                  Given_<Input>_<Type_2> : in Set.<Type_2>;
                  Given_<Input>_<Type_3> : in Set.<Type_3>);
  --|************************************************************
  --| Description:
  --|   Initiates a change in the specified <object> object's
  --|   state given the <input>_<type_1>, <input>_<type_2>,
  --|   and the <input>_<type_3>.
  --|
  --| Parameter Description:
  --|   A_<object> identifies the <object> whose state is to be changed.
  --|   Given_<input>_<type_1> is the <input> <type_1>, in ??units
  --|   Given_<input>_<type_2> is the <input> <type_2>, in ??units
  --|   Given_<input>_<type_3> is the <input> air flow, in ??units
  --|************************************************************

  pragma Inline (Give_<State_1>_To);

private
  type <Object>_Representation;
  -- incomplete type, defined in package body

  type <Object> is access <Object>_Representation;
  -- pointer to an <object> representation

end <Object>_Object_Manager;
```

Figure 3.8. Object Manager Template Example [Rissman and others, 1988:pp42]

```
with Standard_Engineering_Types;

package Burner_Object_Manager is

   package Set renames Standard_Engineering_Types;

  type Burner is private ;
  – an Burner is an abstraction of a Burner within an Engine.

  type Spark is (None, Low, High);
  – burner needs only to know relative spark size

  type Fuel_Flow is (None, Flowing);
  – the burner needs to know only if it has fuel available

  function New_Burner return Burner;

  procedure Give_Inlet_Air_To
        (A_Burner             : in Burner;
         Given_Inlet_Pressure    : in Set.Pressure;
         Given_Inlet_Temperature : in Set.Temperature;
         Given_Inlet_Air_Flow    : in Set.Air_Flow);

  procedure Get_Discharge_Air_From
              (A_Burner : in Burner;
               Returning_Discharge_Pressure    : out Set.Pressure;
               Returning_Discharge_Temperature : out Set.Temperature;
               Returning_Discharge_Air_Flow    : out Set.Air_Flow);

  procedure Give_Fuel_Flow_To
              (A_Burner       : in Burner;
               Given_Fuel_Flow : in Fuel_Flow);

  procedure Give_Spark_To (A_Burner   : in Burner;
                           Given_Spark : in Spark);

pragma Inline (Give_Inlet_Air_To,
               Get_Discharge_Air_From,
               Give_Fuel_Flow_To,
               Give_Spark_To)

private
  type Burner_Representation;
  – incomplete type, defined in package body

  type Burner is access Burner_Representation;
  – pointer to an Burner representation

end Burner_Object_Manager;
```

Figure 3.9. Burner Object Manager Package Specification
[Rissman and others, 1988:pp23]

3.4 Summary of Advantages and Characteristics of the OOD-Paradigm Architecture

3.4.1 Supports Design Reuse. Design reuse can be discussed at two levels: reuse within an application and reuse between applications. Design reuse within an application was the accomplishment of the OOD-Paradigm. They first demonstrated the design for the engine system; then they used the templates to instantiate the design again for the electrical system of the simulator in a follow-on report [Rissman and others, 1989a].

Design reuse between applications is reusing a design solution from one application in a second different application. This idea can be further divided into reuse within the domain and reuse between domains. Reusing the flight-simulator solution for a C141 on a C5 simulator would be reuse within the domain. This thesis presents a method that reuses the design between domains. We utilize the principles, structure, and constructs of the design to achieve a mapping method that develops designs that are similar to each other and that exhibit the benefits listed in this section.

Many advantages are inherent in reusing designs. Some of them are the following.

- **Less Testing Effort.** Once the soundness of the basic design is established, testing can be less stringent for reuses of the design. Also many of the test procedures can be reused.
- **Higher Reliability.** Reliability will have been proven and refined through previous uses of the design.
- **Less Maintenance Effort.** Maintenance personnel can more easily understand the design since it consists of reoccurring patterns.
- **Less Documentation Effort.** Much of the documentation can be reused from one instantiation of the design to the next.

3.4.2 Supports Component Reuse. The architecture supports both kinds of component reuse mentioned in section 2.7 as follows:

- Objects can be exchanged for similar objects within an application because templates and connections provide the plug-in type of interface needed for "swapping-out" objects. Because connections insulate objects from compilation dependencies, they provide a very elegant way to encourage this kind of reuse. This kind of reuse is handy for modifying performance characteristics of the simulator by exchanging an object for one that has somewhat different characteristics.

  Recompiling the package body of the changed object is all the recompiling necessary since the specification of the object manager will not change. This is true even if the new object's type representation is somewhat different since this private-type definition can be placed in the package body.

- Object reuse between applications will be enhanced since objects developed for the flight simulator are not tightly coupled to the simulator. This low coupling means the object is independent, only modeling some real-world abstraction, and is easily reused in a different application.

3.4.3 Easier Development Process. Reoccurring design patterns and low coupling will make the OOD-Paradigm design easy to implement. Some background on how this is important is provided by Booch as he quotes Britton and Parnas:

> The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently ... Each modules's structure should be simple enough that it can be understood fully; it should be possible to change implementation of other modules without knowledge of the implementation of other modules and without affecting the behavior of other modules: [and] the ease

of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed. [Booch, 1991:pp51]

Connections facilitate independent development and reuse by insulating the objects from each other and from compilation dependencies. Objects and systems become stand-alone; each can be developed independently. Connecting procedures provide a "firewall:" changes to objects on one side of a connection do not affect the objects on the other side. Objects do not have to conform to the entire system, only to the specifications and interface of the point where they plug in. Connections also provide a systematic way to handle data-type mismatches.

3.4.4 More Efficient Implementation. The OOD-Paradigm implementation should be efficient owing to the no-nesting policy and the use of connections to move data directly between objects (see also the timing and sizing discussion in Section 5.2.2). Following one of the design goals, nested objects were avoided in the architecture. The layers in the hierarchy are logical layers. Because of this, objects at the lowest levels in the hierarchy can be interfaced without passing data through all the intermediate levels. This can lead to a more natural and efficient simulation of the real world.

Three of the authors, Mr D'Ippolito, Mr Rissman, and Mr Stewart pointed out their two-fold rationale for avoiding object nesting during a personal meeting (paraphrased):

1. The real world is better reflected when nested objects are avoided. For example if a fuel tank object is part of a wing object and needs to be refueled, with nested objects you refuel the wing rather than the fuel tank. Also, the engine system contains a burner which needs a spark from the ignition system. With nested objects the spark would be applied to the engine then passed down to the burner. This is not like the real-world case where the burner gets the spark directly from the ignition via wires (i.e. the connector in their implementation).

2. Design complexity and overhead is increased with nested objects. The highest-level object becomes a choke point for data and invoke calls.

Just because the analysis process produces a decomposition of objects does not mean the implementation of these must be nested. The use of a logical hierarchy as described in the OOD-Paradigm is an example of a method which avoids nested objects. [Rissman and others, 1989c]

The idea of avoiding object nesting runs contrary to *de facto* object-oriented design: object nesting is a result of most object-oriented decompositions. Seidewitz describes just this process as part of his General Object Oriented Development (GOOD) method: higher-level objects are decomposed into lower-level objects and implemented as nested objects [Seidewitz, 1989:pp101].

The OOD-Paradigm authors have carried the logical-hierarchy process a bit farther than creating logical layers to implement analysis hierarchies. With the *de facto* method of object decomposition, each decomposition may have a different number of layers. Their method, however, utilizes standard levels with standard names. Each decomposition looks remarkably like the next.

3.5 A Mapping Method from an OORA Method to a Design Following OOD-Paradigm Principles

3.5.1 Background and Goals of the Mapping Method. The OOD-Paradigm was developed to solve problems in the flight simulator domain, which consists of time-driven problems. Although the mapping method presented produces designs based on the principles in the OOD-Paradigm, the problems used for March's analysis are event driven, and therefore fundamentally different. The mapping method produces a design with a very similar structure, but with a somewhat different control sequence. All the advantages given in section 3.4 are evident in the designs resulting from the mapping method (see analysis in Section 4.4).

An additional requirement for designs resulting from the mapping method are that they satisfy the quality criteria presented by [Pressman, 1987:pp216]. To summarize his criteria, we can say that a design should exhibit a hierarchical organization, consist of independent modules, be derived using a repeatable method driven by information obtained during software requirements analysis, and contain a distinct and separable representation of data and procedure.

The design resulting from the mapping method satisfies the above quality criteria. It is a high-level design and is summarized by two diagrammatic tools:

1. The **Hierarchical-Structure Diagram**, which represents the hierarchical or static structure of the design.
2. The **Object-Event Interconnection Diagram**, which represents the dynamics or "procedure" of the system.

These two diagrams represent Pressman's criteria of distinct representations for data and procedure. The mapping method for obtaining the two diagrams represent Pressman's criteria of a repeatable method.

3.5.2 Overview of the Mapping Method. The mapping method involves two transformation steps and introduces four representation tools for conducting the transformations. These tools are the following:

1. The **Object-Mapping Table** which maps analysis *objects* to implementation objects and implementation parameters.
2. The **Hierarchical-Structure Diagram** which organizes the implementation objects defined in the Object-Mapping Table into a hierarchical/static design structure. This diagram represents "levels of abstraction" and "aggregation" as discussed by [Booch, 1991:pp58-59], and is our equivalent to the "Overall Software Architecture" diagram (figure 3-5) of the OOD-Paradigm paper (copied in Figure 3.1 of this document).

3. The **Event-Mapping List** which maps analysis *events* to the dynamic representation of the design.
4. The **Object-Event Interconnection Diagram** which summarizes the information from the Event-Mapping List into a quickly understandable diagrammatic format. This is our equivalent to the "Turbofan Engine Object Diagram" of the OOD-Paradigm (figure 4-2), which is copied as Figure 3.6 of this document.

The first transformation step consists of the four substeps of transforming the analysis requirements to the four representation tools listed above. The second step transforms these four representations into Ada specifications. The second step also includes defining design templates for object managers and overall design reuse parameters. The first step is described in Sections 3.5.3 through 3.5.5, the second step is described in Sections 3.5.6 through 3.5.8.

3.5.3 Map All Objects Using an Object-Mapping Table. The first substep is to map analysis objects. Not all objects from the analysis will map to implementation objects, many will become high-level parameters. Most of these parameters will be used to represent attributes and states of the implementation objects and will be passed as data values in messages between the objects.

Create the Object-Mapping Table as a tool to conduct and document the mapping. List each object from the *Organized Object List* of the analysis into the first column of the table. Form successive columns for **implementation object**, **attribute/state**, and **parameter**. Mark an **X** in the appropriate column for each object in the first column. For those objects mapped to anything besides the second column (implementation object), indicate which implementation object this analysis object is becoming a state/attribute of, or parameter between (see Table 3.1).

ANALYSIS OBJECTS	IMPLEMENTATION OBJECTS	ATTRIBUTE/ STATE	PARAMETER
Object Name 1	X		
Object Name 2	X		
Object Name 3	X		
Object Name 4		X (Object Name 2)	X
Object Name 5		X (Object Name 2)	X
Object Name 6	X		
Object Name 7		X (Object Name 6)	X
Object Name 8		X (Object Name 6)	
Object Name 9		X (Object Name 6)	
Object Name 10	X		
Object Name 11		X (Object Name 6)	X
Object Name 12		X (Object Name 6)	

Table 3.1. Object-Mapping Table

3.5.3.1 Heuristics for Mapping Objects. Implementation objects should usually represent real-world entities or procedural abstractions (those which have a coordinating effect over many of the other objects). The *External Interface Diagram* and some of the high level *Concept Maps* from the analysis can be helpful in identifying implementation objects that represent the real-world entities. The *High-Level Actor Object Identification* section can be helpful in identifying important procedural abstractions. Analysis objects that interface directly to a hardware component should become implementation objects.

Analysis objects that can be used to identify an instance of an implementation object should be an attribute. Analysis objects that can identify a state of an implementation object such as current location, direction of travel, altitude, etc., should be states. Analysis objects that do not represent attributes or states but can be used to carry message information between objects, such as a hardware address, a filename, a steering course value that needs to be made, etc., should be marked as parameters. Most states and attributes will also be parameters.

The bottom line for identifying implementation objects is to be aware that each will either initiate events, respond to events, or both. A look at the *Event/Response List* and the *Message Senders and Receivers* list will help identify these initiators and responders.

3.5.4 Structural Representation: Organize the Objects into the Hierarchical-Structure Diagram.

Now that we have our objects identified, assemble them into a hierarchical diagram that reflects the structure of the problem. This is the Hierarchical-Structure Diagram diagram and it is the first of two high level graphical representations of the design. This representation should reflect the static decomposition of the problem, the data representation.

As our counterpart to the "Overall Software Architecture" (figure 3-5) in the OOD Paradigm paper, the Hierarchical-Structure Diagram should follow the same

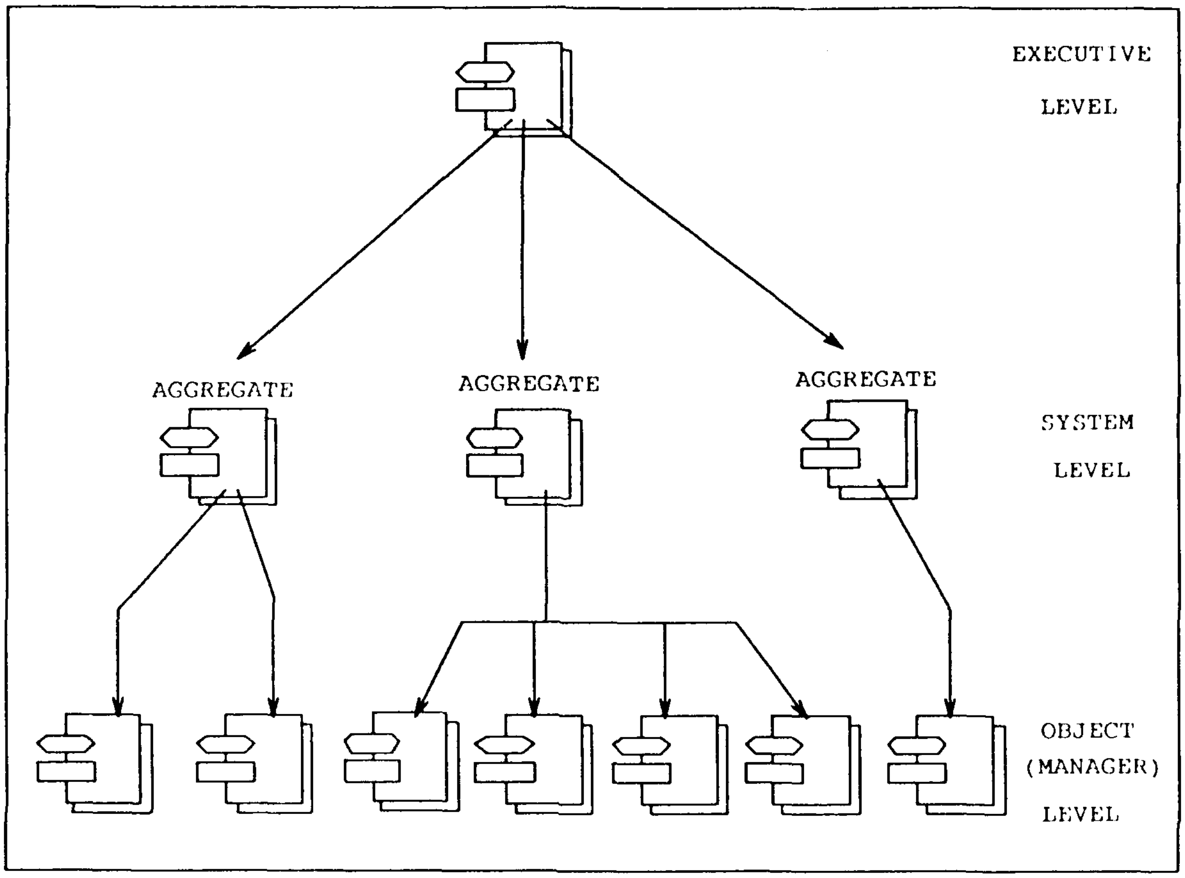

Figure 3.10. Hierarchical-Structure Diagram

three levels: executive, system, and object.

There is only one true level of objects in the diagram: the object level. Systems are aggregates of objects, and executives are aggregates of systems. This design therefore represents a "flat architecture" (Figure 3.10).

3.5.4.1 Heuristics For Mapping Requirements to the Hierarchical-Structure Diagram. The highest level *concept map* from the analysis can *be helpful in identifying logical* objects (aggregates) to become systems for the "system" level. Also the *Overall Organized Preliminary Object List* contains hints for deciding what goes under what in the decomposition. For example, in section A.9 (page A 30), many objects in the list include the note: "associated with each eleva tor." Also the *Structure Diagrams* and *Interface Diagrams* in the *Object Dictionary* can help in putting objects in the right place on the diagram.

The same principle that was used to identify implementation objects can also be used to draw the Hierarchical-Structure Diagram. That is, follow real-world analogies, model the solution after the real-world problem. This is a fundamental object-oriented principle, as discussed in section 2.7.1, and should be used when drawing this diagram.

3.5.5 Procedural Representation: Connect Objects with Events Using the Event-Mapping List and Object-Event Interconnection Diagram.

Following the example of the the OOD-Paradigm (and to some degree Kiem's Keystone method [Kiem, 1989:pp101]) **messages** between objects are passed via **connectors**. No direct interfacing between objects will occur in the implementation.

The mapping method assumes an event-driven problem because this is the type of problem analyzed in March's thesis. Event-driven problems are also common in embedded systems for which Ada was developed. In an event-driven problem, events are initiated within objects, usually reflecting the occurrence of an event in the outside world. The object responds by sending messages to other objects. On the other hand, in the time-driven OOD-Paradigm, events are initiated by a clock, which invokes the *connectors* to transfer data between objects.

The mapping method uses an event-response model to develop the design for event-driven problems. The event-response model is used for both identifying the sequence of messages that need to be passed between objects and documenting the responses that objects and connectors should take to events and messages.

3.5.5.1 Use an Event-Mapping List.

The Event-Mapping List is the tool used to map the *Message Senders and Receivers* list and the *Event/Response List* to objects, connectors, and messages. Each *event* is mapped to a connector and the connector assumes the name of the event. The initiation of an event is mapped to an object. Responses to events are also mapped to objects. At runtime, the initiator invokes the connector when the event occurs, which in turn invokes the responder objects (passing appropriate data of course). This implements the message-passing process. During this mapping process, important parameters, variables, and types are identified.

The procedural design is determined during the Event-Mapping List process and as such the completed Event-Mapping List constitutes one of the most important parts of the top-level design. Implementation objects were identified using the Object-Mapping Table; now the connectors, and interactions between connectors and objects, will be identified.

During the process of event mapping, The *Message Senders and Receivers* list should be used to identify the initiator and responder objects for each event. The process consists of progressing through the *events* in the *Message Senders and Receivers* list (or *Event-Response List*) one-by-one, and describing the message passing needed to accomplish each *responses* listed for each *event.* All responses must be assigned to an object, or in some cases a small procedural response may be assigned to the message-carrying connector.

If the initiator/responder is a system-level object in the analysis, then a determination must be made as to which object in that system the initiator/responder actually is. If the answer is not apparent, review the *Description of Messages Sent and Received* in the *Object Encyclopedia* for each object in this system.

Defining a message that needs to be passed between objects generally adds a message to the connector currently being defined. Often, connectors will send only one message. Responder objects may respond by becoming initiators of new messages, indicating the need for an additional connector, or it may be found that an already identified connector can be called.

Some *responses* should be assigned to the initiator of the *event*; so no message passing will be necessary for that particular response. Record the response in the list, later it will be assigned to the object manager of the object.

A possible enhancement, to introduce time-driven requirements into this event-driven model, would be to create a time-keeping object to initiate events.

The Event-Mapping List should use a format similar to the following; it may be important to enforce the format by putting it on a form:

- **Events 1 & 2:** Copy the number and name of the *event* from the *list.* Put all copied requirements in *italics* so they can be easily identified as requirements being mapped.
  - **Initiator : Object_Name** Name of the object that initiates the event.
  - **Responses:** Copy the *list* of *responses* from the *Event/Response List* and describe how each requirement in each response is satisfied. Break up a response into sublists if the response involves more than one requirement. Each response should be mapped either to a message that needs to be passed or to a statement that the response will be conducted internal to the initiator of the event. Example:
    - **3a** *Read the floor number from the floor sensor input register.* Accomplish internally to Floor_Sensor upon occurrence of event.
    - **3b & 3c** *Extinguish the light on the location panel for the elevator for the previous floor. & Illuminate the light on the location panel for the current floor.*
      - **Connector needed:** From Floor_Sensor to Location_Panel.
      - **Connector name:** Floor_Approach.
      - **Location_Panel Command Needed:** Update_Location_Indicator.

    · **Parameters/Variables:** New_Floor_Number, Elevator_Number.

  – **Maximum response time:** 0.1 seconds.

Since the Event-Mapping List process creates an important part of the high-level design, keep the following rules of thumb in mind:

- Have objects send messages directly to responding objects (via connectors).
- Objects do not interface directly to other objects, objects are not nested in objects, objects only interface to connectors, connectors only interface to objects.
- Keep architecture flat. Remember there is only one true level of objects.
- If an object is expected to perform one of the responses internally, and it was not the initiator of the event, then be sure at least one of the connectors notifies the object that the event occurred.

3.5.5.2 Object-Event Interconnection Diagram.

The next step is to represent the information from the Event-Mapping List in diagrammatic format using the the Object-Event Interconnection Diagram. This diagram is the equivalent of the "Object Diagram" (figure 4-2) of the OOD-Paradigm. It is equivalent in the sense that it diagrammatically shows all objects, connections, and message passing betwee them. Both can be transformed directly to Ada specifications (see also Figure 3.11).

On the diagram, represent objects with Ada package icons and connectors with roundtangles. The lines represent message passing. Line up the objects horizontally across the middle of the diagram to represent the flat aichitecture. Because of this flatness, "systems" and "executives" are not shown. Multiple instances of objects within systems also are not shown since message passing is the same for each instance.

In the implementation, the connectors will have to reference both the system level to access the instance of an object and the object level to access its operations.

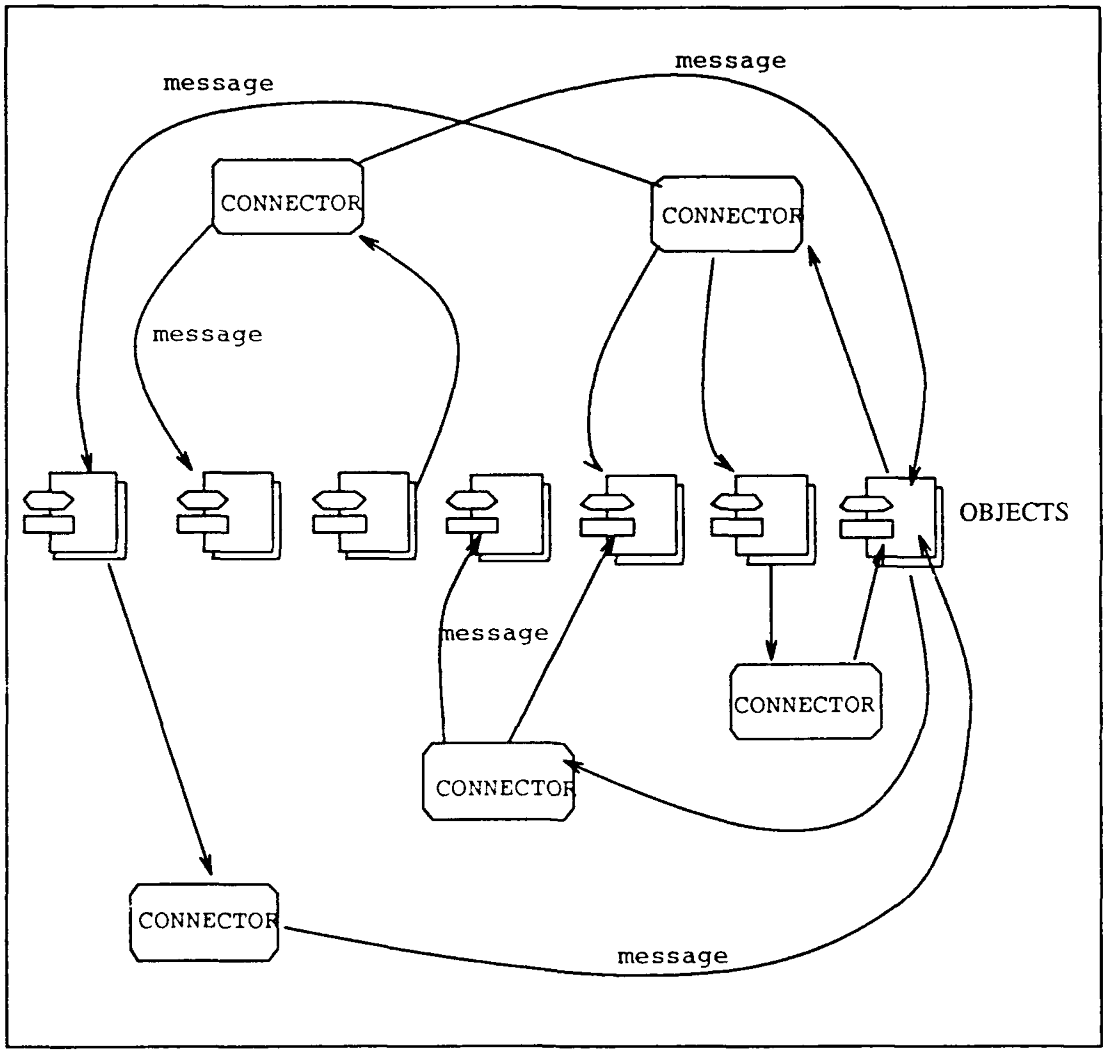

Figure 3.11. Object-Event Interconnection Diagram

3.5.5.3 Sufficient or Complete Set of Object Operations.

The above mapping process results in a "sufficient" set of operations for this application and for reuse at the design level. To implement the objects to be reusable in many applications would require a "complete" set of operations [AFIT, 1990]. March defined a much more extensive set of operations for each object in his analysis which, if implemented with his set, would result in a near complete set of operations [March, 1989:ppA-35 ... A-88]. Berard points out that "completeness" is more than just operations; he says we should also include exportable constants and exceptions [Berard, 1990b].

3.5.6 Mapping to Ada Specifications.

3.5.6.1 Ada Mapping Overview.

Now that we have our four high-level representations of the design, we're ready to map them to Ada specifications. The Ada specifications map directly from the two diagrams. The components map as follows:

- **Objects.** Each object defined at the object level of the Hierarchical-Structure Diagram will map to an Ada "package," which will be named $< Object\_Name >$ $\_Manager$. Each object-manager package will contain a definition of the object as a private type and the required operations on the object. A template will be developed for instantiating object-manager packages (see section 3.5.7).
- **Connectors.** Each connector defined in the Event-Mapping List will map to an Ada procedure named for the connector. Connector procedures may be defined stand alone or may be grouped into a package for convenience.
- **Systems.** Each system kind defined in the Hierarchical-Structure Diagram will map to a package named $< System\_Name > \_System\_Aggregate$. These packages will contain a data structure that contains instances of the objects making up the system. Identification of these objects for grouping into the

system aggregate package also comes from the Hierarchical-Structure Diagram. The data structure contains an additional dimension so that more than one instance of a system can exist in the package; that is, only one system-aggregate package is needed for each kind of system, regardless of the number of these systems that will be used.

- **Executives.** In a similar way to systems, each executive kind from the Hierarchical-Structure Diagram will map to a package called *< Executive_Name > _Executive_Aggregate.*

- **Parameters.** Most of the parameters, variab[illegible], and types defined during object and event mapping, which are used in conjunction with more than one object, will be defined as types in the *Standard_Engineering_Types* package. This package also contains the information needed to instantiate the design (see Sections 3.5.8, A.1, and B.1).

3.5.6.2 Ada Mapping Products. The results of the Ada mapping process will be the following:

- **Complete Object_Manager package specifications.** The object itself should be defined as a private type. This private type should be an access type which points to an instance of the representation of the object. The representation generally will be either a record or a task type. Task types are used if the object is one that will accept hardware interrupts. In either case, objects will contain unique state and configuration information for each instance.

- **Complete System_Aggregate Package Specifications and Bodies.** The body of this package exists only to initialize the data structure containing instances of the system objects.

Executive aggregate packages will exist if more than one executive exits. Multiple executives generally will be defined in larger systems or systems that will be multi-processed.

- **Complete Connector Procedures.** Not just procedure specifications.
- **Complete Standard_Engineering_Types Package.** Like the aggregate packages, the types package may include a package body for initializing data into a data structure.
- **Elaboration Order Where Appropriate.** For example, the system-aggregate packages will be calling the object managers from the package body-initialization section to initialize their data structures, so the object managers will need to be elaborated before the system-aggregate packages.

3.5.6.3 Information Mapping. The Hierarchical-Structure Diagram provides the information needed to build the static structure of the software. This is done by defining the objects as object managers, the systems as system aggregate packages, and executives as executive aggregate packages, and the corresponding groupings for each.

The Object-Event Interconnection Diagram supplemented with the Event-Mapping List, provides the information needed to identify the operations each object manager should export, the identity of the connector procedures, the connectors each object manager will need visibility to from the package body (using "with" clauses), the object managers each connector will need visibility to, and the identity of necessary messages with corresponding parameters.

The Object-Mapping Table identifies the parameters that will need to be defined as types in the Standard_Engineering_Types package.

All the information from the analysis should be on hand as supplement information during the Ada mapping process. For example, the *Metarequirements* section of the analysis will be needed to define the configuration parts of the *Standard_Engineering_Types* package (see Sections 3.5.8, A.1, and B.1).

3.5.6.4 Ada Mapping Process. The process of defining the Ada specifications is to walk through the Event-Mapping List using the Object-Event Interconnection Diagram as a guide. The diagram identifies object managers and their operations, connector procedures, and the "with"ing between them. Use the List to be sure the definitions of the objects and connectors are complete in the sense that all the requirements represented in the List are mapped to Ada specifications. Check-off items in the List as they are mapped.

Create new instances of the object managers at the first reference to them in the List (see object manager template discussion Section 3.5.7 and example Sections A.2, A.3 and B.2). The "with" list at the top of each object manager can be derived from the arrows on the diagram; arrows point to "with"ed components. Fill these in, though later some should be moved to the package body if visibility to the "with"ed component is not needed at the specification level. Object managers usually should "with" only the *Standard_Engineering_Types* package from the specification part, connectors should be "with"ed from the package body.

3.5.6.5 Document Mapped Requirements in Ada Specifications. The requirements mapped to each object should be documented in the package specification in a consistent and structured format. This format should be formally defined in the object templates as discussed in the next section.

3.5.7 Develop Templates for Instantiating Object Managers. Ultimately, defining the object packages will become a process of filling in the blanks of an object template. The rationale for using templates is discussed in Sections 3.3 and and 3.4. Examples from the elevator problem of an object-manager template and instantiated object-manager templates are in appendix Sections A.2, A.3, and B.2.

The basic principle is to generalize existing Object Managers to realize the template and then use the template for instantiating further object managers [Plinta and Lee, 1989:pp66]. Use the guidelines in the following sections for defining templates:

3.5.7.1 Look for the Reoccurring Pattern of Object Managers. This concept is best explained through contrasting examples: In the OOD-Paradigm for Flight Simulators, they call their model the "Object Connection and Update Model" because they're modeling a system of objects that are updated at specific time intervals (time driven) [Rissman and others, 1988:pp4]. On each time interval, the connectors are invoked and they read the state from one object and pass it to another. With this pattern in mind, they defined the reoccurring operations of the object managers to be ones that *get* state from an object and *give* state to an object.

Event-driven problems lead to a stimulus/response kind of object operation. Therefore the reoccurring operations are defined to be **Applying a Stimulus** and **Responding to a Stimulus**. Since the example applications are embedded systems, stimuli may come from either exported operations or hardware interrupts. Responses are either internal state changes, hardware commands, or message sending. All stimuli and corresponding responses should be systematically documented in the object-manager package specifications (see examples in the appendices).

Because abstract data types are used to implement objects, a consistent mechanism for issuing new instances of the object is needed. For this purpose, each object manager will export a function named: *New_ < Object >*. This operation will initialize the object with state and configuration data obtained from the Standard_Engineering_Types package as appropriate. This operation will be called by the aggregate package during elaboration, in order to initialize each "system."

3.5.7.2 Develop A Standard Format For the Object Managers. The standard format for the template should contain exported operation templates, abstract data type templates, documentation templates, etc. The places where names are needed to instantiate the templates should be enclosed in brackets < ... >.

The documentation template is important for the purpose of completing the requirements trace. It should be structured to promote a clear understanding of the nature of the object's operation (stimulus/response in the case of our examples). Requirements from the Event-Mapping List (and other earlier listed sources, as needed, such as the *Object Dictionary*) should be copied or paraphrased systematically to the documentation template, along with how each requirement is satisfied by the package. The implementor of the object manager should be able to properly implement the package body from the documentation in the package specification and the information in the Standard_Engineering_Types package.

3.5.7.3 Identify The "Generic" Parts. These generic parts are templates needing to be filled in with instantiation parameters. The most fundamental of these parameters are the $< Object\_Name >$ and the $< System\_Name >$. The generic parts include exported-operation templates, "with"ing templates, object-representation templates in the private part, etc (see Section A.2 for an example).

To avoid confusion, compare the two kinds of "generics," for which instances are created, involved in this discussion of object managers (but neither includes the using of Ada generics): The object-manager templates are used to instantiate new object-manager packages during the current design process. The objects defined in the object managers are abstract data types from which multiple instances are created during run time; these are kept in a System Aggregate package.

3.5.7.4 Template Instantiation. The object templates are "instantiated" by replacing all generic information with the particular information for the object. Names are instantiated by using the global "find & replace" function of an editor for each generic parameter. For example, the first find and replace will be to replace the string "$< Object >$" for the actual object name.

Instantiating the documentation part of the template is very important for tracing requirements, as mentioned, because this is where mapping of requirements to Ada code is documented (see the object-manager examples in the appendices).

3.5.8 Develop the Standard_Engineering_Types package for Instantiating the Design. This package serves two purposes:

1. It contains type definitions for the parameters defined during object and event mapping. These are defined in the areas marked "PARAMETER AREA #X:."
2. It configures the design for reuse. The areas marked: "CONFIGURATION AREA #X:" contain data used to instantiate the design. See the example in Section A.1.

Adjusting the configuration values in the *Standard_Engineering_Types* package is the most direct way of reusing the design (See Section 3.4 for a discussion of other levels of reuse). The elevator problem example package contains 21 configurable values, including the number of elevators controlled, number of floors in the building, the weight capacity of each elevator, hardware address, hardware commands, and hardware interrupt vectors. This package can be thought of as a template for instantiating the design implementation.

The source of information needed to decide what parameters should be made generics in the configuration area of the *Standard_Engineering_Types* package should come from a domain analysis. Information needed to instantiate these parameters can be found in the *Metarequirements* and *Object Dictionary* sections of the analysis.

3.5.9 Cross Check Transformations as a Tracing Step. Both transformation steps should be cross checked. First reread the analysis and verify that all the requirements are mapped into at least one of the four representation tools. Then walk through the four tools and verify that each item is mapped to the Ada specifications. The most important parts of the analysis to cross check are the *Event/Response List* and the *Metarequirements* section.

IV. Validation of the Mapping Method

4.1 Introduction

The mapping method presented in Chapter III can be validated by applying it to two sample problems and comparing the results. One goal is that the resulting designs for the two problems would be very similar. In the problem statement of this thesis (Chapter I), we stated that one of the main problems with current design methods was their tendency to create unique solutions, which greatly reduced the potential for design reuse. A method that develops consistent designs is a large step toward design reuse.

Other goals for the resulting designs are that they would exhibit the benefits of Section 3.4 and would be able to be instantiated within at least some limited domain.

This chapter applies the mapping method to the two problems analyzed by March in his thesis [March, 1989:pp3-18 ... 3-39, A-1 ... A-89]; these are the Elevator Controller and Cruise Control problems. This chapter contains the first transformation step for both problems, that is, transformation to the four representation tools listed in Section 3.5.2. The second transformation step, to Ada specifications, is included as the first two appendices. As an additional validation step, the elevator design is implemented and the results are discussed at the end of the chapter.

4.2 The Elevator Problem.

4.2.1 Important Products from March's Analysis.

The two items from March's analysis most needed by the mapping method are the *Organized Preliminary Object List* and the *Message Senders and Receivers* list. They are copied in the next two sections for convenience:

4.2.1.1 Organized Preliminary Object List.

Elevator Control System

Elevator

- Elevator 1
- Elevator 2
- Elevator 3
- Elevator 4

Direction

(Associated with each elevator.)

Floor Sensor

(Associated with each elevator.)

Elevator ID

(Associated with each elevator.)

Elevator Motor

(Associated with each elevator.)

Weight Sensor

(Associated with each elevator.)

Weight
- Current Weight (Associated with each elevator.)
- Load Capacity (Associated with each elevator.)

Control Panel

- Elevator Control Panel (Associated with each elevator.)
- UP Request Panel
- DOWN Request Panel

Location Panel

- (Associated with each elevator.)

List

- Destination List (Associated with each elevator.)
- Outstanding Request List

Floor

Summons Request

Input Register

- Elevator Control Panel Input Register (1 for each elevator)
- UP Request Panel Input Register
- DOWN Request Panel Input Register
- Floor Sensor Input Register (1 for each elevator)

Output Register

- Elevator Control Panel Output Register (1 for each elevator)
- UP Request Panel Output Register
- DOWN Request Panel Output Register
- Location Panel Output Register (1 for each elevator)

Address

- (Associated with each input or output register.)

Interrupt Number

- (Associated with each control panel and floor sensor.)

4.2.1.2 Message Senders and Receivers.

Event1: A passenger issues an "up" summons from a particular floor (interrupt).
Sender: UP Request Panel Receiver: Elevator Control System

Resp.1a: Read the Up Summons input register to determine the floor number of the request. (Performed by UP Request Panel)

R1b: Illuminate the light behind the button on the UP summons request panel. (Performed by UP Request Panel)

R1c: If there is an idle (parked) elevator, send it to the floor where the summons was issued. (Performed by Elevator Control System)

R1d: Add the request to the list of outstanding requests. (Performed by Elevator Control System)

E2: A passenger issues a "down" summons from a particular floor (interrupt).
Sender: DOWN Request Panel
Receiver: Elevator Control System

R2a: Read the DOWN Summons input register to determine the floor number of the request. (Performed by DOWN Request Panel)

R2b: Illuminate the light behind the button on the DOWN summons request panel. (Performed by DOWN Request Panel)

R2c: If there is an idle (parked) elevator, send it to the floor where the summons was issued. (Performed by Elevator Control System)

R2d: Add the request to the list of outstanding requests. (Performed by Elevator Control System)

E3: A sensor for an elevator signals its arrival at a particular floor (interrupt).
Sender: Elevator Floor Sensor
Receiver: Elevator
Forwarded To: Elevator Control System

R3a: Read the floor number from the floor sensor input register for that elevator. (Performed by Floor Sensor)

R3b: Extinguish the light on the location panel for the elevator for the previous floor. (Performed by Elevator)

R3c: Illuminate the light on the location panel for the current floor. (Performed by Elevator)

R3d: If the floor is listed in the destination list for the elevator, then stop the elevator at the floor and extinguish the light behind the floor number on the elevator's control panel. After stopping, remove the floor from the destination list, wait 3 seconds, then proceed to the next destination. (Performed by Elevator)

R3e: If the floor and direction are listed in the outstanding request list, then stop the elevator at the floor. Extinguish the light behind the floor button on the proper request panel, and remove the summons request from the outstanding request list. After stopping, wait 3 seconds, then proceed to the next destination. (Performed by Elevator Control System)

E4: A passenger presses a destination button on the control panel of a particular elevator (interrupt).
Sender: Elevator Control Panel
Receiver: Elevator

R4a: Read the control panel input register to determine the desired floor number. (Performed by Control Panel)

R4b: Illuminate the light behind the button on the control panel for the elevator. (Performed by Control Panel)

R4c: Add the floor to the destination list for the elevator. (Performed by Elevator)

E5: An elevator becomes overloaded.
Inquiry Sender: Elevator
Inquiry Receiver: Weight Sensor

R5a: Disable the elevator so that it does not move until the overload condition is gone. (Performed by Elevator)

R5b: Periodically (approximately every 5.0 seconds) check to see if the overload is eliminated. (Performed by Elevator)

E6: Time to check elevator weight sensor (periodic).

R6: If current weight is less than max load, then respond to commands. Otherwise, delay another 5 seconds and check the weight sensor again. (Performed by Elevator)

4.2.2 Elevator Object-Mapping Table. This section maps the analysis objects from the elevator problem to implementation objects. The list of analysis objects is included as Section 4.2.1.1.

Following the heuristics of Section 3.5.3.1, of the 17 analysis objects (20 if each elevator is counted as a separate object), 7 are mapped to implementation objects (Table 4.1). For this embedded system, object identification follows real-world analogies. One procedural abstraction is also identified as an object – the *Elevator Control System*, which we call the Scheduler. Most of the other objects become attributes, states, and parameters. *Input* and *Output Register* become address parameters. The analysis object *Summons Request* was identified as an event so it is not mapped as an implementation object.

4.2.3 Elevator Hierarchical-Structure Diagram. Now that we have our implementation objects identified, we are ready to group them into their natural hierarchy as described in Section 3.5.4.1. The objective is to group the objects into systems and, if needed, executives. The elevator-control system is not complex enough to be decomposed into multiple executives, so no executive will exist (except perhaps as a null procedure for the purpose of linking and initial invocation), but it does decompose into three systems: the Floor_Panels, Elevators, and Scheduler. Although the elevator was identified as an object in the Object-Mapping Table, here we recognize that the elevator is not an object but an aggregate of objects. The Scheduler and Floor_Panels systems each consist of one kind of object (Figure 4.1).

Among the heuristics used to draw the Hierarchical-Structure Diagram are real-world analogies and Figure A.22 from the analysis - *Elevator Control System External Interface Diagram.*

ANALYSIS OBJECTS	IMPLEMENTATION OBJECTS	ATTRIBUTE/ STATE	PARAMETER
Elevator Controller System (Scheduler)	X		
Elevator	X		
Direction		X (Elevator)	X
Floor Sensor	X		
Elevator ID		X (Elevator)	X
Elevator Motor	X		
Weight Sensor	X		
Weight		X (Elevator)	X
Control Panel	X		
Location Panel	X		
List			Internal to Scheduler
Floor		X (Elevator, others)	X
Summons Request	Event		
Address			X
Input Register	(Address)		X
OutPut Register	(Address)		
Interrupt Number			X

Table 4.1. Elevator Object-Mapping Table

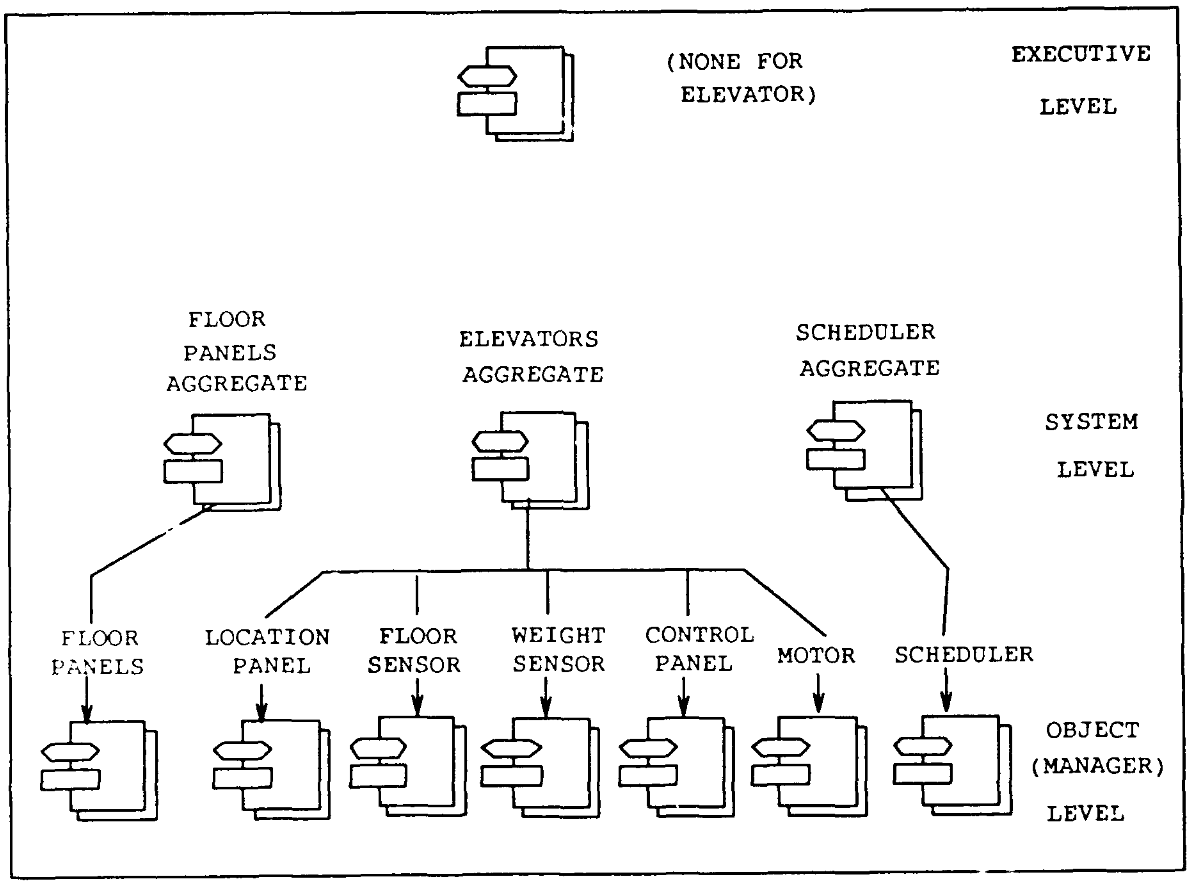

Figure 4.1. Elevator Hierarchical-Structure Diagram

4.2.4 Elevator Event-Mapping List.

We are now ready to develop the Event-Mapping List as described in Section 3.5.5.1. The mapping follows the *Message Senders and Receivers* list, which was included as Section 4.2.1.2. For this problem, March faithfully reproduced the events and responses in the Message Senders and Receivers list; therefore, all that is needed from the event/response list is maximum response times.

Events 1 and 2 were combined since they are the same event, differing only in the value of the parameter: "Direction." The Event-Mapping List follows:

- **Events 1 & 2:** *A passenger issues an "up"/"down" summons from a particular floor (interrupt).*
  - **Initiator:** Floor_Panel.
  - **Responses** as follows:
    - **R1a & R2a:** *Read input register to determine floor number of request.* Accomplish internally to Floor_Panel upon occurrence of event.
    - **R1b & R2b:** *Illuminate the light behind the button on the Up Summons request panel.* Accomplish internally to Floor_Panel upon occurrence of event.
    - **R1c & R2c:** *If there is a an idle (parked) elevator, send it to the floor where the summons was issued.*
      - **Connector needed:** From Scheduler to Motor.
      - **Connector Name:** Proceed.
      - **Motor Command needed:** Go.
      - **Parameters/Variables:** Elevator_Number, Floor, Direction.
    - **R1d & R2d:** *Add the requests to the list of outstanding requests.*
      - **Connector needed:** From Floor_Panel to Scheduler.
      - **Connector Name:** Summons.

      - **Scheduler Command Needed:** New_Summons.
      - **Parameters/Variables:** Floor, Direction.
      - **Connector Processing:** Call New_Summons with the parameters.
  - **Average response time:** 20 seconds for elevator to arrive.
- **Event 3:** *A sensor for an elevator signals its arrival at a particular floor.*
  - **Initiator:** Floor_Sensor.
  - **Responses** as follows:
    - **R3a:** *Read the floor number from the floor sensor input register.* Accomplish internally to Floor_Sensor upon occurrence of event.
    - **R3b & R3c:** *Extinguish the light on the location panel for the elevator for the previous floor. & Illuminate the light on the location panel for the current floor.*
      - **Connector needed:** From Floor_Sensor to Location_Panel.
      - **Connector name:** Floor_Approach.
      - **Location_Panel Command Needed:** Update_Location_Indicator.
      - **Parameters/Variables:** New_Floor_Number, Elevator_Number.
    - **R3d:** *If the floor is listed in the destination list for the elevator, then stop the elevator at the floor and extinguish the light behind the floor number on the elevator's control panel. After stopping, remove the floor from the destination list, wait 3 seconds, then proceed to the next destination.*
    - **R3e:** *If the floor and direction are listed in the outstanding request list, then stop the elevator, at the floor. Extinguish the light behind*

the floor button on the proper request panel, and remove the summons request from the outstanding request list. After stopping, wait 3 seconds, then proceed to the next destination.

Combine R3d and R3e and then break up as follows:

* **R3di:** *if the floor is listed in the destination/direction list for the elevator then.*
    - **Connector needed:** Floor_Sensor to Scheduler.
    - **Connector name:** Floor_Approach.
    - **Scheduler Command Needed:** Floor_Approaching.
    - **Parameters/Variables:** Elevator_Number, Floor_Number, Direction.
* **R3dii:** *stop the elevator at the floor.*
    - **Connector needed:** From Scheduler to Motor.
    - **Connector name:** Arrives.
    - **Motor Command Needed:** Stop.
    - **Parameters/Variables:** Elevator_Number.
* **R3diii:** *extinguish the light behind the floor number on the elevators control panel.*
    - **Connector needed:** Scheduler to Control_Panel.
    - **Connector name:** Arrives.
    - **Control_Panel Command Needed:** Button_Light_Out.
    - **Parameters/Variables:** Elevator_Number, Floor_Number.
* **R3ei:** *extinguish the light behind the floor number on the floor panel for this direction and floor.*
    - **Connector needed:** Scheduler to Floor_Panel.
    - **Connector name:** Arrives.

        · **Floor_Panel Command Needed:** Light_Out.

        · **Parameters/Variables:** Floor_Number, Direction.

    * **R3div:** *After stopping, remove the floor from the destination list.* Done internally by Scheduler.

    * **R3dv:** *Also after stopping, wait 3 seconds, then proceed to the next destination.*

        · **Connector needed:** From Scheduler to Motor.

        · **Connector name:** Proceed.

        · **Motor Command Needed:** Go.

        · **Parameters/Variables:** Elevator_Number, Direction.

  – **Maximum response time:** 0.1 seconds.

- **Event 4:** *A passenger presses a destination button on the control panel of an elevator.*

  – **Initiator:** Control_Panel.

  – **Responses** as follows:

    * **R4a:** *Read the Control Panel input register to determine the desired floor number.* Done internally by Control_Panel.

    * **R4b:** *Illuminate the light behind the button on the Control Panel for the elevator.* Done internally by Control_Panel.

    * **R4c:** *Add the floor to the destination list for the elevator.*

        · **Connector needed:** From Control_Panel to Scheduler.

        · **Connector name:** Destination_Requested.

        · **Scheduler Command Needed:** Destination_Requested.

        · **Parameters/Variables:** Elevator_Number, Floor_Number.

    * **R4z:** (added) *if the Elevator is idle then dispatch it toward the floor selected.*

    - **Connector needed:** From Scheduler to Motor.
    - **Connector name:** Proceed.
    - **Motor Command Needed:** Go.
    - **Parameters/Variables:** Elevator_Number, Direction.
  - **Maximum response time:** 0.1 seconds.

- **Events 5 & 6:** *An elevator becomes overloaded.*
  - **Initiator:** Weight_Sensor.
  - **Responses** as follows:
    * **R5a & R6:** *Disable the elevator so that it does not move until the overload condition is gone.* Make this job part of the function of the Proceed connector (event). Before calling Go of Motor the Weight_Sensor should be called to verify that the elevator is not overweight. Proceed is initiated by the Scheduler.
    * **R5b & R6:** *Periodically (approximately every 5.0 seconds) check to see if the overload is eliminated.* Also, make this job part of the function of the Proceed connector. If the elevator was found to be overweight, then repeatedly call the Weight_Sensor (every 5 seconds) to see if the weight has changed to below the maximum. If it has then the Proceed connector can go ahead and call the corresponding Motor to Go.
  - **Maximum response time:** 0.1 seconds.

4.2.5 Elevator Object-Event Interconnection Diagram. With the completed Event-Mapping List for the Elevator problem, the Object-Event Interconnection Diagram can now be directly drawn as described in Section 3.5.5.2. Arcs from connectors to responding objects are labeled with command or data names to make the diagram more independently descriptive.

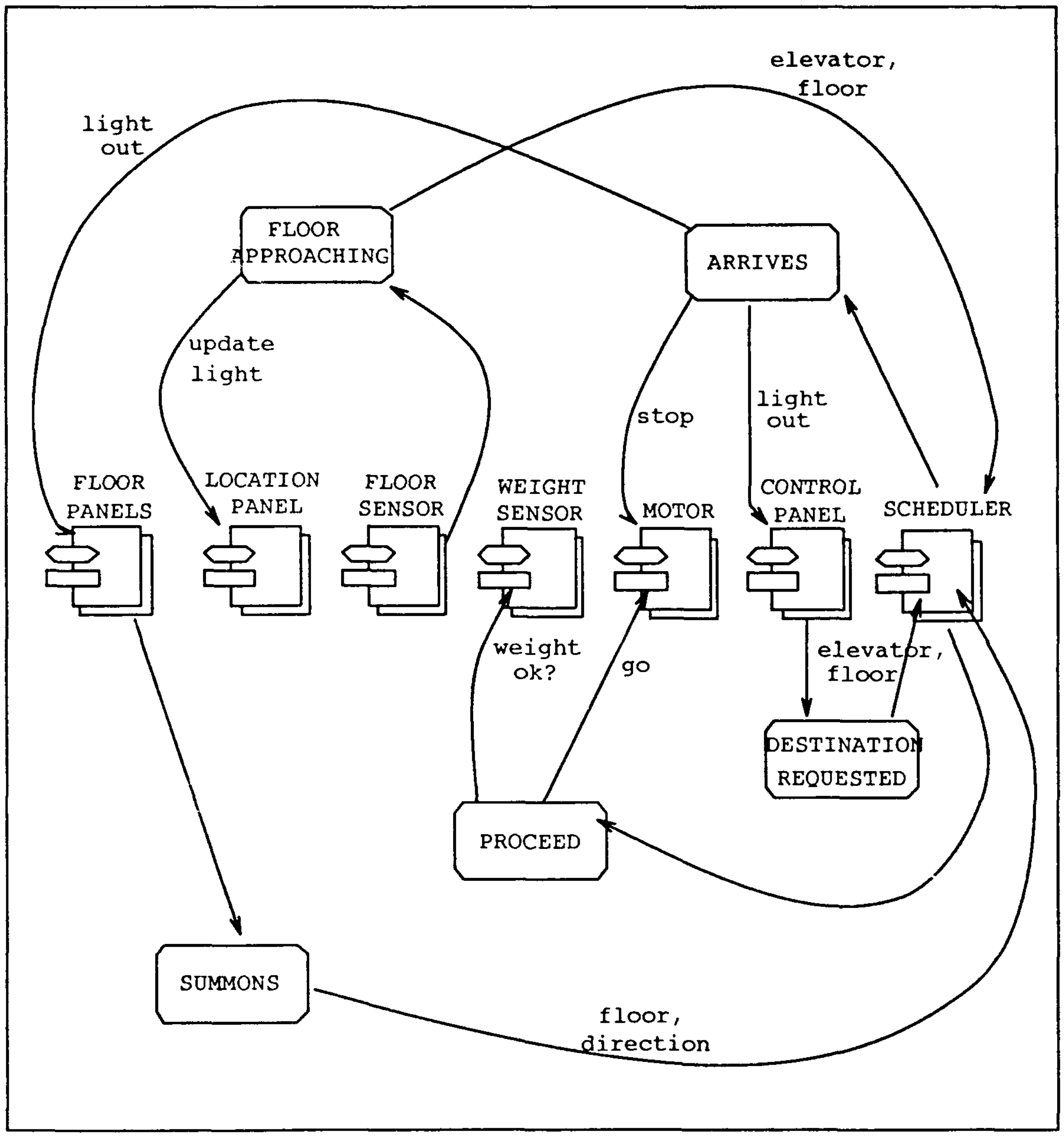

Figure 4.2. Elevator Object-Event Interconnection Diagram

4.2.6 Mapping to Ada Specifications.

Now with the first transformation step of the mapping method complete for the Elevator problem, the second step can be accomplished: mapping the results of the first step to Ada specifications following the guidance of Section 3.5.6. The result is included as Appendix A.

The specifications are presented in the order they were written. This order approximates dependency order (which is defined by the diagrams and the "with" clauses in the code). Depended upon components were written before dependent components. Standard_Engineering_Types was written first, then the hierarchy defined by the Hierarchical-Structure diagram was established by writing the object managers and system aggregate packages. The procedural representation was then established by writing the connectors. The object managers and connectors were written directly from the Object-Event Interconnection diagram.

The Object Manager Template was written after the first few object manager packages as described in Section 3.5.7. These first few Object Managers were written as prototypes and then generalized to define the template (as was done by the Granite Sentry project discussed in Section 2.6.3). The template was then used to instantiate the other object managers both for this problem and the Cruise Control Problem. See the documentation sections of the Ada specifications for details. The documentation section of the Standard_Engineering_Types package describes the configuration parameters that must be modified to instantiate the design for other applications within the domain (elevator controllers in this case).

This step completes the design of the Elevator problem. The only code remaining to be written by the implementor is the object manager package bodies.

4.3 The Cruise Control Problem

The mapping method will now be applied to the other problem analyzed by March in Chapter III of his thesis: The Cruise Control problem.

4.3.1 Important Products from March's Analysis.

The analysis items most needed by the mapping method are the *Organized Preliminary Object List*, and *Message Senders and Receivers* list, and for this problem we also need the *Event/Response List* since March didn't copy all the responses to the Message Senders and Receivers list this time. The corresponding analysis sections are copied in the next three sections for convenience:

4.3.1.1 Cruise Control Event/Response List.

Event1: The on button is pressed.
Resp.1: The cruise control system is activated.

Maximum response time: 0.5 seconds.

E2: Set speed button is pressed.
R2a: Cruise control system is engaged.
R2b: Set the desired speed equal to the current speed.

Maximum response time: 0.25 seconds.

E3: Time to update the throttle position (periodic).
R3: If engaged, then set the throttle based on the current speed vs. the desired speed.

Projected event rate: 10 / second.

E4: Brake is pressed.
R4: Cruise control system is disengaged.

Maximum response time: 0.1 seconds.

E5: Resume button is pressed.

R5: Cruise control is engaged.

Maximum response time: 0.25 seconds.

E6: Accelerate button is pushed.
R6: Increment desired speed.

Maximum response time: 0.25 seconds.

E7: The off button is pressed.
R7a: Throttle control is disengaged.
R7b: Cruise control is deactivated.

Maximum response time: 0.1 seconds.

4.3.1.2 Cruise Control Preliminary Object List.

```
Cruise Control


Throttle control


Speed
    Current Speed
    Desired Speed


Button
    Set Button
    On Button
    Off Button
    Resume Button
    Accelerate Button
```

```
Timer
```

4.3.1.3 Cruise Control Senders and Receivers of the Messages / Events.

Event1: The on button is pressed.
Sender: On Button
Receiver: Cruise Control

E2: Set speed button is pressed.
Sender: Set Button
Receiver: Cruise Control

R2a: Cruise control system is engaged. (Performed by Cruise Control)

R2b: Set the desired speed equal to the current speed. (Performed by Cruise Control)

E3: Time to update the throttle position (periodic).
Sender: Timer
Receiver: Cruise Control

E4: Brake is pressed.
Sender: Brake
Receiver: Cruise Control

E5: Resume button is pressed.
Sender: Resume Button
Receiver: Cruise Control

E6: Accelerate button is pushed.
Sender: Accelerate Button
Receiver: Cruise Control

E7: The off button is pressed.
Sender: Off Button
Receiver: Cruise Control

R7a: Throttle control is disengaged. (Performed by Cruise Control)

R7b: Cruise control is deactivated. (Performed by Cruise Control)

4.3.2 Completion of Cruise Control Hardware Interface Requirements.

The Cruise Control Analysis is sufficiently complete to use the mapping method except in the area of the hardware interface. Hardware interfacing information is needed for the Standard_Engineering_Types package. This information will be needed by the implementor of the object manager package bodies and for instantiating the design implementation for different hardware. The hardware interface information should be in the *Metarequirements* part of the analysis as it was for the elevator problem.

To demonstrate the use of the mapping method on this problem, the analysis in this area is expanded by assuming that the hardware interface works in a similar way as the elevator problem, with the following added details:

1. Each **button** has an associated interrupt (including the brake), so each "button push" generates its associated interrupt. No registers must be read in association with the interrupts. The interrupt numbers are as follows:

Set	: 16#B1#
On	: 16#B2#
Off	: 16#B3#
Resume	: 16#B4#
Accelerate	: 16#B5#
Brake	: 16#B6#

2. The **Speed Sensor** hardware has one 8-bit register. Its address is 16#A1#. The Speed Sensor hardware maintains the speed of the vehicle in this register. The speed is a hex integer and is maintained in units of miles-per-hour. The Speed_Sensor software object need only read this register to get the speed value.

3. The **Throttle Control** hardware has one 8-bit register. Its address is 16#A2#. The throttle control hardware reads this register periodically to get commands. The throttle control hardware has two states: "holding" and "release." It interprets commands as follows:

   - **Set:** 16#10# – Hold the current throttle setting and put the hardware in the "holding" state. This command is accepted in both hardware states.
   - **Speed Adjustment:** Commands to change the throttle setting. The hardware reads these commands only when in the "holding" state.

     A value of 16#FF# tells the hardware to continue holding the current throttle setting. Integer values of one to ten tell it to change the throttle by various amounts: one is the smallest amount and ten is the largest. A zero in the high-order bit means the change will be a deceleration, a one indicates an acceleration. While in the active state, if any value is read other than those listed below, the throttle control releases the throttle and returns to the release state.

     The actual command values are as follows:

     – No change: 16#FF#
     – Decelerate Increment 1: 16#01#
     – Decelerate Increment 2: 16#02#
     – Decelerate Increment 3: 16#03#
     – Decelerate Increment 4: 16#04#
     – Decelerate Increment 5: 16#05#
     – Decelerate Increment 6: 16#06#
     – Decelerate Increment 7: 16#07#
     – Decelerate Increment 8: 16#08#
     – Decelerate Increment 9: 16#09#
     – Decelerate Increment 10: 16#0A#

- Accelerate Increment 1: 16#81#
- Accelerate Increment 2: 16#82#
- Accelerate Increment 3: 16#83#
- Accelerate Increment 4: 16#84#
- Accelerate Increment 5: 16#85#
- Accelerate Increment 6: 16#86#
- Accelerate Increment 7: 16#87#
- Accelerate Increment 8: 16#88#
- Accelerate Increment 9: 16#89#
- Accelerate Increment 10: 16#8A#

4.3.3 Cruise Control Object-Mapping Table. This section maps the analysis objects from the Cruise Control problem to implementation objects. The list of analysis objects is in Section 4.3.1.2.

Following the heuristics of Section 3.5.3.1, of the 13 analysis objects, 5 are mapped to implementation objects (Table 4.2). For this embedded system, object identification follows real-world analogies. All the buttons, including the brake, will be handled by one object called Button. The "Cruise Control" object will be mapped to the "system" level in the next section and therefore is not a true object in the sense of our mapping method.

A place to maintain system states is needed, because all the objects map to hardware interfaces. In the elevator problem, the Scheduler object had sufficient information to maintain any states it needed. For this problem, however, states such as the speed that the Cruise Control is maintaining, and whether it is currently engaged or not, needs to be commonly available. For this reason we add an additional object called System_States.

ANALYSIS OBJECTS	IMPLEMENTATION OBJECTS	ATTRIBUTE/ STATE	PARAMETER
Cruise Control	X		
Throttle Control	X		
Speed Sensor	X		
Current Speed		X (Speed Sensor)	X
Desired Speed		X (Speed Sensor)	X
Button	X		
On Button		X (Button)	X
Off Button		X (Button)	X
Accelerate Button		X (Button)	X
Set Button		X (Button)	X
Resume Button		X (Button)	X
Brake Pedel		X (Button)	X
Timer	X		
	System_States		

Table 4.2. Cruise Control Object-Mapping Table

4.3.4 Cruise Control Hierarchical-Structure Diagram.

The implementation objects are now identified and are ready to be grouped into their natural hierarchy as described in Section 3.5.4.1. The objective is to group the objects into systems and, if needed, executives. The Cruise Control system is not complex enough to be decomposed into multiple executives, so no executive will exist (except perhaps as a null procedure for the purpose of linking and initial invocation).

Recognize here that the Cruise Control object it not really an object but an aggregate of objects. It is, in fact, an aggregate of all the other objects. This small application therefore maps to one "system" that is called the "Cruise Control System"(Aggregate).

Real-world analogies and Figure 3.9 from the analysis - *Cruise Control System External Interface Diagram* - are among the heuristics used in drawing the Hierarchical-Structure Diagram.

4.3.5 Cruise Control Event-Mapping List.

The next step is to develop the Event-Mapping List as described in Section 3.5.5.1. The mapping follows the *Event/Response List* and *Message Senders and Receivers* list, which are included as Sections 4.3.1.1 and 4.3.1.3. We are mainly following the Event/Response list of the analysis because the events were not reproduced in the Senders and Receivers list. In this problem the mapping creates exactly one connector for each event.

Because procedural objects do not occur in this problem, the connector for event 3 ("Update") does more than just pass messages. It does the work of coordinating the throttle and the speed when the "time to update throttle" event initiates from the timer object.

Mapping List:

- **Event 1** : *The on button is pressed.*
  - **Initiator:** Buttons.

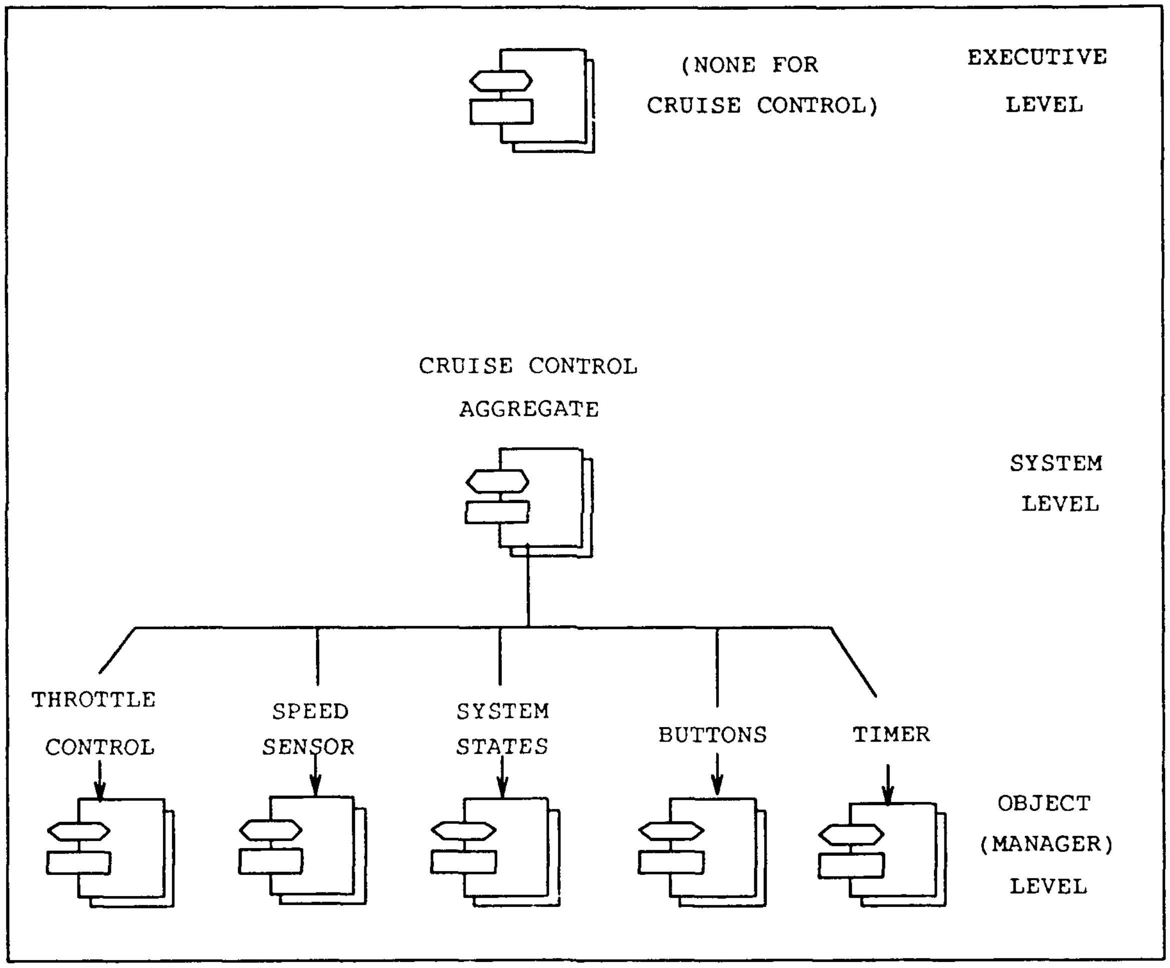

Figure 4.3. Cruise Control Hierarchical-Structure Diagram

- **Responses:** as follows:
  - **R1:** *The Cruise Control System is activated.*
    - **Connector needed:** From Buttons to System_States.
    - **Connector name:** Turn_On.
    - **Parameters/Variables:** None.
    - **System_States Command Needed:** Apply_On (sets a boolean indicating the Cruise Control is turned on).
    - **Connector Processing:** Invoke Apply_On.
- **Maximum response time:** 0.5 seconds.

- Event 2: *Set speed button is pressed.*
  - **Initiator:** Buttons.
  - **Responses** as follows:
    - **R2a:** *Cruise Control System is engaged*
      - **Connector needed:** From Buttons to System_States.
      - **Connector name:** Set_Speed.
      - **System_States Command Needed:** Return_On (returns the state of the "on" boolean), Apply_Engaged (sets a boolean indicating that the Cruise Control is engaged).
      - **Connector Processing:** Invoke Return_On; if true invoke Apply_Engaged.
      - **Parameters/Variables:** None.
    - **R2b:** *Set the desired speed equal to the current speed.*
      - **Connector needed:** From Buttons to Speed_Sensor, System_States, and Throttle_Controller.
      - **Connector name:** Set_Speed.

- **Speed_Sensor Command Needed:** Return_Speed (returns the current speed).
- **Throttle_Controller Command Needed:** Set (tells the throttle_controller to hold at the curr^a^t setting).
- **System_States Com. - ınd Needed:** Apply_Desired_Speed (loads a variable maintaining the desired speed).
- **Connector Processing:** Invoke Set, Invoke Return_Speed in Speed_Sensor and then invoke Apply_Desired_Speed with the speed value.
- **Parameters/Variables:** Current_Speed.

– **Maximum response time:** 0.25 seconds.

- **Event 3:** *Time to Update the throttle position (periodic).*

  – **Initiator:** Timer.

  – **Responses** as follows:

  * **R3:** *If engaged, then set the throttle based on the current speed vs. the desired speed.*

    Connector needed: From Timer to Speed_Sensor, System_States, and Throttle_Controller.

    - **Connector name:** Update.
    - **Speed_Sensor Command Needed:** Return_Speed (returns the current speed of travel).
    - **Throttle_Contı ller Command Needed:** Change_Throttle_Setting (this command should have a parameter to indicate if an acceleration or deceleration is desired, and a scalar parameter to indicate the relative amount of change needed).

        - **System_States Command Needed:** Return_Engaged (returns boolean), Return_Desired_Speed (returns the desired speed as it was last set).
        - **Connector Processing:** While Return_Engaged returns true the connector should loop until Return_Desired_Speed is equal to Return_Speed. If a change to Change_Throttle_Setting is needed, use a large increment if the speed is off by a large amount, and a small increment if the speed is close. Delay and reread Current_Speed for each loop, adjust the speed as necessary. Check Return_Engaged frequently, abort if it changes to false. Refine to make for a smooth speed adjustment.
        - **Parameters/Variables:** Desired_Speed, Acceleration / Deceleration scalar.
  - **Maximum response time:** Update every 10 seconds, but do not invoke if the last invocation has not completed.

- Event 4: *Brake is pressed.*
  - **Initiator:** Buttons.
  - **Responses** as follows:
    - **R4:** *Cruise Control System is disengaged.*
        - **Connector needed:** From Buttons to Throttle_Control and System_States.
        - **Connector name:** Brake_Pressed.
        - **Throttle_Controller Command Needed:** Release (Releases the throttle completely).
        - **System_States Command Needed:** Disengage (sets an engaged boolean to false).
        - **Connector Processing:** Invoke Release, Invoke Disengage.

        · **Parameters/Variables:** None.

  - **Maximum response time:** 0.1 seconds.

- **Event 5:** *Resume button is pressed.*
  - **Initiator:** Buttons.
  - **Responses** as follows:
    - **R5:** *Cruise Control System is engaged.*
      - · **Connector needed:** From Buttons to System_States.
      - · **Connector name:** Resume.
      - · **System_States Command Needed:** Apply_Engaged.
      - · **Connector Processing:** Invoke Apply_Engaged (note: speed will be adjusted next time the timer invokes).
      - · **Parameters/Variables:** None.
  - **Maximum response time:** 0.25 seconds.

- **Event 6:** *Accelerate button is pushed.*
  - **Initiator:** Buttons.
  - **Responses** as follows:
    - **R6:** *Increment Desired Speed.*
      - · **Connector needed:** From Buttons to System_States and Throttle_Controller.
      - · **Connector name:** Accelerate.
      - · **Throttle_Controller Command Needed:** Change_Throttle_Setting.
      - · **System_States Command Needed:** Return_Engaged.

        - **Connector Processing:** Invoke Return_Engaged, if true then Invoke Change_Throttle_Setting by one small increment in the acceleration direction.
        - **Parameters/Variables:** None.
  - **Maximum response time:** 0.25 seconds.

- **Event 7:** *The off button is pressed.*
  - **Initiator:** Buttons.
  - **Responses** as follows:
    - **R7a:** *Throttle control is disengaged.*
      - **Connector needed:** From Buttons Throttle_Controller.
      - **Connector name:** Turn_Off.
      - **Throttle_Controller Command Needed:** Release.
      - **Connector Processing:** Invoke Release.
      - **Parameters/Variables:** None.
    - **R7b:** *Cruise Control System deactivated.*
      - **Connector needed:** From Buttons to System_States.
      - **Connector name:** Turn_Off.
      - **System_States Command Needed:** Disengage, Apply_Off.
      - **Connector Processing:** Invoke Disengage, invoke Apply_Off.
      - **Parameters/Variables:** None.
  - **Maximum response time:** 0.1 seconds.

4.3.6 Cruise Control Object-Event Interconnection Diagram.

With the completed Event-Mapping List for the Cruise Control problem, the Object-Event Interconnection Diagram can now be directly drawn as described in Section 3.5.5.2. Arcs from connectors to responding objects are labeled with command or data names to make the diagram more independently descriptive.

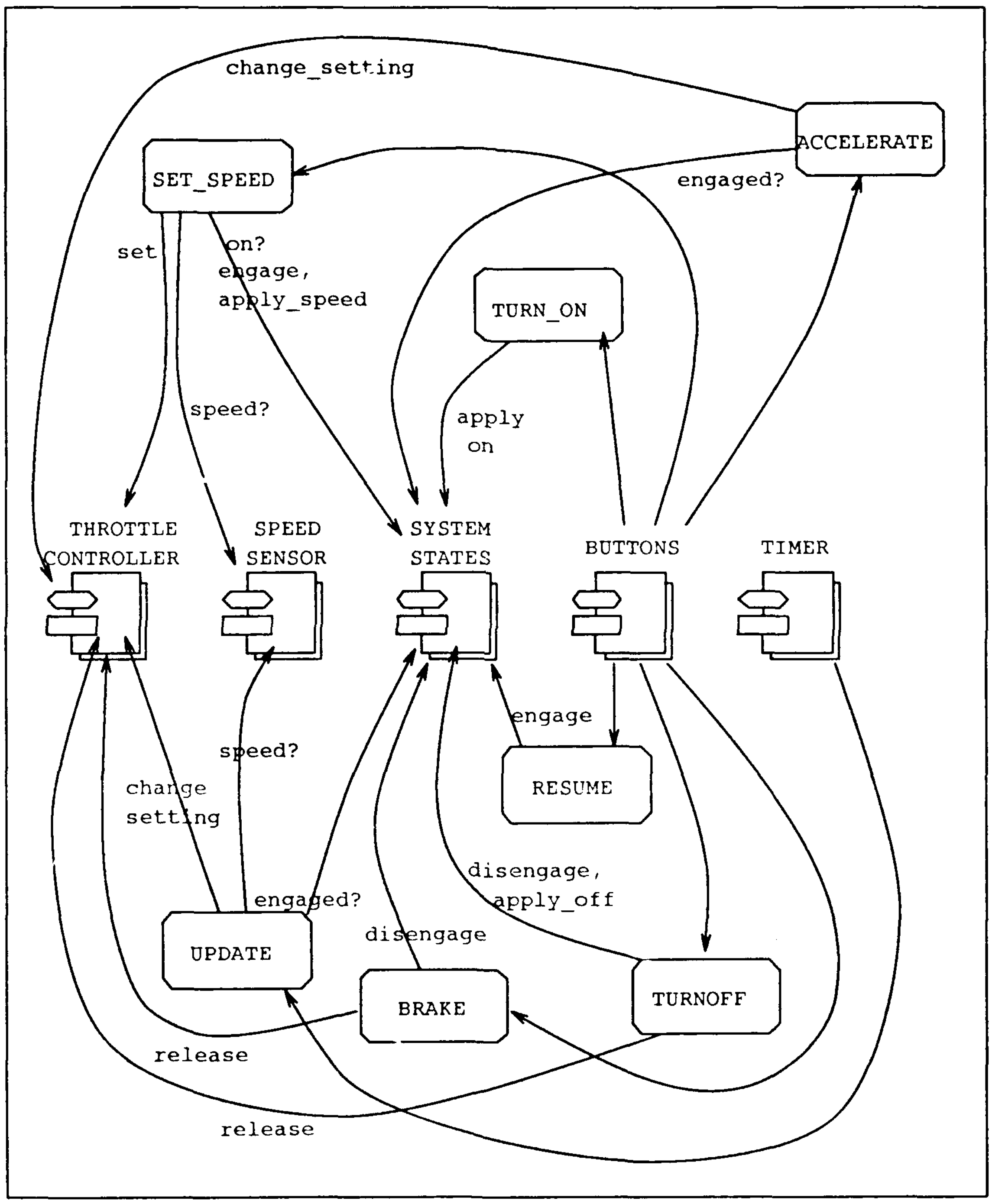

Figure 4.4. Cruise Control Object-Event Interconnection Diagram

4.3.7 Mapping to Ada Specifications.

Now with the first transformation step of the mapping method complete for the Cruise Control problem, the second step can be accomplished: mapping the results of the first step to Ada specifications following the guidance of Section 3.5.6. The result is included as Appendix B. The Ada specifications were developed using the same process as for the elevator problem described in Section 4.2.6 and the same comments apply.

4.4 Analysis

All the advantages of design reuse presented in Section 3.4, which can be shown through design, are evident in these designs, as follows:

- The two designs are very similar. Both designs follow the same structure pattern, use the same concepts and principles, and are represented using the same tools. The same object template was used to instantiate the object managers for both problems.
- The second problem was developed much more quickly than the first owing to design pattern reuse, reuse of the object template, and reuse of other design constructs.
- Each design can be instantiated within its somewhat limited domain.
- The objects have high potential for reuse because of very-low coupling. Each depends only on the Standard_Engineering_Types package, and a connector procedure if the object is an event initiator. However, many of the objects have hardware dependencies. "Swapping out" the implementation part of an object should also prove to be easy because of low coupling.
- Implementation will be easier because no direct compilation dependencies exist between the objects. Each object can be developed in isolation of the others.

- The implementations should prove efficient because message passing employs only one intermediary: a connector. No hierarchical bottlenecks are encountered in moving a message from initiator to responder.
- The designs are object oriented and closely resemble the real-world problem.

A fundamental benefit of these designs is consistency, both internally and between designs. An implementor or maintenance programmer who was familiar with one of these designs could quickly become familiar with the other. Similarly, a development organization having developed an application using one of these designs could quickly develop an application using the other. Also, because of internal consistency, a programmer familiar with one part of the design, could easily become familiar with another part.

4.5 Suggestions for Design Implementation

The implementor of the foregoing designs will need to write the package bodies for the object managers. Also, because no main driver is inherit in the design, the implementor will need to write a main driver procedure for the purpose of compiling, linking, loading, and invoking the system as a unit. This driver only needs to be a null procedure that directly and indirectly "with"s the other components. For example, this procedure only needs to with the aggregate packages, the rest of the components will be included since all object managers are withed by the aggregates, each connector is withed by at least one (usually exactly one) object manager; the Standard_Engineering_Types package is withed by the object managers also. Similarly, if executives exist, then the main procedure need only with these executives.

4.6 Simulation Implementation of the Elevator Design

As an additional validation step, the elevator design presented in Section 4.2 and Appendix A was implemented as a simulation. The implementation demonstrates that a directly usable design results from application of the mapping method.

The simulation was developed in Ada on a personal computer and targets to the same. The design code from Appendix A was used; implementation of the package bodies is included as Appendix C.

The simulation runs in real time for a realistic simulation; that is, it runs interactively like an actual elevator system. Images of the elevators and buttons appear on the simulation screen. The simulation operator can enter summons requests and destination requests from the keyboard at any time during the simulation. The elevators can be seen to move up and down and the appropriate buttons are illuminated on the screen in response to requests. Button lights are extinguished when the elevator arrives. The simulation runs until the operator quits.

The elevator design was instantiated for 16 floors and four elevators using the *Number_of_Elevators* and *Number_of_Floors* parameters in CONFIGURATION AREA #1: of the Standard_Engineering_Types package. All the hardware addresses, interrupts, and command values in this package were commented out because they were not needed for the simulation.

A few minor changes had to be made to the design code in Appendix A. Procedures had to be added to some of the object managers to supplement entry statements assigned to interrupts. For example, the entry waiting for an interrupt from the floor sensor had to be supplemented with an exported procedure to receive floor-approaching messages from the motor simulator. Another minor change was necessitated by a perceived elaboration order problem by the binder that came with the compiler. The system-aggregate packages were using the pragma "elaborate" to ensure that they were elaborated last because they load data into their data structures at elaboration time. The change was to load the data using an initialize procedure instead.

Implementation was accomplished in about 5 working days. The package body for the simulation screen controller was written by another student. A semi-colon count to approximate Lines-Of-Code (LOC) was conducted and the results follow:

Standard Engineering Types (no body)	: 33	LOC
Object Manager Specifications	: 110	LOC
System Aggregate Packages	: 30	LOC
Connectors	: 65	LOC
Design Code TOTAL	: 238	LOC
Object Manager Bodies	: 333	LOC
Mapping Method TOTAL	: 571	LOC
Simulation Driver	: 98	LOC
Screen Controller Specification	: 13	LOC
Screen Controller Body	: 222	LOC
Simulation Grand TOTAL	: 904	LOC

The results of the simulation implementation are very encouraging. Implementation of the design was quick and easy. The Ada design code produced by the mapping method was used with no significant change. The design was easily instantiated for the numbers of floors and elevators. By far the bulkiest of the object-manager bodies was the Scheduler manager. This package appears to have high potential for reuse, at the component level, due to its simple, straight-forward interfaces. The implementation of the elevator problem demonstrates that the mapping method produces a design that works.

V. Conclusions and Recommendations

5.1 Summary of Contribution

5.1.1 Identification of Design Reuse Importance, Benefits, and Characteristics. This thesis seeks to refocus the emphasis of reuse research from small component reuse to design reuse. Chapter I identified the importance of design reuse for improving software development productivity. Chapter II developed a better definition of the idea through literary definitions, discussions of related issues, and presentations of ground-breaking research and development in this area. Chapter II also discussed characteristics a reusable design should exhibit to support smaller component reuse as a side benefit in Section 2.7. Chapter III enumerated advantages and characteristics of a particular reusable design developed at the Software Engineering Institute in Section 3.4. An important goal of this thesis is to define design reuse and to push the software development community toward evolving the technology in this area.

5.1.2 A Mapping Method for Consistent, Reusable Designs. A specific contribution of this thesis is to present a method for mapping different problems in the same application domain to similar design solutions. The fundamental idea is that different software problems within the same domain should have very similar design solutions; such designs become candidates for reuse within the domain.

This method directly addresses the problem statement of Chapter I, which argues that design reuse is the threshold that needs to be broken to gain true advances in the area of increased software-development productivity through reuse.

The method presented is suitable for embedded, event-driven software problems. The method maps from the products of March's Object-Oriented Analysis method to a design based very closely on the principles of the OOD-Paradigm [March, 1989], [Rissman and others, 1988].

The method is validated by applying it to two problems and recognizing the similar results, by assuming that a design following the principles of the OOD-Paradigm will exhibit the benefits of Section 3.4, and by implementing one of the resulting designs.

5.1.3 A Method of Design Representation.

In Chapter I, Biggerstaff and Richter were quoted as complaining that no method existed for representing designs in such form that they could be reused (like code is reused, see Section 1.1.1). This problem was addressed in the following ways:

1. Creation of the Hierarchical-Structure diagram and the Object-Event Interconnect Diagram. A purpose of these diagrams is to force designs within the domain to look the same. These diagrams lead to the same pattern from application to application. The corresponding Ada specifications also look the same.
2. Development of the method of following a design-pattern model to help in generalizing a reusable design. The model used for the mapping method was an event-response model.
3. Adoption of the Object Template from the OOD-Paradigm. The Object Template is a design-component generic.
4. Adoption of the Standard_Engineering_Types package from the OOD-Paradigm. The Standard_Engineering_Types package was expanded here to contain specific configuration areas for instantiating an implementation within a domain of application.

5.1.4 Designs That Are Quick to Implement.

Because of the consistent design patterns between object managers, implementation of the elevator design was fast and painless. As described in Section 4.6, implementation of the elevator design required only 5 days for about 670 lines of code (including 98 lines for the driver).

This time period included integration with the screen simulator, testing, debugging, and time to install and become familiar with the compiler.

Implementation was speeded because very little design documentation had to be referenced beyond the documentation in the specifications of the object managers. Many of the implementation chores became repetitious and mechanical. Some creativity was required, but none beyond the basic skills expected of a competent programmer. Implementation did not require deviation from the design.

The consistent design patterns inherent in these designs, owing to the goal of design reuse, can be concluded to enhance the implementation process.

See also Section 1.5, "Maximum Expected Gain," for more information on the contribution of this thesis.

5.2 Related Further Research

This thesis breaks ground in the area of design reuse; many related issues remain to be explored. These issues include both how these generalized solutions will perform and in finding these general solutions.

5.2.1 Application to Larger Systems. Today's software systems are large and problems of complexity multiply with larger systems. The real test of any development method would be to have it succeed for a large system [Booch, 1991:pp2-23].

To handle large applications, the mapping method and OOD-Paradigm are based on the idea of having two levels of aggregation: the system level and the executive level. The two example problems solved were not sufficiently large to utilize more than one executive, therefore no executives were defined. The principle of the executive, as pioneered by the SAE team at the SEI, is to distribute each executive onto its own processor. This means that the connector concept may have to be expanded since distribution would likely mean that the executives would have to communicate via networks. Despite this, most message passing would still be conducted at the lower levels of abstraction due to a basic interaction principle: "interactions inside subsystems are more frequent than interactions between subsystems" (Booch quotes Courtois [Booch, 1987:pp556]).

Even without the concern of distribution, larger systems may cause the number of connectors to become unruly. A few possible approaches would be to group connectors in packages or if the number of systems becomes very large some simplicity may be restored by having the aggregate packages provide the additional function of the system-level connectors.

These and other concerns related to application to larger systems need to be explored. Large system development should be an important area of research at AFIT.

5.2.2 Timing and Sizing Studies. Experience has demonstrated that a major shortcoming in the software engineering community is an inability to predict the satisfaction of nonfunctional requirements from a design (to get the same experience become a regular reader of the comp.lang.ada bulletin board.). In particular, Ada developers are developing the reputation for not being able to predict, or meet, timing and sizing constraints. This factor is often the justification cited in requests for Ada waivers.

Prediction of satisfaction of nonfunctional requirements from design is certainly an area that could benefit greatly from design reuse. Once a design has been implemented, a good basis exists for predicting the timing and sizing characteristics for further uses of the design.

Of course, the design must follow good principles of performance engineering design to begin with. A few leads for research in this area follow:

- The SEI series book: "Performance Engineering of Software Systems" by Connie U. Smith [Smith, 1990:pp33-110], presents many important principles of performance engineering that are applicable at the design stage. A good suggestion might be to critique the OOD-Paradigm and mapping-method designs against the "Principles for Creating Responsive Software" in chapter 2 of her book.

  Intuitively it would appear that, due to flatness of the resulting designs, the mapping-method designs would fare well against at least the "independent" type of principles listed in table 2.1 of her book.

- Also, the mapping-method designs appear amenable to performance measurements owing to the flat architecture and the reuse aspects of the design and components. Because the designs consist of communicating components at a single level and reused objects may have known performance characteristics, measurements may be possible by summing the values for each object and then adding some overhead factor.

5.2.3 Categorizing Reusable Designs by Application Domains. Not all software solutions can be made to fit the same general solution. Different kinds of software problems will map to different kinds of general solutions. Brown and Quanrud make the following statement in this regard:

> A generic architecture is not intended for use outside of its specific domain. The expectation is that a separate architecture will be needed for each different application domain. [Brown and Quanrud, 1988:pp390]

Plinta and Lee, who call reusable designs "models," point out that many of these "models" need to be accumulated. They summarize a method of arriving at them in the following quote:

> To realize these payoffs [from Design Reuse], model databases must be populated ... First, domain experts need to identify reoccurring problems in their domains ... Second, model solutions need to be developed and verified ... Verification is based on both functionality and performance ... After the solutions are verified, the prototype solutions are generalized to produce code templates and generics. The templates and generics help to insure that each instantiation of the model provides the functionality specified by the model. They also promote code and comment consistency. These characteristics encourage reuse. [Plinta and Lee, 1989:pp66]

Chapter II demonstrated that general designs from two different domains can appear quite different. Our suggestion for further research is to categorize general solutions by domains. With such information in hand, a developer would have a good idea of what the design patterns should look like very early in the development just by knowing the problem's application domain

5.3 Suggestions For March's Analysis Method.

5.3.1 Make Requirements Tracing Easier. Overall March's analysis method was found to be quite satisfactory. However, there were some difficulties using March's products in the area of requirements traceability. Paragraph 4.2.6 of [DoD-STD 2167A, 1985:pp14] calls for traceability of requirements to design. Traceability implies some way of labeling and/or enumerating requirements. Of the products of March's Analysis, only the Event/Response list and the Message Senders and Receivers list enumerate requirements. Indeed, these lists are the key to tracing requirements from his products and are the major products used by the mapping method. Our concern is that all the other products of March's analysis (listed in Section 3.2) do not have ready labels or numbers that can be used for tracing.

During the mapping process documented in Chapter IV of this thesis, the major benefit of much of the information in March's analysis was to enhance familiarity with the problem, but they were not necessarily mapped directly to design. Also, a good deal of overlap occurs from one product to another because many of the products are actually intermediate steps used to understand the problem and to derive the other products. For example, his analysis is bulging with concept maps, but most of these do not map directly to design, although used in combination with other products they did help in making design mapping decisions.

The problems with traceability come in when trying to decide which products need to be mapped to design, and how to label them. Which products are traced and which are there for the purpose of enhancing understanding? We choose to directly map the Event/Response list, the Message Senders Receivers list, the Preliminary Object List, and the Metarequirements. Perhaps the analysis should be more clear on which products are to be traced, which are intermediate products, and which are for the purpose of enhancing understanding of the problem.

5.3.2 Other Possible Uses.

March's Object-Oriented analysis actually does more than just define user requirements. It defines every concept from the concept maps as an object and attempts to define all useful suffered operations for each object. For this reason, it could probably be used as a domain analysis tool.

Because the method defines a structure between objects and operations for objects, it also could be considered a high-level design method as eluded to in [Umphress, 1990]. However, if the design from the analysis were used, the result would be subject to the same complaints raised in the problem statement of Chapter I.

5.4 Closing Remarks

Design reuse needs to become common practice in the software-engineering community to help propel us toward overcoming the symptoms of the software crisis. Reusable designs should be developed using principles developed in this thesis to achieve the benefits listed in Section 3.4. Some of these benefits of design reuse are greater development productivity; less documentation, maintenance, and testing effort; and greater reliability. The mapping method presented in this thesis leads to consistently-structured designs that exhibit the potential for all these benefits. The mapping method breaks new ground in the area of providing a reusable representation for designs. The mapping method presented here yields a basis for application and further study in the area of reusable designs.

Appendix A. *Ada Specifications for the Elevator Problem*

A.1 Standard_Engineering_Types

```
with System;
package Standard_Engineering_Types is

--This package serves two purposes:
-- 1. It defines types for parameters that are used in the elevator
--    control system. These parameters were mapped from the
--    Object-Mapping Table that mapped all the objects defined in the
--    analysis. Areas marked "PARAMETER AREA #X:" contain parameters
--    that were mapped from the analysis.
--
-- 2. It configures the design for reusability. The areas marked:
--    "CONFIGURATION AREA #X:" contain data used to instantiate the
--    design. The parameters that can be instantiated are:
--       --The number of elevators.
--       --The number of floors.
--       --The weight capacity of each elevator.
--       --All the hardware interface values, including:
--           --The interrupt vectors for each elevator, including:
--               --The floor sensor interrupt vector.
--               --The control panel interrupt vector.
--           --The register address for each elevator, including:
--               --The weight sensor register address.
--               --The control panel input register address.
--               --The control panel output register address.
--               --The floor sensor input register address.
--               --The location panel output register address.
--               --The motor control register address.
--           --The Motor Control Commands for each elevator, including:
--               --Motor Up command.
--               --Motor Down command.
--               --Motor Stop command.
--
--           --All the following values for the floor summons panels:
--               --The Up Interrupt vector.
--               --The Down Interrupt vector.
--               --The up input register address.
--               --The down input register address.
--               --The up output register address.
--               --The down output register address.
--
```

```
-- This means that the elevator control system can be configured for
-- different elevator hardware systems just by instantiating these
-- values.
--
--This design could be expanded to have multiple Schedulers and Summons
--Controllers.
--
--
--
--       --CONFIGURATION AREA #1:
--
Number_of_Elevators               : constant := 4;
Number_of_Floors                  : constant := 40;
--Number_of_Floor_Panel_Controllers : constant := 1;
--Number_of_Schedulers              : constant := 1;



       --PARAMETER AREA #1:

type Elevator_ID_Type    is range 1..Number_of_Elevators;
--type Floor_Panel_ID_Type is range 1..Number_of_Floor_Panel_Controllers;
--type Scheduler_ID_Type   is range 1..Number_of_Schedulers;

type Floor_Type       is range 1..Number_of_Floors;
subtype Weight_Type   is integer;
type Direction_Type   is (up, down, parked);



       --CONFIGURATION AREA #2:

--Bits : constant :=1;
--type Byte_Type is range 16#00#..16#FF#;
--for Byte_Type'size use 8*Bits;



       --PARAMETER AREA #2:

--I'd prefer to make these "Derived Types"
subtype Interrupt_Num_Type    is System.Address;
subtype Register_Address_Type is System.Address;
subtype Command_Byte_Type     is System.Address;
```

```
type Elevator_Data is record
  Control_Panel_Interrupt         : Interrupt_Num_Type;
  Floor_Sensor_Interrupt          : Interrupt_Num_Type;

  Weight_Sensor_Register          : Register_Address_Type;
  Control_Panel_Input_Register    : Register_Address_Type;
  Control_Panel_Output_Register   : Register_Address_Type;
  Floor_Sensor_Input_Register     : Register_Address_Type;
  Location_Panel_Output_Register : Register_Address_Type;
  Motor_Control_Register          : Register_Address_Type;

  Motor_Command_Up                : Command_Byte_Type;
  Motor_Command_Down              : Command_Byte_Type;
  Motor_Command_Stop              : Command_Byte_Type;
  Weight_Capacity_Hundreds        : Weight_Type;

end Record;

type Elevator_Data_Array_Type is array(Elevator_ID_Type) of Elevator_Data;
Elevator_Data_Array : Elevator_Data_Array_Type;


type Floor_Panel_Data_Type is record
  Up_Interrupt          : Interrupt_Num_Type;
  Down_Interrupt        : Interrupt_Num_Type;

  Up_Input_Register     : Register_Address_Type;
  Down_Input_Register   : Register_Address_Type;
  Up_Output_Register    : Register_Address_Type;
  Down_Output_Register : Register_Address_Type;

end record;

Floor_Panel_Data : Floor_Panel_Data_Type;

end Standard_Engineering_Types;
--=====================================================================
package body Standard_Engineering_Types is


begin
--Configure Elevator Data by plugging in the Interrupt Vectors, Register
--Address, and Hardware Commands:
      --CONFIGURATION AREA #3:


--First Elevator:
  ------------------------------------------------------
  Elevator_Data_Array(1).Control_Panel_Interrupt         := 16#01#;
  Elevator_Data_Array(1).Floor_Sensor_Interrupt          := 16#05#;

  Elevator_Data_Array(1).Weight_Sensor_Register          := 16#31#;
  Elevator_Data_Array(1).Control_Panel_Input_Register    := 16#35#;
  Elevator_Data_Array(1).Control_Panel_Output_Register   := 16#39#;
  Elevator_Data_Array(1).Floor_Sensor_Input_Register     := 16#41#;
  Elevator_Data_Array(1).Location_Panel_Output_Register  := 16#45#;
  Elevator_Data_Array(1).Motor_Control_Register          := 16#51#;

  Elevator_Data_Array(1).Motor_Command_Up                := 16#01#;
  Elevator_Data_Array(1).Motor_Command_Down              := 16#02#;
  Elevator_Data_Array(1).Motor_Command_Stop              := 16#03#;
  Elevator_Data_Array(1).Weight_Capacity_Hundreds        := 100;
                                                   --(10,000) lbs
  ------------------------------------------------------


--Second Elevator:
  ------------------------------------------------------
  Elevator_Data_Array(2).Control_Panel_Interrupt         := 16#02#;
  Elevator_Data_Array(2).Floor_Sensor_Interrupt          := 16#06#;

  Elevator_Data_Array(2).Weight_Sensor_Register          := 16#32#;
  Elevator_Data_Array(2).Control_Panel_Input_Register    := 16#36#;
  Elevator_Data_Array(2).Control_Panel_Output_Register   := 16#3A#;
  Elevator_Data_Array(2).Floor_Sensor_Input_Register     := 16#42#;
  Elevator_Data_Array(2).Location_Panel_Output_Register  := 16#46#;
  Elevator_Data_Array(2).Motor_Control_Register          := 16#52#;

  Elevator_Data_Array(2).Motor_Command_Up                := 16#01#;
  Elevator_Data_Array(2).Motor_Command_Down              := 16#02#;
  Elevator_Data_Array(2).Motor_Command_Stop              := 16#03#;
  Elevator_Data_Array(2).Weight_Capacity_Hundreds        := 100;
                                                   --(10,000) lbs
  ------------------------------------------------------

--Third Elevator:
  ------------------------------------------------------
  Elevator_Data_Array(3).Control_Panel_Interrupt         := 16#03#;
  Elevator_Data_Array(3).Floor_Sensor_Interrupt        := 16#07#;

  Elevator_Data_Array(3).Weight_Sensor_Register        := 16#33#;
  Elevator_Data_Array(3).Control_Panel_Input_Register  := 16#37#;
  Elevator_Data_Array(3).Control_Panel_Output_Register := 16#3B#;
  Elevator_Data_Array(3).Floor_Sensor_Input_Register   := 16#43#;
  Elevator_Data_Array(3).Location_Panel_Output_Register := 16#47#;
  Elevator_Data_Array(3).Motor_Control_Register        := 16#53#;

  Elevator_Data_Array(3).Motor_Command_Up              := 16#01#;
  Elevator_Data_Array(3).Motor_Command_Down            := 16#02#;
  Elevator_Data_Array(3).Motor_Command_Stop            := 16#03#;
  Elevator_Data_Array(3).Weight_Capacity_Hundreds      := 200;
                                                --(20,000) lbs
  -----------------------------------------------------

--Forth Elevator:
  -----------------------------------------------------
  Elevator_Data_Array(4).Control_Panel_Interrupt       := 16#04#;
  Elevator_Data_Array(4).Floor_Sensor_Interrupt        := 16#08#;

  Elevator_Data_Array(4).Weight_Sensor_Register        := 16#34#;
  Elevator_Data_Array(4).Control_Panel_Input_Register  := 16#38#;
  Elevator_Data_Array(4).Control_Panel_Output_Register := 16#3C#;
  Elevator_Data_Array(4).Floor_Sensor_Input_Register   := 16#44#;
  Elevator_Data_Array(4).Location_Panel_Output_Register := 16#48#;
  Elevator_Data_Array(4).Motor_Control_Register        := 16#54#;

  Elevator_Data_Array(4).Motor_Command_Up              := 16#01#;
  Elevator_Data_Array(4).Motor_Command_Down            := 16#02#;
  Elevator_Data_Array(4).Motor_Command_Stop            := 16#03#;
  Elevator_Data_Array(4).Weight_Capacity_Hundreds      := 200;
                                                --(20,000) lbs
  -----------------------------------------------------


--Configure Summons Panel Data by plugging in the Addresses:

  ------------------------------------------------
  Floor_Panel_Data.Up_Interrupt          := 16#0A#;
  Floor_Panel_Data.Down_Interrupt        := 16#0B#;

  Floor_Panel_Data.Up_Input_Register     := 16#4A#;
  Floor_Panel_Data.Down_Input_Register   := 16#4B#;
  Floor_Panel_Data.Up_Output_Register   := 16#4C#;
  Floor_Panel_Data.Down_Output_Register := 16#4D#;
  --------------------------------------------------

end Standard_Engineering_Types;
```

A.2 Object_Manager_Template

```
with Standard_Engineering_Types;
--with System --This with is needed if the object will receive
              --interrupts.

--with <List of other "withed" components needed by package body. This
        list should contain one or more connectors. This list should
        be deleted upon implementation of the package body>;

package <Object>_Manager is

    package SET renames Standard_Engineering_Types;

--------------------> OBJECT REQUIREMENTS <--------------------------

         STIMULUS SUMMARY:
         ---------------------------------------------------
         List stimuli and their sources (i.e. all hardware
         interrupts, all calls to exported functions/procedures).

         RESPONSE SUMMARY:
         ---------------------------------------------------
         Copy list from stimulus summary above and explain the
         following for each:

            - Response to stimulus.
            - Whether the response is conducted internal to the object
              manager or external. Sending messages to other objects
              or software events outside the control of this object
              should be considered external. Writing to hardware under
              direct management of the object, or reading data from
              the specification of another package should be
              considered internal.

            - Include references to the exported operations (below),
              their parameters, and the fields of the
              <Object>_Representation. It should be apparent what
            *all* these things are for.


-- Here is an example of the format:



--      STIMULUS SUMMARY:
--      ---------------------------------------------------
--      1. Receives a call to create a new instance of the object.
--
--         Source of Stimulus: call to exported procedure
--                             "New_<Object>."
--
--            .
--            .
--        Continued list of stimuli
--            .
--            .
--
--      RESPONSE SUMMARY:
--      ---------------------------------------------------
--      1. Receives a call to create a new instance of the object.
--
--         a. Internal Response:
--
--            i. Uses the interrupt value to create a new
--               instance of the task; it does this by assigning the
--               interrupt value to the Floor_Sensor_Interrupt_Num
--               variable before creating the new instance.
--               Maintains copies of the incoming register value
--               and Elevator_ID value for later use.
--
--
--            ii. Returns an instance of the object to the caller.
--
--
--         b. External Response:
--
--            i. none.
--
--
--
--      2. Copied Stimulus #2 from above
--           .
--              .
--          Continued List of Responses for each stimulus.
--              .
--              .
--
--
--------------------> END OBJECT REQUIREMENTS <---------------------



--------------------> STATES MAINTAINED <-------------------------

    --All the fields in the <Object>_Representation.
                         .
                         .
                         .



----------------- -- OBJECT DEFINITION <-------------------------


  type <object>_Type is private;


--------------------> EXPORTED OPERATIONS <-------------------------



  function New_<Object>(<System_Level>_ID : SET.<System_Level>_ID_Type;
                          Data needed to fill in fields of the
                          <Object>_Representation);
                                                  return <Object>_Type;

  procedure Apply_<Stimulus>_To(This_<Object> : in <Object>_Type;
                                 <Parameters>  : in <Parameter>_Type;
                                              .
                                              .
                                              .                      );


  function Return_<State>_From(This_<Object> : in <Object>_Type)
                                              return <State>_Type;
```

```
private
--if no interups:
     type <Object>_Representation is record
       <System_Level>_ID : SET.<System_Level>_ID_Type;
                    .
                    .--States and attributes needed for this instance of the
                     --object.
                    .
     end record;
--if this object will receive interrupts:

     <Kind>_Interrupt_Num : SET.Interrupt_Num_Type;

     task type <Object>_Representation is
        entry Initialize
                   (Up_Input_Register    : in SET.Register_Address_Type;
                         .
                         .
                     <Data needed by object>);

        entry <Kind>_Interrupt;
        <entrys to carry out required operations>;

        for <Kind>_Interrupt use at <Kind>_Interrupt_Num;
      end <Object>_Representation;

           --The full definition may be moved to the package body
           --after implementation of the body is complete.

   type <Object>_Type is access <Object>_Representation;
          --pointer to a <Object>_Representation

end <Object>_Manager;
```

A.3 Object_Managers

A.3.1 Floor_Panel_Manager.

```
with Standard_Engineering_Types;
with System;

--with Summons (connector); --needs to be "withed" from
                            --the package body.

package Floor_Panel_Manager is
package SET renames Standard_Engineering_Types;


--------------------> OBJECT REQUIREMENTS <-------------------------
--
--      STIMULUS SUMMARY:
--      ---------------------------------------------------
--      1. Receives a call to create a new instance of the object.
--
--         Source of Stimulus: call to exported procedure
--                             "New_Floor_Panel."
--
--      2. Receives an interrupt indicating that a summons button was
--         pressed from one of the floors.
--
--         Source of Stimulus : interrupt from hardware
--                              (Up/Down_Interrupt).
--
--      3. Receives a call indicating that the light under one of the
--         summons buttons should be extinguished.
--
--         Source of Stimulus : call to exported procedure
--                              (Apply_Light_Out_To).
--
--      RESPONSE SUMMARY:
--      ----------------------------------------------------
--      1. Receives a call to create a new instance of the object.
--
--         a. Internal Response:
--
--            i. Uses the two interrupt values in creating a new
--               instance of the task; it does this by assigning the
--               interrupt values to the Up/Down_Interrupt_Address
--               variables before creating the new instance.
--
--            ii. Uses correspondingly named fields of the
--                Floor_Panel_Data record to initialize the task
--                using the "initialize" entry. The Floor_Panel_Data
--                record is in the Standard_Engineering_Types
--                package.
--
--            iii. Returns an instance of the object to the caller.
--
```

```
--
--          b. External Response:
--
--             i. none.
--
--       2. Receives an interrupt indicating that a summons button was
--          pressed from one of the floors.
--
--          a. Internal Response:
--
--             i. Reads the memory mapped eight-bit input register
--                (Up/Down_Input_Register) to determine from which
--                floor the summons button was pushed.
--
--             ii. Writes to the appropriate output register
--                 (Up/Down_Output_Register) to turn on the
--                 appropriate button light (Writing the floor number
--                 to this register toggles the light).
--
--          b. External Response:
--
--             i. Invokes the _Summons_ connector (event) procedure to
--                indicate that a summons has occurred. The parameters
--                needed to call Summons are Floor and Direction.
--
--
--       3. Receives a call indicating that the light under one of the
--          summons buttons should be extinguished.
--
--          a. Internal Response:
--
--              i. Writes to the appropriate output register
--                 (Up/Down_Output_Register) to turn off the
--                 appropriate button light (Writing the floor number
--                 to this register toggles the light).
--
--          b. External Response:
--
--             i. none
--
---------------------> END OBJECT REQUIREMENTS <----------------------
--
--
--
-------------------->  STATES MAINTAINED  <--------------------------
```

```
--
--        --All the fields in the Floor_Panel_Representation
--
--        --The state of the lights is recognized by the hardware. If
--          one of the buttons is illuminated the hardware will not
--          cause an interrupt if that button is pushed.
--
--
--
--------------------> OBJECT DEFINITION <---------------------------


  type Floor_Panel_Type is private;


--------------------> EXPORTED OPERATIONS <---------------------------


  function New_Floor_Panel
              (Up_Interrupt         : SET.Interrupt_Num_Type;
               Down_Interrupt       : SET.Interrupt_Num_Type)
                                               return Floor_Panel_Type;

  procedure Apply_Light_Out_To(This_Floor_Panel : in Floor_Panel_Type;
                               Floor            : in SET.Floor_Type;
                               Direction        : in SET.Direction_Type);


private

   Up_Interrupt_Address   : SET.Interrupt_Num_Type;
   Down_Interrupt_Address : SET.Interrupt_Num_Type;

   task type Floor_Panel_Representation is
      entry Initialize
                 (Up_Input_Register    : in SET.Register_Address_Type;
                  Down_Input_Register  : in SET.Register_Address_Type;
                  Up_Output_Register   : in SET.Register_Address_Type;
                  Down_Output_Register : in SET.Register_Address_Type);
      entry Up_Interrupt;
      entry Down_Interrupt;
      entry Light_Out(Floor     : in SET.Floor_Type;
                      Direction : in SET.Direction_Type);

      for Up_Interrupt use at Up_Interrupt_Address;
      for Down_Interrupt use at Down_Interrupt_Address;
    end Floor_Panel_Representation;

            --This entire private part definition can be moved to the
            --package body after implementation of the body is
            --complete.

   type Floor_Panel_Type is access Floor_Panel_Representation;
            --pointer to a Floor_Panel_Representation

end Floor_Panel_Manager;
```

A.3.2 Weight_Sensor_Manager.

```
with Standard_Engineering_Types;

--with: No packages "withed" from body

package Weight_Sensor_Manager is

    package SET renames Standard_Engineering_Types;

--------------------> OBJECT REQUIREMENTS <--------------------------

--        STIMULUS SUMMARY:
--        ----------------------------------------------------
--        1. Receives a call to create a new instance of the object.
--
--           Source of Stimulus: call to exported procedure
--                               "New_Weight_Sensor."
--
--        2. Receives a call to determine if the Elevator Weight is
--           below or equal to the maximum for this elevator.
--
--           Source of Stimulus: call to exported procedure
--                               "Return_Weight_OK_From."
--
--        RESPONSE SUMMARY:
--        ----------------------------------------------------
--        1. Receives a call to create a new instance of the object.
--
--           a. Internal Response:
--
--              i. Loads values from the Elevator_Data_Array to the
--                 corresponding fields of the
--                 Weight_Sensor_Representation. The
--                 Elevator_Data_Array is in the
--                 Standard_Engineering_Types package. The proper
--                 record in the array is found by indexing using the
--                 Elevator_ID passed in.
--
--              ii. Returns an instance of the object to the caller.
--
--
--           b. External Response:
--
--              i. none.
--
--
--        2. Receives a call to determine if the elevator weight is
--           exceeds the maximum for this elevator.
--
--           a. Internal Response:
--
--              i. Reads the memory mapped eight-bit input register
--                 (Weight_Sensor_Register) to determine the current
--                 weight of the elevator.
--
--              ii. Compares the current weight of the elevator to the
--                  maximum weight for this elevator
--                  (Weight_Capacity_Hundreds).
--
--              iii. Returns false in the elevator weight exceeds the
--                   maximum, returns true otherwise.
--
--           b. External Response:
--
--              i. none.
--
--
--
--------------------> END OBJECT REQUIREMENTS <----------------------


-------------------->  STATES MAINTAINED  <--------------------------

    --All the fields in the Weight_Sensor_Representation.
```

```
-------------------> OBJECT DEFINITION <--------------------------

  type Weight_Sensor_Type is private;

-------------------> EXPORTED OPERATIONS <-------------------------


  function New_Weight_Sensor
          (Elevator_ID : SET.Elevator_ID_Type) return Weight_Sensor_Type;


  function Return_Weight_OK_From
                (This_Weight_Sensor : Weight_Sensor_Type) return boolean;


private

   type Weight_Sensor_Representation is record
      Elevator_ID               : SET.Elevator_ID_Type;
      Weight_Sensor_Register    : SET.Register_Address_Type;
      Weight_Capacity_Hundreds  : SET.Weight_Type;
   end record;
           --The full definition may be moved to the package body
           --after implementation of the body is complete.

   type Weight_Sensor_Type is access Weight_Sensor_Representation;
          --pointer to a Weight_Sensor_Representation

end Weight_Sensor_Manager;
```

A.3.3 Scheduler_Manager.

```
with Standard_Engineering_Types;

--with Arrives, Proceed;

package Scheduler_Manager is

package SET renames Standard_Engineering_Types;
```

```
--------------------> OBJECT REQUIREMENTS <--------------------------

--       STIMULUS SUMMARY:
--       --------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          Source of Stimulus: call to exported procedure
--                              "New_Scheduler."
--
--       2. Receives a summons request indicating a would-be passenger
--          is waiting on one of the floors.
--
--          Source of stimulus : exported procedure (Apply_Summons_To)
--
--       3. Receives a destination indicating a passenger selected a
--          floor button from inside an elevator.
--
--          Source of stimulus : exported procedure
--                               (Apply_Destination_Request_To).
--
--
--       4. Receives an indication that an elevator is approaching a
--          floor.
--
--          Source of stimulus : exported procedure
--                               (Apply_Floor_Approaching).
--
--
--       RESPONSE SUMMARY:
--       ---------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          a. Internal Response:
--
--             i. Return an instance of the Scheduler to the caller
--
--
--          b. External Response:
--
--             i. none.
--
--
--       2. Receives a summons request indicated a would-be passenger
--          is waiting on one of the floors.
```

```
--
--          a. Internal Response:
--
--             i. Add the request to a list of outstanding requests
--                for that floor and direction.
--
--          b. External Response:
--
--             i. If there is an idle elevator, dispatch it to the
--                floor where the summons was issued by invoking the
--                connector "Proceed."
--
--       3. Receives a destination indicating a passenger selected a
--          floor button from inside an elevator.
--
--          a. Internal Response:
--
--             i. Add the request to the destination list for the
--                elevator.
--
--          b. External Response:
--
--             i. If the elevator is idle then dispatch it toward
--                the selected floor by invoking the connector
--                "Proceed."
--
--       4. Receives an indication that an elevator is approaching a
--          floor.
--
--          a. Internal Response:
--
--             i. Check to see if the elevator is scheduled to stop
--                at this floor for this direction (the Scheduler
--                knows which direction the elevator is traveling);
--
--            ii. After stopping, remove floor from the destination
--                list for this elevator.
--
--          b. External Response:
--
--             i. If the elevator is scheduled to stop at this floor
--                and direction then call connector "Arrives."
--
--            ii. After the elevator is stopped for three seconds
--                then have it proceed to the next destination by
--                    calling connector "Proceed."
--
--                    Note: Since the floor sensor does not signal
--                    stopped, we will have to estimate how long it
--                    take the elevator to stop and add that to the
--                    three seconds. This additional delay time could
--                    be added as a configuration item in the
--                    Standard_Engineering_Types package.
--
--
--------------------> END OBJECT REQUIREMENTS <----------------------


--------------------> STATES MAINTAINED <--------------------------

--    1. List of pending destinations for each elevator
--    2. List of pending Summons for each for each floor and direction
--    3. Current state of each elevator sufficient for efficient
--       scheduling. Sufficient information is available from the
--       knowledge of the last floor a elevator reported from, and the
--       direction it was dispatched. There is no need to query elevator
--       components about their state.

--------------------> OBJECT DEFINITION <--------------------------

type Scheduler_Type is private;

--------------------> EXPORTED OPERATIONS <--------------------------



  function New_Scheduler return Scheduler_Type;

  procedure Apply_Summons_To(This_Scheduler : in Scheduler_Type;
                             From_Floor     : in SET.Floor_Type;
                             Direction      : in SET.Direction_Type);

  procedure Apply_Destination_Request_To
                                (This_Scheduler : in Scheduler_Type;
                                 Elevator       : in SET.Elevator_ID_Type;
                                 Floor          : in SET.Floor_Type);

  procedure Apply_Floor_Approaching
                               (This_Scheduler : in Scheduler_Type;
                                Floor          : in SET.Floor_Type;
                                Elevator       : in SET.Elevator_ID_Type);

private

   type Scheduler_Representation;
          --incomplete type, defined in package body

   type Scheduler_Type is access Scheduler_Representation;
          --pointer to a Floor_Panel_Representation

end Scheduler_Manager;
```

A.3.4 Location_Panel_Manager.

```
with Standard_Engineering_Types;

--with: No packages "withed" from body

package Location_Panel_Manager is

    package SET renames Standard_Engineering_Types;

--------------------> OBJECT REQUIREMENTS <--------------------------

--        STIMULUS SUMMARY:
--        ---------------------------------------------------
--        1. Receives a call to create a new instance of the object.
--
--           Source of Stimulus: call to exported procedure
--                               "New_Location_Panel."
--
--
--        2. Receives a call to update the lights in the floor
--           indicator panel.
--
--           Source of Stimulus: call to exported procedure
--                               "Apply_Update_Location_Indicator."
--
--
--
--        RESPONSE SUMMARY:
```

```
--      ----------------------------------------------------
--      1. Receives a call to create a new instance of the object.
--
--         a. Internal Response:
--
--            i. Loads values from the Elevator_Data_Array to the
--               corresponding fields of the
--               Location_Panel_Representation. The
--               Elevator_Data_Array is in the
--               Standard_Engineering_Types package. The proper
--               record in the array is found by indexing using the
--               Elevator_ID passed in.
--
--            ii. Returns an instance of the object to the caller.
--
--
--         b. External Respor.se:
--
--            i. none.
--
--
--      2. Receives a call to update the lights in the floor
--         indicator panel.
--
--         a. Internal Response:
--
--             i. Writes the floor number of the indicator light
--                which is currently lit
--                (Current_Floor_Indicator_Lit) to the appropriate
--                output register (Location_Panel_Output_Register) to
--                turn off the light to the previous floor. (Writing
--                the floor number to this register toggles the
--                light).
--
--             ii. Writes the floor number of the indicator light
--                 which is to be lighted (New_Floor) to the
--                 appropriate output register
--                 (Location_Panel_Output_Register) to turn on the
--                 light. (Writing the floor number to this register
--                 toggles the light).
--
--             iii. Updates Current_Floor_Indicator_Lit to equal
--                  New_Floor.
--
--         b. External Response:
```

```
--
--             i. none
--
--
--------------------> END OBJECT REQUIREMENTS <----------------------



--------------------> STATES MAINTAINED <--------------------------

    --All the fields in the Location_Panel_Representation.



--------------------> OBJECT DEFINITION <--------------------------


  type Location_Panel_Type is private;


--------------------> EXPORTED OPERATIONS <--------------------------




  function New_Location_Panel(Elevator_ID : SET.Elevator_ID_Type)
                                              return Location_Panel_Type;

  procedure Apply_Update_Location_Indicator
                     (This_Location_Panel : in Location_Panel_Type;
                      New_Floor           : in SET.Floor_Type);


private

   type Location_Panel_Representation is record
     Elevator_ID_Type                : SET.Elevator_ID_Type;
     Current_Floor_Indicator_Lit     : SET.Floor_Type;
     Location_Panel_Output_Register : SET.Register_Address_Type;
   end record;
           --The full definition may be moved to the package body
           --after implementation of the body is complete.


   type Location_Panel_Type is access Location_Panel_Representation;
```

```
        --pointer to a Location_Panel_Representation

end Location_Panel_Manager;
```

A.3.5 Control_Panel_Manager.

```
with Standard_Engineering_Types;
with System;

--with Destination_Requested (connector); --needs to be "withed" from
                                         --the package body.

package Control_Panel_Manager is

    package SET renames Standard_Engineering_Types;

--------------------> OBJECT REQUIREMENTS <--------------------------

--       STIMULUS SUMMARY:
--       ---------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          Source of Stimulus: call to exported procedure
--                              "New_Control_Panel."
--
--
--       2. Receives an interrupt indicating that a floor has been
--          requested.
--
--          Source of Stimulus : interrupt from hardware
--                               (Control_Panel_Interrupt).
--
--       3. Receives a call indicating that the light under one of the
--          destination buttons should be extinguished.
--
--          Source of Stimulus : call to exported procedure
--                               (Apply_Light_Out_To).
--
--
--       RESPONSE SUMMARY:
--       ----------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          a. Internal Response:
--
```

```
--             i. Uses the interrupt value to create a new
--                instance of the task; it does this by assigning the
--                interrupt value to the Control_Panel_Interrupt_Num
--                variable before creating the new instance.
--
--             ii. Uses correspondingly named fields of the
--                 Elevator_Data_Array to initialize the task using
--                 the "initialize" entry. The Elevator_Data_Array is in
--                 the Standard_Engineering_Types package. The proper
--                 record in the array is found by indexing using the
--                 Elevator_ID passed in.
--
--             iii. Returns an instance of the object to the caller.
--
--
--          b. External Response:
--
--             i. none.
--
--       2. Receives an interrupt indicating that a floor has been
--          requested.
--
--          a. Internal Response:
--
--             i. Reads the memory mapped eight-bit input register
--                (Control_Panel_Input_Register) to determine which
--                floor was selected.
--
--             ii. Writes to the appropriate output register
--                 (Control_Panel_Output_Register) to turn on the
--                 appropriate button light (Writing the floor number
--                 to this register toggles the light).
--
--          b. External Response:
--
--             i. Invokes the _Destination_Requested_ connector
--                procedure to indicate that a destination request has
--                occurred. The parameters needed to call
--                _Destination_Requested_ are Elevator_ID and Floor.
--
--
--
--       3. Receives a call indicating that the light under one of the
--          destination buttons should be extinguished.
--
```

```
--          a. Internal Response:
--
--              i. Writes to the appropriate output register
--                 (Control_Panel_Output_Register) to turn off the
--                 appropriate button light (Writing the floor number
--                 to this register toggles the light).
--
--          b. External Response:
--
--             i. none
--
--
--
--------------------> END OBJECT REQUIREMENTS <----------------------



-------------------->  STATES MAINTAINED  <--------------------------

--  --All the fields in the Control_Panel_Representation
--
--  --The state of the lights is recognized by the hardware. If one of
--    the buttons is illuminated the hardware will not cause an
--    interrupt if that button is pushed.
--



--------------------> OBJECT DEFINITION <--------------------------


  type Control_Panel_Type is private;


--------------------> EXPORTED OPERATIONS <--------------------------



  function New_Control_Panel
            (Elevator_ID                  : SET.Elevator_ID_Type;
             Control_Panel_Interrupt      : SET.Interrupt_Num_Type)
                                             return Control_Panel_Type;


  procedure Apply_Light_Out_To
```

```
                           (This_Control_Panel : in Control_Pa[illegible]
                            Floor              : in SET.Flo[illegible]ype);



private

   Control_Panel_Interrupt_Num   : SET.Interrupt_Num_Type;

   task type Control_Panel_Representation is
      entry Initialize
          (Elevator_ID                   : in SET.Elevator_ID_Type;
           Control_Panel_Input_Register  : in SET.Register_Address_Type;
           Control_Panel_Output_Register : in SET.Register_Address_Type);
      entry Control_Panel_Interrupt;
      entry Light_Out(Floor : in SET.Floor_Type);
      for Control_Panel_Interrupt use at Control_Panel_Interrupt_Num;
   end Control_Panel_Representation;

           --The full definition may be moved to the package body
           --after implementation of the body is complete.


   type Control_Panel_Type is access Control_Panel_Representation;
          --pointer to a Control_Panel_Representation

end Control_Panel_Manager;
```

A.3.6 Floor_Sensor_Manager.

```
with Standard_Engineering_Types;
with System;

--with Floor_Approaching (Connector); --Move this to the package body

package Floor_Sensor_Manager is

    package SET renames Standard_Engineering_Types;

--------------------> OBJECT REQUIREMENTS <--------------------------

--        STIMULUS SUMMARY:
--        ---------------------------------- ---------------
--        1. Receives a call to create a new instance of the object.
--
```

```
--          Source of Stimulus: call to exported procedure
--                              "New_Floor_Sensor."
--
--       2. Receives interrupt indicating that the elevator is
--          approaching a floor.
--
--          Source of Stimulus: interrupt from hardware
--                              (Floor_Sensor_Interrupt).
--
--
--       RESPONSE SUMMARY:
--       ----------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          a. Internal Response:
--
--             i. Uses the interrupt value to create a new
--                instance of the task; it does this by assigning the
--                interrupt value to the Floor_Sensor_Interrupt_Num
--                variable before creating the new instance.
--
--             ii. Uses correspondingly named fields of the
--                 Elevator_Data_Array to initialize the task using
--                 the "initialize" entry. The Elevator_Data_Array is in
--                 the Standard_Engineering_Types package. The proper
--                 record in the array is found by indexing using the
--                 Elevator_ID passed in.
--
--
--             iii. Returns an instance of the object to the caller.
--
--
--          b. External Response:
--
--             i. none.
--
--
--       2. Receives interrupt indicating that the elevator is
--          approaching a floor.
--
--          a. Internal Response:
--
--             i. Reads the memory mapped eight-bit input register
--                (Floor_Sensor_Input_Register) to determine which
--                floor is being approached.
```

```
--
--          b. External Response:
--
--             i. Invokes the _Floor_Approaching_ connector procedure
--                to indicate that the elevator is approaching a
--                floor. Parameters needed to call _Floor_Approaching_
--                are Elevator_ID and Floor.
--
--
--------------------> END OBJECT REQUIREMENTS <---------------------
--
--
--
-------------------->  STATES MAINTAINED  <-------------------------

    --All the fields in the Floor_Sensor_Representation.



--------------------> OBJECT DEFINITION <--------------------------


  type Floor_Sensor_Type is private;


--------------------> EXPORTED OPERATIONS <-------------------------



  function New_Floor_Sensor
           (Elevator_ID                  : SET.Elevator_ID_Type;
            Floor_Sensor_Interrupt       : SET.Interrupt_Num_Type)
                                                 return Floor_Sensor_Type;


private

   Floor_Sensor_Interrupt_Num : SET.Interrupt_Num_Type;

   task type Floor_Sensor_Representation is
      entry Initialize
           (Elevator_ID                  : in SET.Elevator_ID_Type;
            Floor_Sensor_Input_Register : in SET.Register_Address_Type);
      entry Floor_Sensor_Interrupt;
      for Floor_Sensor_Interrupt use at Floor_Sensor_Interrupt_Num;
   end Floor_Sensor_Representation;

           --The full definition may be moved to the package body
           --after implementation of the body is complete.

   type Floor_Sensor_Type is access Floor_Sensor_Representation;
          --pointer to a Floor_Sensor_Representation

end Floor_Sensor_Manager;
```

A.3.7 Motor_Manager.

```
with Standard_Engineering_Types;
package Motor_Manager is

    package SET renames Standard_Engineering_Types;


-------------------> OBJECT REQUIREMENTS <-------------------------

--       STIMULUS SUMMARY:
--       ---------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          Source of Stimulus: call to exported procedure "New_Motor."
--
--
--       2. Receives a call to make the motor go. "Go" can be up or
--          down.
--
--          Source of stimulus: call to exported procedure.
--
--       3. Receives a call to make the motor stop.
--
--          Source of stimulus: call to exported procedure.
--
--
--       RESPONSE SUMMARY:
--       ----------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          a. Internal Response:
--
--             i. Loads values from the Elevator_Data_Array to the
--                    corresponding fields of the Motor_Representation.
--                    The Elevator_Data_Array is in the
--                    Standard_Engineering_Types package. The proper
--                    record in the array is found by indexing using the
--                    Elevator_ID passed in.
--
--                 ii. Returns an instance of the object to the caller.
--
--
--              b. External Response:
--
--                 i. none.
--
--
--           2. Receives a call to make the motor go. "Go" can be up or
--              down.
--
--              a. Internal Response:
--
--                 i. Write appropriate output command
--                    (Motor_Command_Up/Down) to the motor control
--                    register for this motor (Motor_Control_Register).
--
--              b. External Response:
--
--                  i. none.
--
--
--           3. Receives a call to make the motor stop.
--
--              a. Internal Response:
--
--                 i. Write appropriate output command
--                    (Motor_Command_Stop) to the motor control register
--                    for this motor (Motor_Control_Register).
--
--              b. External Response:
--
--                 i. none.
--
--
--------------------> END OBJECT REQUIREMENTS <---------------------
```

```
-------------------->  STATES MAINTAINED  <--------------------------

    --All the fields in the Motor_Representation.



--------------------> OBJECT DEFINITION <--------------------------


  type Motor_Type is private;


--------------------> EXPORTED OPERATIONS <--------------------------


  function New_Motor(Elevator_ID : in SET.Elevator_ID_Type)
                                                     return Motor_Type;

  procedure Apply_Go_To(This_Motor : in Motor_Type;
                        Direction  : in SET.Direction_Type);

  procedure Apply_Stop_To(This_Motor : in Motor_Type);



private

   type Motor_Representation is record
      Elevator_ID            : SET.Elevator_ID_Type;
      Motor_Control_Register : SET.Register_Address_Type;
      Motor_Command_Up       : SET.Command_Byte_Type;
      Motor_Command_Down     : SET.Command_Byte_Type;
      Motor_Command_Stop     : SET.Command_Byte_Type;
   end record;
           --The full definition may be moved to the package body
           --after implementation of the body is complete.


   type Motor_Type is access Motor_Representation;
          --pointer to a Floor_Panel_Representation

end Motor_Manager;
```

A.4 System_Aggregate Packages

A.4.1 Elevator_System_Aggregate.

```
with Standard_Engineering_types;

with Weight_Sensor_Manager, Location_Panel_Manager, Control_Panel_Manager,
     Floor_Sensor_Manager, Motor_Manager;

pragma elaborate (Weight_Sensor_Manager, Location_Panel_Manager,
                  Control_Panel_Manager, Floor_Sensor_Manager,
                  Motor_Manager);

package Elevator_System_Aggregate is

   package SET renames Standard_Engineering_types;

   type Elevator_Representation is record
      The_Weight_Sensor   : Weight_Sensor_Manager.Weight_Sensor_Type;
      The_Location_Panel  : Location_Panel_Manager.Location_Panel_Type;
      The_Control_Panel   : Control_Panel_Manager.Control_Panel_Type;
      The_Floor_Sensor    : Floor_Sensor_Manager.Floor_Sensor_Type;
      The_Motor           : Motor_Manager.Motor_Type;
    end record;


    type Elevator_Type is array(SET.Elevator_ID_Type)
                                             of Elevator_Representation;

    Elevators : Elevator_Type;

end Elevator_System_Aggregate;
------------------
package body Elevator_System_Aggregate is

begin
 for Current_Elevator in SET.Elevator_ID_Type loop

     Elevators(Current_Elevator).The_Weight_Sensor :=
             Weight_Sensor_Manager.New_Weight_Sensor(Current_Elevator);

     Elevators(Current_Elevator).The_Location_Panel :=
           Location_Panel_Manager.New_Location_Panel(Current_Elevator);

     Elevators(Current_Elevator).The_Control_Panel :=
          Control_Panel_Manager.New_Control_Panel
            (Elevator_ID           => Current_Elevator,
             Control_Panel_Interrupt => SET.Elevator_Data_Array
                      (Current_Elevator).Control_Panel_Interrupt);

     Elevators(Current_Elevator).The_Floor_Sensor :=
               Floor_Sensor_Manager.New_Floor_Sensor
                  (Elevator_ID             => Current_Elevator,
                   Floor_Sensor_Interrupt => SET.Elevator_Data_Array
                         (Current_Elevator).Floor_Sensor_Interrupt);

     Elevators(Current_Elevator).The_Motor :=
                        Motor_Manager.New_Motor(Current_Elevator);

end loop;
end Elevator_System_Aggregate;
```

A.4.2 Scheduler_System_Aggregate.

```
with Scheduler_Manager;
pragma elaborate (Scheduler_Manager);

package Scheduler_System_Aggregate is

   Scheduler : constant Scheduler_Manager.Scheduler_Type :=
                              Scheduler_Manager.New_Scheduler;

end Scheduler_System_Aggregate;
```

A.4.3 Floor_Panel_Aggregate.

```
with Floor_Panel_Manager;
with Standard_Engineering_Types;
pragma elaborate (Floor_Panel_Manager);
package Floor_Panel_System_Aggregate is

   package SET renames Standard_Engineering_types;

   Floor_Panels : constant Floor_Panel_Manager.Floor_Panel_Type :=
      Floor_Panel_Manager.New_Floor_Panel
         (Up_Interrupt         => SET.Floor_Panel_Data.Up_Interrupt,
          Down_Interrupt       => SET.Floor_Panel_Data.Down_Interrupt);

end Floor_Panel_System_Aggregate;
```

A.5 Connector/Event Procedures

A.5.1 Summons.

```
with Standard_Engineering_Types;
with Scheduler_System_Aggregate;
with Scheduler_Manager;
use Standard_Engineering_Types;

procedure Summons(From_Floor         : in Floor_Type;
                  Desired_Direction  : in Direction_Type) is

   package SSA renames Scheduler_System_Aggregate;
   package SM  renames Scheduler_Manager;


    begin
      SM.Apply_Summons_To(This_Scheduler => SSA.Scheduler,
                          From_Floor     => From_Floor,
                          Direction      => Desired_Direction);

end Summons;
```

A.5.2 Arrives.

```
with Standard_Engineering_Types;

--System Level packages:
with Elevator_System_Aggregate;
with Floor_Panel_System_Aggregate;

--Object Level Packages:
with Control_Panel_Manager;
with Motor_Manager;
with Floor_Panel_Manager;

use Standard_Engineering_Types;

procedure Arrives(This_Elevator : in Elevator_ID_Type;
                  Floor         : in Floor_Type;
                  Direction     : in Direction_Type) is

   package ESA  renames Elevator_System_Aggregate;
   package FPSA renames Floor_Panel_System_Aggregate;
   package CPM  renames Control_Panel_Manager;
   package MM   renames Motor_Manager;
   package FPM  renames Floor_Panel_Manager;

begin


   MM.Apply_Stop_To(This_Motor => ESA.Elevators(This_Elevator).The_Motor);

   CPM.Apply_Light_Out_To
            (This_Control_Panel =>
                 ESA.Elevators(This_Elevator).The_Control_Panel,
             Floor              => Floor);

   FPM.Apply_Light_Out_To (This_Floor_Panel => FPSA.Floor_Panels,
                           Floor            => Floor,
                           Direction        => Direction);


end Arrives;
```

A.5.3 Proceed.

```
with Standard_Engineering_Types;

--System Level packages:
with Elevator_System_Aggregate;

--Object Level Packages:
with Motor_Manager;
with Weight_Sensor_Manager;

use Standard_Engineering_Types;

procedure Proceed (This_Elevator : in Elevator_ID_Type;
                   Direction     : in Direction_Type) is

   package ESA renames Elevator_System_Aggregate;
   package WSM renames Weight_Sensor_Manager;
   package MM  renames Motor_Manager;

begin

  loop
    if WSM.Return_Weight_OK_From
            (ESA.Elevators(This_Elevator).The_Weight_Sensor) then

       MM.Apply_Go_To
          (This_Motor => ESA.Elevators(This_Elevator).The_Motor,
           Direction  => Direction);

       exit;

    else
      delay 5.0;
    end if;

  end loop;


end Proceed;
```

A.5.4 Destination_Requested.

```
with Standard_Engineering_Types;

--system level packages:
with Scheduler_System_Aggregate;

--object level packages:
with Scheduler_Manager;

use Standard_Engineering_Types;

procedure Destination_Requested(Elevator_ID : in Elevator_ID_Type;
                                To_Floor    : in Floor_Type) is

   package SSA renames Scheduler_System_Aggregate;
   package SM  renames Scheduler_Manager;


begin
  SM.Apply_Destination_Request_To
          (This_Scheduler => SSA.Scheduler,
             Elevator       => Elevator_ID,
             Floor          => To_Floor);


end Destination_Requested;
```

A.5.5 Floor_Approaching.

```
with Standard_Engineering_Types;
with Elevator_System_Aggregate;
with Scheduler_System_Aggregate;
with Location_Panel_Manager;
with Scheduler_Manager;
use Standard_Engineering_Types;

procedure Floor_Approaching(This_Elevator : in Elevator_ID_Type;
                            Floor         : in Floor_Type) is

   package ESA renames Elevator_System_Aggregate;
   package SSA renames Scheduler_System_Aggregate;
   package LPM renames Location_Panel_Manager;
   package SM  renames Scheduler_Manager;


  begin

    LPM.Apply_Update_Location_Indicator
            (This_Location_Panel =>
                    ESA.Elevators(This_Elevator).The_Location_Panel,
             New_Floor           => Floor);


    SM.Apply_Floor_Approaching
              (This_Scheduler => SSA.Scheduler,
               Floor          => Floor,
               Elevator       => This_Elevator);

end Floor_Approaching;
```

Appendix B. *Ada Specifications for the Cruise Control Problem*

B.1 Standard_Engineering_Types

```
with System;
package Standard_Engineering_Types is
          --(for the Cruise Control System)

--This package serves two purposes:
-- 1. It contains type definitions for parameters that are used in the
--    Cruise Control control system. These parameters were mapped from
--    the Object-Mapping Table which mapped all the objects defined in
--    the analysis. Areas marked "PARAMETER AREA #X:" contain
--    parameters that were mapped from the analysis
--
-- 2. It configures the design for reusability. The areas marked:
--    "CONFIGURATION AREA #X:" contain data used to instantiate the
--    design. The parameters that can be instantiated are:
--         -The maximum speed expected of the vehicle.
--         -The maximum speed at which the Cruise Control can be used.
--         -The time interval at which the speed is checked and
--          updated.
--         -All the hardware interface values, including:
--            --Six interrupt vectors.
--            --Two I/O register addresses.
--            --23 throttle control command values.
--
-- This means that the Cruise Control control software system can be
-- configured for different Cruise Control hardware systems by
-- instantiating these values.
--
--
--
--
--     --CONFIGURATION AREA #1:
--
Max_Vehicle_Speed : constant := 150; --Not included in analysis.
Max_Cruise_Speed  : constant := 100; --To satisfy
                                     --metarequirement of analysis.
Update_Interval   : constant duration := 1.0;
                                     --(seconds), this interval was not
                                     --included in the analysis.
```

```
    --PARAMETER AREA #1:

subtype Speed_Type is integer range 0..Max_Vehicle_Speed;


subtype Interrupt_Num_Type    is System.Address;
subtype Register_Address_Type is System.Address;
subtype Command_Byte_Type     is System.Address;

type Cruise_Control_Data_Type is record
--Interrupt Numbers:
    Set_Button_Interrupt            : Interrupt_Num_Type;
    On_Button_Interrupt             : Interrupt_Num_Type;
    Off_Button_Interrupt            : Interrupt_Num_Type;
    Resume_Button_Interrupt         : Interrupt_Num_Type;
    Accelerate_Button_Interrupt     : Interrupt_Num_Type;
    Brake_Pedal_Interrupt           : Interrupt_Num_Type;

--Input/Output Registers:
    Speed_Sensor_Input_Register     : Register_Address_Type;
    Throttle_Control_Output_Register : Register_Address_Type;

--Throttle_Control Commands:
    Set_Throttle                    : Command_Byte_Type;
    Release_Throttle                : Command_Byte_Type;
    No_Change                       : Command_Byte_Type;
    Decelerate_Increment_1          : Command_Byte_Type;
    Decelerate_Increment_2          : Command_Byte_Type;
    Decelerate_Increment_3          : Command_Byte_Type;
    Decelerate_Increment_4          : Command_Byte_Type;
    Decelerate_Increment_5          : Command_Byte_Type;
    Decelerate_Increment_6          : Command_Byte_Type;
    Decelerate_Increment_7          : Command_Byte_Type;
    Decelerate_Increment_8          : Command_Byte_Type;
    Decelerate_Increment_9          : Command_Byte_Type;
    Decelerate_Increment_10         : Command_Byte_Type;
    Accelerate_Increment_1          : Command_Byte_Type;
    Accelerate_Increment_2          : Command_Byte_Type;
    Accelerate_Increment_3          : Command_Byte_Type;
    Accelerate_Increment_4          : Command_Byte_Type;
    Accelerate_Increment_5          : Command_Byte_Type;
    Accelerate_Increment_6          : Command_Byte_Type;
    Accelerate_Increment_7          : Command_Byte_Type;
    Accelerate_Increment_8          : Command_Byte_Type;
    Accelerate_Increment_9          : Command_Byte_Type;
    Accelerate_Increment_10          : Command_Byte_Type;
end record;


  Cruise_Control_Data : Cruise_Control_Data_Type;



end Standard_Engineering_Types;
--=====================================================================
package body Standard_Engineering_Types is


begin
--Configure the Cruise Control implementation by plugging in the
  interrupt numbers, register addresses, and hardware commands:

       --CONFIGURATION AREA #2:

  ------------------------------------------------
--Interrupt Numbers:
    Cruise_Control_Data.Set_Button_Interrupt            := 16#B1#;
    Cruise_Control_Data.On_Button_Interrupt             := 16#B2#;
    Cruise_Control_Data.Off_Button_Interrupt            := 16#B3#;
    Cruise_Control_Data.Resume_Button_Interrupt         := 16#B4#;
    Cruise_Control_Data.Accelerate_Button_Interrupt     := 16#B5#;
    Cruise_Control_Data.Brake_Pedal_Interrupt           := 16#B6#;

--Input/Output Registers:
    Cruise_Control_Data.Speed_Sensor_Input_Register     := 16#A1#;
    Cruise_Control_Data.Throttle_Control_Output_Register := 16#A2#;

--Throttle_Control Commands:
    Cruise_Control_Data.Set_Throttle                    := 16#10#;
    Cruise_Control_Data.Release_Throttle                := 16#AA#;
                                               --This one only needs
                                               --to be different from
                                               --the others.

    Cruise_Control_Data.No_Change                       := 16#FF#;
    Cruise_Control_Data.Decelerate_Increment_1          := 16#01#;
    Cruise_Control_Data.Decelerate_Increment_2          := 16#02#;
    Cruise_Control_Data.Decelerate_Increment_3          := 16#03#;
    Cruise_Control_Data.Decelerate_Increment_4          := 16#04#;
    Cruise_Control_Data.Decelerate_Increment_5          := 16#05#;
    Cruise_Control_Data.Decelerate_Increment_6          := 16#06#;
    Cruise_Control_Data.Decelerate_Increment_7          := 16#07#;
    Cruise_Control_Data.Decelerate_Increment_8          := 16#08#;
    Cruise_Control_Data.Decelerate_Increment_9          := 16#09#;
    Cruise_Control_Data.Decelerate_Increment_10         := 16#0A#;
    Cruise_Control_Data.Accelerate_Increment_1          := 16#81#;
    Cruise_Control_Data.Accelerate_Increment_2          := 16#82#;
    Cruise_Control_Data.Accelerate_Increment_3          := 16#83#;
    Cruise_Control_Data.Accelerate_Increment_4          := 16#84#;
    Cruise_Control_Data.Accelerate_Increment_5          := 16#85#;
    Cruise_Control_Data.Accelerate_Increment_6          := 16#86#;
    Cruise_Control_Data.Accelerate_Increment_7          := 16#87#;
    Cruise_Control_Data.Accelerate_Increment_8          := 16#88#;
    Cruise_Control_Data.Accelerate_Increment_9          := 16#89#;
    Cruise_Control_Data.Accelerate_Increment_10         := 16#8A#;
   --------------------------------------------------

end Standard_Engineering_Types;
```

B.2 Cruise Control Object_Managers

B.2.1 Throttle_Control_Manager.

```
with Standard_Engineering_Types;

--with: No known withing is needed from the package body.

package Throttle_Control_Manager is

    package SET renames Standard_Engineering_Types;

--------------------> OBJECT REQUIREMENTS <--------------------------

--       STIMULUS SUMMARY:
--       ---------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          Source of Stimulus: call to exported procedure
--                              "New_Throttle_Control."
--
--       2. Receives a call to set the throttle at the current position.
--
--          Source of stimulus: call to exported procedure:
--                              "Apply_Set_To."
--
```

```
--       3. Receives a call to change the throttle setting.
--
--          Source of stimulus: call to exported procedure:
--                              "Apply_Change_Setting_To."
--
--       4. Receives a call to release the throttle.
--
--          Source of stimulus: call to exported procedure:
--                              "Apply_Release_To."
--
--       RESPONSE SUMMARY:
--       ----------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--          a. Internal Response:
--
--             i. Loads values from the Cruise_Cont:ol_Data variable
--                in Standard_Engineering_Types to the corresponding
--                fields of the Cruise_Control_Representation.
--
--             ii. Returns an instance of the object to the caller.
--
--          b. External Response:
--             i. none.
--
--       2. Receives a call to set the throttle at the current
--          position.
--
--          a. Internal Response:
--
--             i. Write Set output command to the
--                Throttle_Control_Output_Register.
--
--             ii. Delay for a short period (at least as long as the read
--                 cycle of the hardware) and write the No_Change Command
--                 into the Throttle_Control_Output_Register.
--
--          b. External Response:
--              i. none.
--
--
--       3. Receives a call to change the throttle setting.
--          a. Internal Rosponse:
--
--             i. Write the Decelerate/Accelerate command, associated with
--                incoming Change and Change_Amount parameters, to the
--                Throttle_Control_Output_Register.
--
--             ii. Delay for a short period (at least as long as the read
--                 cycle of the hardware) and write the No_Change Command
--                 into the Throttle_Control_Output_Register.
--
--          b. External Response:
--              i. none.
--
--       4. Receives a call to release the throttle.
--          a. Internal Response:
--
--             i. Write the the Release command to the
--                Throttle_Control_Output_Register.
--
--          b. Ex+ernal Response:
--              i. none.
--
--
--------------------> END OBJECT REQUIREMENTS <----------------------



-------------------->  STATES MAINTAINED  <--------------------------

    --All the fields in the Throttle_Control_Representation.
    --The state Holding/Released state of the hardware may be
    --maintained if helpful.


--------------------> OBJECT DEFINITION <---------------------------


  type Throttle_Control_Type is private;

--------------------> PARAMETERS/ATTRIBUTES <--------------------------

  type Change_Type is (Deceleration, Acceleration);
  subtype Change_Amount_Type is integer range 1..10;

--------------------> EXPORTED OPERATIONS <--------------------------



  function New_Throttle_Control return Throttle_Control_Type;
  procedure Apply_Set_To
              (This_Throttle_Control : in Throttle_Control_Type);

  procedure Apply_Change_Setting_To
              (This_Throttle_Control : in Throttle_Control_Type;
               Change                : in Change_Type;
               Change_Amount         : in Change_Amount_Type);

  procedure Apply_Release_To
                 (This_Throttle_Control: in Throttle_Control_Type);

private

  type Throttle_Control_Representation is record
    Throttle_Control_Output_Register : SET.Register_Address_Type;
    Set_Throttle                     : SET.Command_Byte_Type;
    Release_Throttle                 : SET.Command_Byte_Type;
    No_Change                        : SET.Command_Byte_Type;
    Decelerate_Increment_1           : SET.Command_Byte_Type;
    Decelerate_Increment_2           : SET.Command_Byte_Type;
    Decelerate_Increment_3           : SET.Command_Byte_Type;
    Decelerate_Increment_4           : SET.Command_Byte_Type;
    Decelerate_Increment_5           : SET.Command_Byte_Type;
    Decelerate_Increment_6           : SET.Command_Byte_Type;
    Decelerate_Increment_7           : SET.Command_Byte_Type;
    Decelerate_Increment_8           : SET.Command_Byte_Type;
    Decelerate_Increment_9           : SET.Command_Byte_Type;
    Decelerate_Increment_10          : SET.Command_Byte_Type;
    Accelerate_Increment_1           : SET.Command_Byte_Type;
    Accelerate_Increment_2           : SET.Command_Byte_Type;
    Accelerate_Increment_3           : SET.Command_Byte_Type;
    Accelerate_Increment_4           : SET.Command_Byte_Type;
    Accelerate_Increment_5           : SET.Command_Byte_Type;
    Accelerate_Increment_6           : SET.Command_Byte_Type;
    Accelerate_Increment_7           : SET.Command_Byte_Type;
    Accelerate_Increment_8           : SET.Command_Byte_Type;
    Accelerate_Increment_9           : SET.Command_Byte_Type;
    Accelerate_Increment_10          : SET.Command_Byte_Type;
  end record;

          --The full definition may be moved to the package body
          --after implementation of the body is complete.

   type Throttle_Control_Type is access Throttle_Control_Representation;
```

```
          --pointer to a Throttle_Control_Representation

end Throttle_Control_Manager;
```

B.2.2 Speed_Sensor_Manager.

```
with Standard_Engineering_Types;

--with: No known withing is needed from the package body.


package Speed_Sensor_Manager is

    package SET renames Standard_Engineering_Types;

-------------------> OBJECT REQUIREMENTS <-------------------------
--
--       STIMULUS SUMMARY:
--       ---------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          Source of Stimulus: call to exported procedure
--                              "New_Speed_Sensor."
--
--       2. Receives a call to return the current speed.
--
--          Source of stimulus: call to exported procedure:
--                              "Return_Speed_From."
--
--
--       RESPONSE SUMMARY:
--       ---------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--          a. Internal Response:
--
--             i. Loads values from the Cruise_Control_Data variable in
--                Standard_Engineering_Types to the corresponding fields
--                of the Cruise_Control_Repiesentation.
--
--             ii. Returns an instance of the object to the caller.
--
--          b. External Response:
--             i. none.
--
--       2. Receives a call to return the current speed.
--          a. Internal Response:
```

```
--
--             i. Read the Speed_Sensor_Input_Register to determine the
--                speed.
--
--             ii. Convert from the hexadecimal speed integer to
--                 Speed_Type and return the value to the caller.
--
--
--          b. External Response:
--              i. none.
--
--
-------------------> END OBJECT REQUIREMENTS <---------------------


-------------------->  STATES MAINTAINED  <-------------------------

    --All the fields in the Speed_Sensor_Representation.


-------------------> OBJECT DEFINITION <---------------------------


  type Speed_Sensor_Type is private;


-------------------> EXPORTED OPERATIONS <-------------------------


  ıunction New_Speed_Sensor return Speed_Sensor_Type;


  function Return_Speed_From
        (This_Speed_Sensor : in Speed_Sensor_Type) return SET.Speed_Type;

private


   type Speed_Sensor_Representation is record
     Speed_Sensor_Input_Register : SET.Register_Address_Type;
   end record;
           --The full definition may be moved to the package body
           --after implementation of the body is complete.
   type Speed_Sensor_Type is access Speed_Sensor_Representation;
          --pointer to a Speed_Sensor_Representation

end Speed_Sensor_Manager;
```

B.2.3 Buttons_Manager.

```
with Standard_Engineering_Types;
with System; -- since this object accepts interrupts

--with : Turn_On, Set_Speed, Accelerate, Turn_Off, Resume, Brake;
--       This object is an event initiator, it "must" with all these
--       connectors to pass messages in responses to the events. Move
--       these "with"s to the package body.

package Buttons_Manager is

    package SET renames Standard_Engineering_Types;

-------------------> OBJECT REQUIREMENTS <------------------------
--
--       STIMULUS SUMMARY:
--       --------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          Source of Stimulus: call to exported procedure
--                                "New_Buttons."
--
--       2. Receives an interrupt indicating that the On Button was
--          pressed.
--
--          Source of Stimulus : On_Button_Interrupt.
--
--       3. Receives an interrupt indicating that the Off Button was
--          pressed.
--
--          Source of Stimulus : Off_Button_Interrupt.
--
--       4. Receives an interrupt indicating that the Set Button was
--          pressed.
--
--          Source of Stimulus : Set_Button_Interrupt.
--
--       5. Receives an interrupt indicating that the Resume Button was
--          pressed.
--
```

```
--          Source of Stimulus : Resume_Button_Interrupt.
--
--       6. Receives an interrupt indicating that the Accelerate Button
--          was pressed.
--
--          Source of Stimulus : Accelerate_Button_Interrupt.
--
--       7. Receives an interrupt indicating that the Brake Pedal was
--          pressed.
--
--          Source of Stimulus : Brake_Pedal_Interrupt.
--
--
--
--       RESPONSE SUMMARY:
--       ----------------------------------------------------
--       --NOTE ON INTERNAL DESIGN:
--          Additional tasks should be created to handle responses that
--          involve waiting for what may be a significant amount of time
--          for a connector to return. The reason is so the
--          Buttons_Representation task is available to receive important
--          interrupts like Brake_Pedal and Turn_Off.
--
--       1. Receives a call to create a new instance of the object.
--
--          a. Internal Response:
--
--             i. Use the six interrupt values in creating a new
--                instance of the task; it does this by assigning the
--                interrupt values to the Up/Down_Interrupt_Address
--                variables before creating the new instance.
--
--             ii. Returns an instance of the object to the caller.
--
--          b. External Response:
--
--             i. none.
--
--       2. Receives an interrupt indicating that the On Button was
--          pressed.
--
--          a. Internal Response:
--             i. None.
--
--          b. External Response:
```

```
--
--            i. Invoke the "Turn_On" Connector.
--
--
--       3. Receives an interrupt indicating that the Off Button was
--          pressed.
--
--          a. Internal Response:
--             i. None.
--
--          b. External Response:
--
--             i. Invoke the "Turn_Off" Connector.
--
--
--
--       4. Receives an interrupt indicating that the Set Button was
--          pressed.
--          a. Internal Response:
--             i. None.
--
--          b. External Response:
--
--             i. Invoke the "Set_Speed" Connector.
--
--       5. Receives an interrupt indicating that the Resume Button was
--          pressed.
--          a. Internal Response:
--             i. None.
--
--          b. External Response:
--
--             i. Invoke the "Resume" Connector.
--
--       6. Receives an interrupt indicating that the Accelerate Button
--          was pressed.
--          a. Internal Response:
--             i. None.
--
--          b. External Response:
--
--             i. Invoke the "Accelerate" Connector.
--
--       7. Receives an interrupt indicating that the Brake Pedal was
--          pressed.
--          a. Internal Response:
--             i. None.
--
--          b. External Response:
--
--             i. Invoke the "Brake" Connector.
--
--
--------------------> END OBJECT REQUIREMENTS <---------------------



--------------------> OBJECT DEFINITION <--------------------------


  type Buttons_Type is private;


--------------------> EXPORTED OPERATIONS <-----------------------



  function New_Buttons
             (On_Button_Interrupt         : SET.Interrupt_Num_Type;
              Off_Button_Interrupt        : SET.Interrupt_Num_Type;
              Set_Button_Interrupt        : SET.Interrupt_Num_Type;
              Resume_Button_Interrupt     : SET.Interrupt_Num_Type;
              Accelerate_Button_Interrupt : SET.Interrupt_Num_Type;
              Brake_Pedal_Interrupt       : SET.Interrupt_Num_Type)
                                                  return Buttons_Type;



private

     On_Button_Interrupt_Num         : SET.Interrupt_Num_Type;
     Off_Button_Interrupt_Num        : SET.Interrupt_Num_Type;
     Set_Button_Interrupt_Num        : SET.Interrupt_Num_Type;
     Resume_Button_Interrupt_Num     : SET.Interrupt_Num_Type;
     Accelerate_Button_Interrupt_Num : SET.Interrupt_Num_Type;
     Brake_Pedal_Interrupt_Num       : SET.Interrupt_Num_Type;

   task type Buttons_Representation is
      entry Initialize;
```

```
        entry On_Interrupt;
        entry Off_Interrupt;
        entry Set_Interrupt;
        entry Resume_Interrupt;
        entry Accelerate_Interrupt;
        entry Brake_Interrupt;
        for On_Interrupt use at On_Button_Interrupt_Num;
        for Off_Interrupt use at Off_Button_Interrupt_Num;
        for Set_Interrupt use at Set_Button_Interrupt_Num;
        for Resume_Interrupt use at Resume_Button_Interrupt_Num;
        for Accelerate_Interrupt use at Accelerate_Button_Interrupt_Num;
        for Brake_Interrupt use at Brake_Pedal_Interrupt_Num;
      end Buttons_Representation;

           --The full definition may be moved to the package body
           --after implementation of the body is complete.

     type Buttons_Type is access Buttons_Representation;
          --pointer to a Buttons_Representation

end Buttons_Manager;
```

B.2.4 System_States_Manager.

```
with Standard_Engineering_Types;

--with: Nothing is known to need withing from the package body.

package System_States_Manager is

    package SET renames Standard_Engineering_Types;

--------------------> OBJECT REQUIREMENTS <--------------------------
--
--       STIMULUS SUMMARY:
--       ---------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          Source of Stimulus: function New_System_States.
--
--       2. Receives a call to save the On state as "on."
--
--          Source of stimulus: procedure Apply_Turn_On_To.
--
--       3. Receives a call to save the engaged state as "engaged."
--
```

```
--          Source of stimulus: procedure Apply_Engage_To.
--
--       4. Receives a call to Save the Desired Speed.
--
--          Source of stimulus: procedure Apply_Desired_Speed_To.
--
--       5. Receives a call to save the engaged state as "disengaged."
--
--          Source of stimulus: procedure Apply_Disengage_To.
--
--       6. Receives a call to save the On state as "off."
--
--          Source of stimulus: procedure Apply_Turn_Off_To.
--
--       7. Receives a call to return the current On state.
--
--          Source of stimulus: function Return_On_State_From.
--
--       8. Receives a call to return the current Engaged state.
--
--          Source of stimulus: function Return_Engaged_State_From.
--
--       9. Receives a call to return the current Desired Speed.
--
--          Source of stimulus: function Return_Desired_Speed_From.
--
--
--       RESPONSE SUMMARY:
--       --------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--          a. Internal Response:
--             i. Returns an instance of the object.
--
--          b. External Response:
--             i. none.
--
--
--       2. Receives a call to save the On state as "on."
--          a. Internal Response:
--             i. Set On_State to true.
--
--          b. External Response:
--             i. none.
--
--       3. Receives a call to save the engaged state as "engaged."
--          a. Internal Response:
--             i. Set Engaged_State to true.
--
--          b. External Response:
--             i. none.
--
--
--       4. Receives a call to Save the Desired Speed.
--          a. Internal Response:
--             i. Assign Desired_Speed to Speed_State.
--
--          b. External Response:
--             i. none.
--
--
--       5. Receives a call to save the engaged state as "disengaged."
--          a. Internal Response:
--             i. Set Engaged_State to false.
--
--          b. External Response:
--             i. none.
--
--       6. Receives a call to save the On state as "off."
--          a. Internal Response:
--             i. Set On_State to false.
--
--          b. External Response:
--             i. none.
--
--       7. Receives a call to return the current On state.
--          a. Internal Response:
--             i. Return On_State.
--
--          b. External Response:
--             i. none.
--
--       8. Receives a call to return the current Engaged state.
--          a. Internal Response:
--             i. Return Engaged_State.
--
--          b. External Response:
--             i. none.
--
--       9. Receives a call to return the current Desired Speed.
--          a. Internal Response:
```

```
--             i. Return Speed_State.
--
--          b. External Response:
--             i. none.
--
--------------------> END OBJECT REQUIREMENTS <----------------------



--------------------> STATES MAINTAINED  <--------------------------

    --All the fields in the System_States_Representation.


--------------------> OBJECT DEFINITION <---------------------------


  type System_States_Type is private;


--------------------> EXPORTED OPERATIONS <-------------------------



  function New_System_States return System_States_Type;

  procedure Apply_Turn_On_To(This_System_States : in System_States_Type);
  procedure Apply_Engage_To (This_System_States : in System_States_Type);
  procedure Apply_Desired_Speed_To
              (This_System_States : in System_States_Type;
               Desired_Speed      : in SET.Speed_Type);

  procedure Apply_Disengage_To
                             (This_System_States : in System_States_Type);

  procedure Apply_Turn_Off_To(This_System_States : in System_States_Type);


  function Return_On_State_From
              (This_System_States : in System_States_Type) return boolean;

  function Return_Engaged_State_From
              (This_System_States : in System_States_Type) return boolean;

  function Return_Desired_Speed_From
```

```
                    (This_System_States : in System_States_Type)
                                                return SET.Speed_Type;


private
--if no interrupts:
     type System_States_Representation is record
       On_State      : boolean := false;
       Engaged_State : boolean := false;
       Speed_State   : SET.Speed_Type;
     end record;

           --The full definition may be moved to the package body
           --after implementation of the body is complete.

   type System_States_Type is access System_States_Representation;
          --pointer to a System_States_Representation

end System_States_Manager;
```

B.2.5 Timer_Manager.

```
with Standard_Engineering_Types;

--with: Update; --connector, move this to package body

package Timer_Manager is

    package SET renames Standard_Engineering_Types;

--------------------> OBJECT REQUIREMENTS <-----------------------------
--
--       STIMULUS SUMMARY:
--       ---------------------------------------------------
--       1. Receives a call to create a new instance of the object.
--
--          Source of Stimulus: function New_Timer.
--
--       2. Time to Update.
--
--          Source of Stimulus: The elapse of the time interval
--                              defined by Update_Interval in
--                              Standard_Engineering_Types.
--
--
--       RESPONSE SUMMARY:
```

```
--      ---------------------------------------------------
--      1. Receives a call to create a new instance of the object.
--         a. Internal Response:
--            i. Initialize the task using the Time_Interval
--               parameter passed in with the Initialize entry.
--
--            ii. Returns an instance of the object.
--
--      2. Time to Update.
--         a. Internal Response:
--            i. Reset the interval timer.
--
--         b. External Response:
--            i. Invoke the "Update" Connector.
--
--------------------> END OBJECT REQUIREMENTS <---------------------




--------------------> OBJECT DEFINITION <--------------------------


  type Timer_Type is private;


--------------------> EXPORTED OPERATIONS <--------------------------




  function New_Timer return Timer_Type;



private

     task type Timer_Representation is
       entry Initialize(Time_Interval : in duration);
     end Timer_Representation;

           --The full definition may be moved to the package body
           --after implementation of the body is complete.

   type Timer_Type is access Timer_Representation;
```

```
        --pointer to a Timer_Representation

end Timer_Manager;
```

B.3 Cruise_Control Aggregate Package

B.3.1 Cruise_Control_System_Aggregate.

```
with Standard_Engineering_types;

with Throttle_Control_Manager, Speed_Sensor_Manager,
     System_States_Manager, Buttons_Manager, Timer_Manager;

pragma elaborate (Throttle_Control_Manager, Speed_Sensor_Manager,
                  System_States_Manager, Buttons_Manager,
                  Timer_Manager);


package Cruise_Control_System_Aggregate is

  package SET renames Standard_Engineering_types;

  type Cruise_Control_Representation is record
    The_Throttle_Control : Throttle_Control_Manager.Throttle_Control_Type;
    The_Speed_Sensor     : Speed_Sensor_Manager.Speed_Sensor_Type;
    The_System_States    : System_States_Manager.System_States_Type;
    The_Buttons          : Buttons_Manager.Buttons_Type;
    The_Timer            : Timer_Manager.Timer_Type;
  end record;

  Cruise_Control : Cruise_Control_Representation;

end Cruise_Control_System_Aggregate;
------------------------------------------------------------------------
package body Cruise_Control_System_Aggregate is

begin
  Cruise_Control.The_Throttle_Control :=
                   Throttle_Control_Manager.New_Throttle_Control;

  Cruise_Control.The_Speed_Sensor :=
                   Speed_Sensor_Manager.New_Speed_Sensor;

  Cruise_Control.The_System_States :=
                   System_States_Manager.New_System_States;
  Cruise_Control.The_Buttons := Buttons_Manager.New_Buttons
        (On_Button_Interrupt        =>
                          SET.Cruise_Control_Data.On_Button_Interrupt,
         Off_Button_Interrupt       =>
                          SET.Cruise_Control_Data.Off_Button_Interrupt,
         Set_Button_Interrupt       =>
                          SET.Cruise_Control_Data.Set_Button_Interrupt,
         Resume_Button_Interrupt    =>
                          SET.Cruise_Control_Data.Resume_Button_Interrupt,
         Accelerate_Button_Interrupt =>
                      SET.Cruise_Control_Data.Accelerate_Button_Interrupt,
         Brake_Pedal_Interrupt      =>
                          SET.Cruise_Control_Data.Brake_Pedal_Interrupt);

  Cruise_Control.The_Timer := Timer_Manager.New_Timer;

end Cruise_Control_System_Aggregate;
```

B.4 Connector/Event Procedures

B.4.1 Turn_On.

```
with System_States_Manager;
with Cruise_Control_Aggregate;
procedure Turn_On is

   package CCA renames Cruise_Control_Aggregate;
   package SSM renames System_States_Manager;

 begin

    SSM.Apply_Turn_On_To(CCA.Cruise_Control.The_System_States);

end Turn_On;
```

B.4.2 Set_Speed.

```
with System_States_Manager;
with Cruise_Control_Aggregate;
with Throttle_Control_Manager;
with Speed_Sensor_Manager;
procedure Set_Speed is

   package CCA renames Cruise_Control_Aggregate;
   package SSM renames System_States_Manager;
   package TCM renames Throttle_Control_Manager;
   package SS  renames Speed_Sensor_Manager;

 begin
                        ----------------------------------------------
                        --If the cruise control is turned on then
                        --engage it, set the throttle, get the current
                        --speed and save it.
                        ----------------------------------------------
  if SSM.Return_On_State_From(CCA.Cruise_Control.The_System_States) then
    SSM.Apply_Engage_To(CCA.Cruise_Control.The_System_States);
    TCM.Apply_Set_To(CCA.Cruise_Control.The_Throttle_Control);
    SSM.Apply_Desired_Speed_To
         (This_System_States => CCA.Cruise_Control.The_System_States,
          Desired_Speed      =>
               SS.Return_Speed_From(CCA.Cruise_Control.The_Speed_Sensor));
  end if;

end Set_Speed;
```

B.4.3 Update.

```
with System_States_Manager;
with Cruise_Control_Aggregate;
with Throttle_Control_Manager;
with Speed_Sensor_Manager;
with Standard_Engineering_Types;
procedure Update is

   package CCA renames Cruise_Control_Aggregate;
   package SSM renames System_States_Manager;
   package TCM renames Throttle_Control_Manager;
   package SS  renames Speed_Sensor_Manager;
   package SET renames Standard_Engineering_Types;

   Current_Speed : SET.Speed_Type :=
        SS.Return_Speed_From(CCA.Cruise_Control.The_Speed_Sensor);
   Desired_Speed : SET.Speed_Type :=
      SSM.Return_Desired_Speed_From(CCA.Cruise_Control.The_System_States);

   Speed_Difference : SET.Speed_Type;

 begin
                        ----------------------------------------------
                        --If the cruise control is engaged and the current
```

```
                             --speed doesn't equal desired speed then adjust
                             --the speed, delay, and check it again. Give up
                             --after about 10 seconds.

                             --NOTE: The Speed_Difference passed to
                             --Throttle_Control_Manager must be greater than 0
                             --and less than 11.
                             --------------------------------------------------
          for i in 1..6 loop

             exit when SSM.Return_Engaged_State_From
                  (CCA.Cruise_Control.The_System_States) = false;

             if Desired_Speed = Current_Speed then
                exit;
             elsif Desired_Speed < Current_Speed then --too fast

                Speed_Difference := Current_Speed - Desired_Speed;
                if Speed_Difference > 10 then
                   Speed_Difference := 10;
                end if;

                TCM.Apply_Change_Setting_To
                     (This_Throttle_Control =>
                               CCA.Cruise_Control.The_Throttle_Control,
                      Change => TCM.Deceleration,
                      Change_Amount => Speed_Difference);

             else                              --too slow
                Speed_Difference := Desired_Speed - Current_Speed;
                if Speed_Difference > 10 then
                   Speed_Difference := 10;
                end if;

                TCM.Apply_Change_Setting_To
                     (This_Throttle_Control =>
                               CCA.Cruise_Control.The_Throttle_Control,
                      Change => TCM.Acceleration,
                      Change_Amount => Speed_Difference);
             end if;

             if i /= 6 then
                 delay SET.Update_Interval;  --seconds
                 Current_Speed :=
                    SS.Return_Speed_From(CCA.Cruise_Control.The_Speed_Sensor);
```

```
      end if;

    end loop;

  end Update;
```

B.4.4 Brake.

```
with System_States_Manager;
with Throttle_Control_Manager;
with Cruise_Control_Aggregate;
procedure Brake is

   package CCA renames Cruise_Control_Aggregate;
   package SSM renames System_States_Manager;
   package TCM renames Throttle_Control_Manager;

 begin

    TCM.Apply_Release_To(CCA.Cruise_Control.The_Throttle_Control);
    SSM.Apply_Disengage_To(CCA.Cruise_Control.The_System_States);
end Brake;
```

B.4.5 Resume.

```
with System_States_Manager;
with Cruise_Control_Aggregate;
procedure Resume is

   package CCA renames Cruise_Control_Aggregate;
   package SSM renames System_States_Manager;

 begin
    SSM.Apply_Engage_To(CCA.Cruise_Control.The_System_States);
end Resume;
```

B.4.6 Accelerate.

```
with System_States_Manager;
with Cruise_Control_Aggregate;
with Throttle_Control_Manager;
procedure Accelerate is
```

```
   package CCA renames Cruise_Control_Aggregate;
   package SSM renames System_States_Manager;
   package TCM renames Throttle_Control_Manager;

 begin

  if SSM.Return_Engaged_State_From
                              (CCA.Cruise_Control.The_System_States) then
     TCM.Apply_Change_Setting_To
        (This_Throttle_Control => CCA.Cruise_Control.The_Throttle_Control,
         Change                => TCM.Acceleration,
         Change_Amount         => 1);
  end if;

end Accelerate;
```

B.4.7 Turn_Off.

```
with System_States_Manager;
with Cruise_Control_Aggregate;
with Throttle_Control_Manager;
procedure Turn_Off is

   package CCA renames Cruise_Control_Aggregate;
   package SSM renames System_States_Manager;
   package TCM renames Throttle_Control_Manager;

 begin

    TCM.Apply_Release_To(CCA.Cruise_Control.The_Throttle_Control);
    SSM.Apply_Disengage_To(CCA.Cruise_Control.The_System_States);
    SSM.Apply_Turn_Off_To (CCA.Cruise_Control.The_System_States);

end Turn_Off;
```

Appendix C. *Ada Package Bodies for Simulation Implementation of the Elevator Problem*

This appendix contains the implementation of the package bodies from the elevator design problem in Chapter 4 and Appendix A. This implementation is discussed in Section 4.6. Not included are the main driver and the simulation screen driver which were developed for the purpose of simulating the elevator controller on a personal computer.

The simulator works as follows: The simulation driver gets keyboard inputs from the operator, interprets them, and calls the appropriate object manager. This simulates the receiving of interrupts caused by the pushing of elevator summons and destination buttons. The object managers had to be modified somewhat to export operations to receive stimulus in this manner. The simulation screen driver draws a picture of the elevators on the screen and exports two kinds of suffered operations: an operation to move one of the elevators up and down, and operations to turn the button lights on and off.

C.1 Floor_Panel_Manager.

```
with Summons, Elevator_Screen_Control;
package body Floor_Panel_Manager is

   package ESC renames Elevator_Screen_Control;
   use Standard_Engineering_Types;


-------------------> EXPORTED OPERATIONS <-------------------------

   task body Floor_Panel_Representation is

     Local_Floor     : SET.Floor_Type;
     Local_Direction : SET.Direction_Type;
     begin
```

```
    loop
      select

        accept Up_Interrupt(From_Floor : in SET.Floor_Type) do
          Local_Floor := From_Floor;
        end Up_Interrupt;

                    --Can't have a up summons from the top floor:
        if not (Local_Floor = SET.Floor_Type'last) then
          Summons(Local_Floor,Up);
          ESC.Change_Floor_Panel_Light_To(On, Local_Floor, Up);
        end if;

      or
        accept Down_Interrupt(From_Floor : in SET.Floor_Type) do
          Local_Floor := From_Floor;
        end Down_Interrupt;

                    --Can't have a down summons from the bottom floor:
        if not (Local_Floor = SET.Floor_Type'first) then
          Summons(Local_Floor,Down);
          ESC.Change_Floor_Panel_Light_To(On, Local_Floor, Down);
        end if;

      or
        accept Light_Out(Floor     : in SET.Floor_Type;
                         Direction : in SET.Direction_Type) do

           Local_Floor     := Floor;
           Local_Direction := Direction;
        end Light_Out;

        ESC.Change_Floor_Panel_Light_To(Off, Local_Floor,
                                               Local_Direction);

      end select;
    end loop;

  end Floor_Panel_Representation;


--------------------------------------------------------------------
  function New_Floor_Panel return Floor_Panel_Type is
    Floor_Panel : Floor_Panel_Type;
  begin
     Floor_Panel := new Floor_Panel_Representation;
     return Floor_Panel;
 end New_Floor_Panel;


---------------------------------------------------------------
 procedure Apply_Light_Out_To(This_Floor_Panel : in Floor_Panel_Type;
                              Floor            : in SET.Floor_Type;
                              Direction        : in SET.Direction_Type) is

  begin

    if Direction = SET.Up then

            --There is no up summons light at the top floor:
      if Floor /= SET.Floor_Type'last then
        This_Floor_Panel.Light_Out(Floor,SET.Up);
      end if;
   else   --down

            --There is no down summons light at the bottom floor:
      if Floor /= SET.Floor_Type'first then
        This_Floor_Panel.Light_Out(Floor,SET.Down);
      end if;
   end if;


 end Apply_Light_Out_To;


--------------------------------------------------------------------
 procedure Summons(This_Floor_Panel : in Floor_Panel_Type;
                   Floor            : in SET.Floor_Type;
                   Direction        : in SET.Direction_Type) is

   begin

    if Direction = up then
      This_Floor_Panel.Up_Interrupt(Floor);
    else
      This_Floor_Panel.Down_Interrupt(Floor);
    end if;
  end Summons;


end Floor_Panel_Manager;
```

C.2 Weight_Sensor_Manager.

```
package body Weight_Sensor_Manager is



-------------------> EXPORTED OPERATIONS <--------------------------


  function New_Weight_Sensor
       (Elevator_ID : SET.Elevator_ID_Type) return Weight_Sensor_Type is

     Weight_Sensor : Weight_Sensor_Type;
    begin
      Weight_Sensor := new Weight_Sensor_Representation;
      return Weight_Sensor;

  end New_Weight_Sensor;

-------------------------------------------------------------------------
  function Return_Weight_OK_From
                (This_Weight_Sensor : Weight_Sensor_Type) return boolean is

    begin
          --Dummy routine for this simulation:
      return true;

  end Return_Weight_OK_From;



end Weight_Sensor_Manager;
```

C.3 Scheduler_Manager.

```
with Arrives, Proceed;
package body Scheduler_Manager is
```

```
use Standard_Engineering_Types;


--------------------> OBJECT DEFINITION <--------------------------


   type Floor_Stops is array (SET.Floor_Type) of boolean;

   type Schedule_Record is record
     Next_Stop     : SET.Floor_Type := SET.Floor_Type'first;
     Current_Floor : SET.Floor_Type := SET.Floor_Type'first;
     Direction     : SET.Direction_Type := SET.Parked;
     Motor_On      : boolean := false;
     Boarding      : boolean := false;
   end record;

   type Elevator_Array is array(1..SET.Elevator_ID_Type'Last) of
                                                     Schedule_Record;
   type Summons_Waiting_Record is record
     Waiting_Up      : boolean := false;
     Waiting_Down    : boolean := false;
   end record;

  type Floor_Summons_Array is array (SET.Elevator_ID_Type) of
                                              Summons_Waiting_Record;

  type Summons_Waiting_Array is array (SET.Floor_Type) of
                                                 Floor_Summons_Array;

  type Destination_Waiting_Array is array (SET.Elevator_ID_Type) of
                                                        Floor_Stops;

               --Finally, the Scheduler representation:
  type Scheduler_Representation is record
    Schedule             : Elevator_Array;
    Waiting_Summons      : Summons_Waiting_Array;
    Waiting_Destination : Destination_Waiting_Array;
  end record;


--------------------> LOCAL OPERATIONS <--------------------------
      --(not visible in package specification)

                        --------------------------------------
                        --This routine sets the "Next_Stop" and
```

```
                      --"Direction" fields of the Scheduler to
                      --set it up for its next action.
                      ---------------------------------------
procedure Set_Next(This_Scheduler : in Scheduler_Type;
                   Elevator       : in SET.Elevator_ID_Type) is



    Current_Direction : SET.Direction_Type :=
                         This_Scheduler.Schedule(Elevator).Direction;

    Current_Floor : SET.Floor_Type :=
                      This_Scheduler.Schedule(Elevator).Current_Floor;

    Floor_Set : boolean := false;

 procedure Search_Up is
        --------------------------------------------------------
        --Search up from the current floor, the next destination
        --found or summons going in the up direction becomes the
        --next floor, if none are found we search for a summons
        --going down starting at the top floor:
        --------------------------------------------------------

  begin
   for i in Current_Floor + 1..SET.Floor_Type'last loop
     if (This_Scheduler.Waiting_Destination(Elevator)(i) = true) or else
       (This_Scheduler.Waiting_Summons(i)(Elevator).Waiting_Up = true) then
         This_Scheduler.Schedule(Elevator).Next_Stop := i;
         This_Scheduler.Schedule(Elevator).Direction := SET.Up;
         Floor_Set := true;
         exit;
      end if;
   end loop;
  if not Floor_Set then
    for i in reverse Current_Floor + 1..SET.Floor_Type'last loop
      if (This_Scheduler.Waiting_Summons(i)(Elevator).
                                        Waiting_Down = true) then
         This_Scheduler.Schedule(Elevator).Next_Stop := i;
         This_Scheduler.Schedule(Elevator).Direction := SET.Up;
         Floor_Set := true;
         exit;
      end if;
    end loop;
  end if;
  end Search_Up;

  procedure Search_Down is
         ---------------------------------------------------------
         --Search Down from the current floor, the next destination
         --found or Summons going down the down direction becomes the
         --next floor, if none are found we search for a summons
         --going up starting at the bottom floor:
         ---------------------------------------------------------

   begin
    for i in reverse SET.Floor_Type'first..Current_Floor - 1 loop
      if (This_Scheduler.Waiting_Destination(Elevator)(i) = true) or else
        (This_Scheduler.Waiting_Summons(i)(Elevator).
                                          Waiting_Down = true) then
          This_Scheduler.Schedule(Elevator).Next_Stop := i;
          This_Scheduler.Schedule(Elevator).Direction := SET.Down;
          Floor_Set := true;
          exit;
       end if;
    end loop;
   if not Floor_Set then
     for i in SET.Floor_Type'first..Current_Floor - 1 loop
       if (This_Scheduler.Waiting_Summons(i)(Elevator).
                                          Waiting_Up = true) then
          This_Scheduler.Schedule(Elevator).Next_Stop := i;
          This_Scheduler.Schedule(Elevator).Direction := SET.Down;
          Floor_Set := true;
          exit;
       end if;
     end loop;
   end if;

  end Search_Down;

begin
        ---------------------------------------------------
        --Always catch all scheduled floors in the current
        --direction first. Park the elevator if there are
        --no more floors scheduled for it:
        ---------------------------------------------------

  if Current_Direction = SET.Down then
    Search_Down;
```

```
      if not Floor_Set then
        Search_Up;
      end if;
      if not Floor_Set then
        This_Scheduler.Schedule(Elevator).Direction := SET.Parked;
      end if;

    else --Current_Direction = SET.Up or SET.Parked
      Search_Up;
      if not Floor_Set then
        Search_Down;
      end if;
      if not Floor_Set then
        This_Scheduler.Schedule(Elevator).Direction := SET.Parked;
      end if;
    end if;

end Set_Next;

--------------------> EXPORTED OPERATIONS <--------------------------


  function New_Scheduler return Scheduler_Type is

      Local_Scheduler : Scheduler_Type;

    begin

      Local_Scheduler := new Scheduler_Representation;
      return Local_Scheduler;

  end New_Scheduler;


----------------------------------------------------------------------
procedure Apply_Summons_To(This_Scheduler : in Scheduler_Type;
                           From_Floor     : in SET.Floor_Type;
                           Direction      : in SET.Direction_Type) is


              ----------------------------------------------------
              --This routine decides which elevator to schedule for
              --the summons, and sends it on its way if it is not
              --being used:
              ----------------------------------------------------
```

```
Number_Candidates : integer := 0;

type This_Candidate is record
  Candidate : boolean := false;
  Distance  : integer := 2000;
end record;

Closest     : integer := 2000;
Closest_One : SET.Elevator_ID_Type;

type Candidate_List is array(
               SET.Elevator_ID_Type) of This_Candidate;

Candidates  : Candidate_List;
Last_Direction : SET.Direction_Type;
Idle_Elevator : boolean := false;

begin

                --check for parked elevators at this floor:
  for i in reverse SET.Elevator_ID_Type loop
    if (This_Scheduler.Schedule(i).Direction = SET.Parked) and then
       (This_Scheduler.Schedule(i).Current_Floor = From_Floor) then
      Idle_Elevator := true;
      Closest_One := i;
      exit;
    end if;
  end loop;

                --check for parked elevators:
  if not Idle_Elevator then
    for i in reverse SET.Elevator_ID_Type loop
      if This_Scheduler.Schedule(i).Direction = SET.Parked then
        Idle_Elevator := true;
        Closest_One := i;
        exit;
      end if;
    end loop;
  end if;

                    --------------------------------------------
                    --Since all the elevators are being used,
                    --schedule one using heuristics of trying to
                    --find the closest one going in the right
```

```
                  --direction:
                  -------------------------------------------
if not Idle_Elevator then

                       --------------------------------------------
                       --Determine how far each elevator is from the
                       --floor where someone is waiting:
                       --------------------------------------------
    for i in SET.Elevator_ID_Type loop
      Candidates(i).distance :=
         This_Scheduler.Schedule(i).Current_Floor - From_Floor;
    end loop;

                       ----------------------------------------------
                       --Establish candidate elevators to go answer
                       --the summons as those going toward the summons
                       --floor:
                       ----------------------------------------------
    for i in SET.Elevator_ID_Type loop
       if (This_Scheduler.Schedule(i).Direction = SET.Up) and
          Candidates(i).distance <= 0 then
            Candidates(i).Candidate := true;
            Number_Candidates := Number_Candidates + 1;
        elsif (This_Scheduler.Schedule(i).Direction = SET.Down) and
          Candidates(i).distance >= 0 then
            Candidates(i).Candidate := true;
            Number_Candidates := Number_Candidates + 1;
        elsif This_Scheduler.Schedule(i).Direction = SET.Parked then
            Candidates(i).Candidate := true;
            Number_Candidates := Number_Candidates + 1;
        end if;
    end loop;

                       ----------------------------------------
                       --If no candidates then send the closest
                       --elevator.
                       ----------------------------------------
     if Number_Candidates = 0 then
       for i in SET.Elevator_ID_Type loop
         if abs(Candidates(i).distance) < Closest then
            Closest_one := i;
            Closest := Candidates(i).distance;
         end if;
       end loop;
```

```
                              ------------------------------
                              --If one candidate then send it:
                              ------------------------------
        elsif Number_Candidates = 1 then
          for i in SET.Elevator_ID_Type loop
            if Candidates(i).Candidate = true then
              Closest_One := i;
              exit;
            end if;
          end loop;

                             ----------------------------------------
                             --If more that one candidate then send the
                             --closest of these:
                             ----------------------------------------
        else --more than one candidate
          for i in SET.Elevator_ID_Type loop
            if (Candidates(i).Candidate = true) and then
               (abs(Candidates(i).distance)) <= Closest then
               Closest_one := i;
               Closest := Candidates(i).distance;
            end if;
          end loop;

        end if;
   end if;

                      -----------------------------------------------
                      --An elevator is selected so now we need to
                      --schedule it. Assign the summons to the summons
                      --table. If the assigned elevator is already at
                      --the right floor and parked, then don't schedule
                      --it, just call the connector "Arrives:"
                      -----------------------------------------------
if  (From_Floor = This_Scheduler.Schedule(Closest_One).Current_Floor)
  and then
    (This_Scheduler.Schedule(Closest_One).Direction = parked) then

       Arrives(Closest,From_Floor, Direction);
       This_Scheduler.Schedule(Closest_One).Motor_On := false;

else  --schedule summons and call elevator if not being used:

   if Direction = SET.Up then
     This_Scheduler.Waiting_Summons(From_Floor)
```

```
                                   (Closest_One).Waiting_Up := true;
      else  --down
        This_Scheduler.Waiting_Summons(From_Floor)
                                   (Closest_One).Waiting_Down := true;
      end if;

                              ----------------------------------------------
                              --Set the elevator for next stop and direction:
                              ----------------------------------------------
      Last_Direction := This_Scheduler.Schedule(Closest_One).Direction;
      Set_Next(This_Scheduler,Closest_One);

                              ------- -------------------------------------
                              --If the motor is off and it's not stopped for
                              --boarding then dispatch the elevator:
                              ---------------------------------------------
      if (This_Scheduler.Schedule(Closest_One).Motor_On = false) and then
         (This_Scheduler.Schedule(Closest_One).Boarding = false) then

         Proceed(Closest_One,
           This_Scheduler.Schedule(Closest_One).Direction);
           This_Scheduler.Schedule(Closest_One).Motor_On := true;
      end if;
   end if;

  end Apply_Summons_To;

--------------------------------------------------------------------
  procedure Apply_Destination_Request_To
                             (This_Scheduler : in Scheduler_Type;
                              Elevator       : in SET.Elevator_ID_Type;
                              Floor          : in SET.Floor_Type) is

    begin


      This_Scheduler.Waiting_Destination(Elevator)(Floor) := true;


                        ----------------------------------------------
                        --Set the elevator for next stop and direction:
                        ----------------------------------------------
      Set_Next(This_Scheduler,Elevator);

                        ---------------------------------------------
```

```
                  --If the motor is off and it's not stopped for
                  --boarding then dispatch the elevator:
                  ----------------------------------------------
    if (This_Scheduler.Schedule(Elevator).Motor_On = false) and then
       (This_Scheduler.Schedule(Elevator).Boarding = false) then
      Proceed(Elevator,
         This_Scheduler.Schedule(Elevator).Direction);
         This_Scheduler.Schedule(Elevator).Motor_On := true;
    end if;

 end Apply_Destination_Request_To;

--------------------------------------------------------------------
 procedure Apply_Floor_Approaching
                            (This_Scheduler : in Scheduler_Type;
                             Floor          : in SET.Floor_Type;
                             Elevator       : in SET.Elevator_ID_Type) is

      Summons_Direction  : SET.Direction_Type;
      Summons_Waiting_Up : boolean := This_Scheduler.Waiting_Summons(Floor)
                             (Elevator).Waiting_Up;
      Summons_Waiting_Down : boolean := This_Scheduler.
                                                   Waiting_Summons(Floor)
                             (Elevator).Waiting_Down;

    begin

      This_Scheduler.Schedule(Elevator).Current_Floor := Floor;

                             --Stop if scheduled to do so:
      if This_Scheduler.Schedule(Elevator).Next_Stop = Floor then

                             --set things up for the next stop:
        Set_Next(This_Scheduler,Elevator);
        Summons_Direction := This_Scheduler.Schedule(Elevator).Direction;

                        -------------------------------------------
                        --Clear schedule of appropriate summons and
                        --destinations. Want to catch at least one
                        --summons if one exist no matter which way
                        --we came from.  Look from the direction we
                        --came from first:
                        -------------------------------------------
      if Summons_Direction = Down then
        if Summons_Waiting_Down  then
        This_Scheduler.Waiting_Summons(Floor)
                             (Elevator).Waiting_Down := false;
        Summons_Direction := Down;
        Summons_Waiting_Down := false;

    elsif Summons_Waiting_Up then
        This_Scheduler.Waiting_Summons(Floor)
                             (Elevator).Waiting_Up := false;
        Summons_Direction := Up;
        Summons_Waiting_Up := false;
    end if;
  else
    if Summons_Waiting_Up then
        This_Scheduler.Waiting_Summons(Floor)
                             (Elevator).Waiting_Up := false;
        Summons_Direction := Up;
        Summons_Waiting_Up := false;
    elsif Summons_Waiting_Down then
        This_Scheduler.Waiting_Summons(Floor)
                             (Elevator).Waiting_Down := false;
        Summons_Direction := Down;
        Summons_Waiting_Down := false;
    end if;
  end if;
                --clear destination request:
    This_Scheduler.Waiting_Destination(Elevator)(Floor) := false;

                --stop the elevator:
    Arrives(Elevator,Floor, Summons_Direction);
    This_Scheduler.Schedule(Elevator).Boarding := true;
    This_Scheduler.Schedule(Elevator).Motor_On := false;




                ---------------------------------------
                --Waiting for boarding if trips to more
                --floors are pending:
                ---------------------------------------
    if This_Scheduler.Schedule
              (Elevator).Direction /= SET.Parked then
        delay 6.0; --so passengers can board before departure
    end if;

                 -----------------------------------------------
                 --Reset for the next floor in case new requests
```

```
                      --were added to the schedule during boarding.
                      --Have the elevator continue now to the next
                      --stop if requests are pending:
                      ----------------------------------------------
         Set_Next(This_Scheduler,Elevator);
         if This_Scheduler.Schedule
                  (Elevator).Direction /= SET.Parked then

             Proceed(Elevator,
                This_Scheduler.Schedule(Elevator).Direction);
             This_Scheduler.Schedule(Elevator).Motor_On := true;
         end if;
         This_Scheduler.Schedule(Elevator).Boarding := false;

       end if;

  end Apply_Floor_Approaching;

end Scheduler_Manager;
```

C.4 Location_Panel_Manager.

```
with Elevator_Screen_Control;
package body Location_Panel_Manager is

   package ESC renames Elevator_Screen_Control;
   use Standard_Engineering_Types;


--------------------> EXPORTED OPERATIONS <--------------------------


----------------------------------------------------------------------
  function New_Location_Panel(Elevator_ID : SET.Elevator_ID_Type)
                                            return Location_Panel_Type is
    Local_Loc_Panel : Location_Panel_Type;
  begin
    Local_Loc_Panel := new Location_Panel_Representation;
    Local_Loc_Panel.Elevator_ID := Elevator_ID;
    Local_Loc_Panel.Current_Floor_Indicator_Lit :=
                                              SET.Floor_Type'first;
    return Local_Loc_Panel;

  end New_Location_Panel;
```

```
----------------------------------------------------------------------
  procedure Apply_Update_Location_Indicator
                      (This_Location_Panel : in Location_Panel_Type;
                       New_Floor           : in SET.Floor_Type) is

    Move_Direction : SET.Direction_Type;
   begin

     if New_Floor > This_Location_Panel.Current_Floor_Indicator_Lit then
       Move_Direction := Up;
     elsif New_Floor < This_Location_Panel.Current_Floor_Indicator_Lit then
       Move_Direction := Down;
     else
       return; --nowhere to move;
     end if;

     This_Location_Panel.Current_Floor_Indicator_Lit := New_Floor;

                       ---------------------------------------------
                       --Call the elevator simulation screen telling
                       --it to move the elevator:
                       ---------------------------------------------
     ESC.Move_Elevator (Direction  => Move_Direction,
                       Elevator_ID => This_Location_Panel.Elevator_ID);


  end Apply_Update_Location_Indicator;


end Location_Panel_Manager;
```

C.5 Control_Panel_Manager.

```
with Destination_Requested;
with Elevator_Screen_Control;
package body Control_Panel_Manager is

   package ESC renames Elevator_Screen_Control;
   use Standard_Engineering_Types;


--------------------> EXPORTED OPERATIONS <---------------------------
```

```
----------------------------------------------------------------------
  function New_Control_Panel (Elevator_ID : SET.Elevator_ID_Type)
                                           return Control_Panel_Type is

     Local_Control_Panel : Control_Panel_Type;

   begin
     Local_Control_Panel := new Control_Panel_Representation;
     Local_Control_Panel.Elevator_ID := Elevator_ID;
     return Local_Control_Panel;

  end New_Control_Panel;


----------------------------------------------------------------------
  procedure Apply_Light_Out_To
                         (This_Control_Panel : in Control_Panel_Type;
                          Floor              : in SET.Floor_Type) is

   begin

                       ---------------------------------------------
                       --Call the elevator simulation screen telling
                       --it to turn-off a button light:
                       ---------------------------------------------
     ESC.Change_Elevator_Panel_Light_To(
                     Off,
                     This_Control_Panel.Elevator_ID,
                     Floor);


  end Apply_Light_Out_To;

----------------------------------------------------------------------
       --subprogram added for simulation implementation:
  procedure Destination_Selected(Elevator_ID : SET.Elevator_ID_Type;
                                 Floor       : SET.Floor_Type) is

   begin
     Destination_Requested(Elevator_ID,Floor);
     ESC.Change_Elevator_Panel_Light_To(On, Elevator_ID, Floor);

  end Destination_Selected;

end Control_Panel_Manager;
```

C.6 Floor_Sensor_Manager.

```
with Floor_Approaching;
package body Floor_Sensor_Manager is



--------------------> EXPORTED OPERATIONS <--------------------------

   task body Floor_Sensor_Representation is
                    --------------------------------------------------
                    --Floor Sensor was retained as a task in the
                    --simulation to prevent the motor task from having
                    --to wait for the Scheduler to board passengers;
                    --this task waits instead:
                    --------------------------------------------------


       Elevator     : SET.Elevator_ID_Type;
       Floor_Number : SET.Floor_Type;

      begin

        accept Initialize (Elevator_ID : in SET.Elevator_ID_Type) do
          Elevator := Elevator_ID;
        end Initialize;

        loop
          accept Floor_Sensor_Interrupt(Floor : in SET.Floor_Type) do
            Floor_Number := Floor;
          end Floor_Sensor_Interrupt;
          Floor_Approaching(Elevator,Floor_Number);
        end loop;

   end Floor_Sensor_Representation;


---------------------------------------------------------------------
  function New_Floor_Sensor (Elevator_ID : SET.Elevator_ID_Type)
                                           return Floor_Sensor_Type is
      Floor_Sensor : Floor_Sensor_Type;

   begin
```

```
    Floor_Sensor := new Floor_Sensor_Representation;
    Floor_Sensor.Initialize(Elevator_ID);
    return Floor_Sensor;

  end New_Floor_Sensor;



----------------------------------------------------------------------
      --procedure added for the simulation implementation:
  procedure Floor_Approacing(Floor_Sensor : in Floor_Sensor_Type;
                             Floor : in SET.Floor_Type) is

    begin
     Floor_Sensor.Floor_Sensor_Interrupt(Floor);
  end Floor_Approacing;


end Floor_Sensor_Manager;
```

C.7 Motor_Manager.

```
with Floor_Sensor_Manager, Elevator_System_Aggregate;
package body Motor_Manager is
   use Standard_Engineering_Types;
   package ESA renames Elevator_System_Aggregate;


-----------------------------------------------------------------
  task body Motor_Representation is
               ------------------------------------------------
               --This task simulates the action of the motor and
               --the movement of the elevator. It notifies the
               --Floor Sensor when a floor is approaching:
               ------------------------------------------------
    Elevator          : SET.Elevator_ID_Type;
    Current_Floor     : SET.Floor_Type := SET.Floor_Type'first;
    Floor_Delay       : duration := 2.0;
    Current_Direction : SET.Direction_Type := SET.parked;

     begin

      accept Initialize (Elevator_ID : in SET.Elevator_ID_Type) do
        Elevator := Elevator_ID;
      end Initialize;
```

```
loop
  select     --stopped, waiting for motion command:

    accept Motor_Command_Up;
      if Current_Floor /= SET.Floor_Type'last then
        Current_Floor := Current_Floor + 1;
      end if;
      Current_Direction := SET.Up;
      Floor_Sensor_Manager.Floor_Approacing(
               ESA.Elevators(Elevator).The_Floor_Sensor,
                                              Current_Floor);

  or
    accept Motor_Command_Down;
      if Current_Floor /= SET.Floor_Type'first then
        Current_Floor := Current_Floor - 1;
      end if;
      Current_Direction := SET.Down;
      Floor_Sensor_Manager.Floor_Approacing(
               ESA.Elevators(Elevator).The_Floor_Sensor,
                                              Current_Floor);

  or --to avoid lock up if this one is called when already stopped:
    accept Motor_Command_Stop;

  end select;

                 --motion loop:
  if Current_Direction /= SET.Parked then
    loop
      select
        accept Motor_Command_Stop;
        Current_Direction := SET.Parked;
        exit;

         --------------------------------------------------------
         --These next two accepts are included here to avoid
         --lockup if move commands are received when the elevator
         --is already moving:
         --------------------------------------------------------
      or
         accept Motor_Command_Up;

      or
```

```
             accept Motor_Command_Down;

          or
            delay Floor_Delay;
                    -------------------------------------
                    --Increment floor and keep moving,
                    --reverse directions if at top moving
                    --up or bottom moving down:
                    -------------------------------------
            if Current_Direction = SET.Up then
              if Current_Floor /= SET.Floor_Type'last then
                 Current_Floor := Current_Floor + 1;
              else
                 Current_Floor := Current_Floor - 1;
                 Current_Direction := SET.Down;
              end if;
            else
              if Current_Floor /= SET.Floor_Type'first then
                 Current_Floor := Current_Floor - 1;
              else
                 Current_Floor := Current_Floor + 1;
                   Current_Direction := SET.Up;
                end if;
            end if;

            Floor_Sensor_Manager.Floor_Approacing(
                     ESA.Elevators(Elevator).The_Floor_Sensor,
                                                 Current_Floor);

          end select;
        end loop;
      end if;
    end loop;
   end Motor_Representation;

-------------------------------------------------------------------
  function New_Motor(Elevator_ID : in SET.Elevator_ID_Type)
                                                  return Motor_Type is
    Motor : Motor_Type;

  begin
    Motor := new Motor_Representation;
    Motor.Initialize(Elevator_ID);
    return Motor;
  end New_Motor;

----------------------------------------------------------------------
  procedure Apply_Go_To(This_Motor : in Motor_Type;
                        Direction  : in SET.Direction_Type) is

   begin
     if Direction = Down then
       This_Motor.Motor_Command_Down;
     else
       This_Motor.Motor_Command_Up;
     end if;
   end Apply_Go_To;


---------------------------------------------------------------------
  procedure Apply_Stop_To(This_Motor : in Motor_Type) is

   begin

       This_Motor.Motor_Command_Stop;
  end Apply_Stop_To;

end Motor_Manager;
```

Bibliography

AFIT, 1990. Air Force Institute of Technology (AFIT). *Object Oriented Requirements Determination, AFIT/ENG Working Paper, sic.* Technical Report, Air Force Institute of Technology (AFIT), 1990.

Batory and others, 1988. Batory, D S, et al. *Construction of File Management Systems from Software Components.* Technical Report, University of Texas, Austin, TX, 1988. Technical Report TR-88-36.

Berard, 1990b. Berard, Edward. "Object Oriented Design." Unpublished Paper Sent Directly from Mr Berard via Electronic Mail, July 1990.

Berard, 1990a. Berard, Edward. "Object Oriented Domain Analysis." Posted in the comp.object newgroup of a public bulletin board, Message-ID: 637@ajpo.sei.cmu.edu, January 1990.

Biggerstaff and Richter, 1987. Biggerstaff, Ted and Charles Richter. "Reusability Framework, Assessment, and Directions," *IEEE Software*, *4*(2):41–49 (March 1987).

Booch, 1983. Booch, Grady. *Software Engineering with Ada.* The Benjamin/Cummings Publishing Company, Inc., 1983.

Booch, 1987. Booch, Grady. *Software Components with Ada.* The Benjamin/Cummings Publishing Company, Inc., 1987.

Booch, 1991. Booch, Grady. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991 (sic).

Brown and Quanrud, 1988. Brown, G R and R B Quanrud. "The Generic Architecture Approach To Reusable Software." In *Proceedings of the Sixth National Conference On Ada Technology, Arlington, VA, Mar 14-17*, pages 390–394, 1988.

CCSO, 1988. Command & Control Systems Office (CCSO). *Reuse of Ada Software Modules.* Technical Report, CCSO, Standard Systems Center, AFCC, USAF, 1988.

D'Ippolito, 1989. D'Ippolito, Richard S. "Using Models in Software Engineering." In *Proceedings of Tri-Ada-89, Pittsburgh, PA*, pages 256–264, October 1989.

DoD-STD 2167A, 1985. DoD-STD 2167A. *Military Standard, Defense System Software Development*, February 1985.

EVB, 1989. EVB Software Engineering Inc. *Object Oriented Requirements Analysis.* Frederick, MD, 1989. Slides From an OORA Course.

Goyden, 1989. Goyden, Maj Mike. "The Software Lifecycle with Ada: A Command and Control Application." In *Proceedings of Tri-Ada-89, Pittsburgh, PA*, pages 40–55, October 1989.

Kaiser and Garlan, 1987. Kaiser, Gail E and David Garlan. "Melding Software Systems from Reusable Building Blocks," *IEEE Software*, *4*(4):17–24 (July 1987).

Kiem, 1989. Kiem, Eric. "The KEYSTONE System Design Methodology," *ACM Ada Letters*, *9*(5):101–108 (July/August 1989).

Ladden, 1989. Ladden, Richard M. "A Survey of Issues to be Considered in the Development of an Object-Oriented Development Methodology for Ada," *ACM Ada Letters*, *9*(2):78–89 (March/April 1989).

Rissman and others, 1988. Lee, K J, et al. *An OOD Paradigm for Flight Simulators, 2nd Edition, CMU/SEI-88-TR-30*. Technical Report, Software Engineering Institute, 1988.

Rissman and others, 1989a. Lee, Kenneth J and Michael S Rissman. *An Object-Oriented Solution Example: A Flight Simulator Electrical System CMU/SEI-89-TR-5*. Technical Report, Software Engineering Institute, 1989.

March, 1989. March, Steven G. *An Object Oriented Analysis Method For Ada and Embedded Systems*. MS thesis, AFIT/GCS/ENC/89D-1, Air Force Institute of Technology, 1989 (ADA202579).

Meyer, 1987. Meyer, Bertrand. "Reusability: The Case for Object-Oriented Design," *IEEE Software*, *4*(2):50–64 (March 1987).

Parnas, 1976. Parnas, David L. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, *SE-2*(1):1–9 (March 1976).

Plinta and Lee, 1989. Plinta, Charles and Kenneth Lee. "A Model Solution for C^3I Domain." In *Proceedings of Tri-Ada-89, Pittsburgh, PA*, pages 56–67, October 1989.

Rissman and others, 1989b. Plinta, Charles, et al. *A Model Solution for C^3I Message Translation and Validation, CMU/SEI-89-TR-12*. Technical Report, Software Engineering Institute, 1989.

Pressman, 1987. Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (2nd Edition). McGraw-Hill, Inc., 1987.

Prieto-Dïaz, 1987. Prieto-Dïaz, Rubén. "Domain Analysis For Reusability." In *Proceedings of COMPSAC'87*, pages 23–29, 1987.

Rajlich, 1984. Rajlich, R. "SNAP - A Language and Environment for Programming-in-the Large." In *Proceedings of the IEEE Workshop on Languages for Automation*, pages 192–195, 1984.

Rajlich, 1985. Rajlich, R. "Paradigms for Design and Implementation In Ada," *Communications of the ACM*, *28*(7):718-727 (July 1985).

Rissman and others, 1989c. Rissman, M., et al. Personnel meetings with members of the Software Architectures Engineering Project team of the Software Engineering Institute during 1989 and 1990.

Ruegsegger, 1988. Ruegsegger, Ted. "Making Reuse Pay: The SIDPERS-3 RAPID Center," *IEEE Communications*, *26*(8):16-_4 (August 1988).

Seidewitz, 1989. Seidewitz, E. "General Object-Oriented Software Development: Background and Experience," *The Journal of Systems and Software*, *9*:95-108 (1989).

Smith, 1990. Smith, Connie U. *Performance Engineering of Software Systems*. The Addison-Wesley Publishing Company, 1990.

SofTech, 1985. SofTech Inc. *Ada Reusability Guidelines*. Technical Report, SofTech, Inc, 1985.

Sommerville, 1989. Sommerville, Ian. *Software Engineering* (3rd Edition). Addison-Wesley Publishing Company, 1989.

St. Dennis, 1987. St. Dennis, Richard J. "Reusable Ada Software Guidelines." In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, pages 513-520, 1987.

Tracz, 1986. Tracz, William J. "Why Reusable Software Isn't." In *Proceedings of the Workshop on Future Directions in Computer Architecture and Software*, pages 171-177, 1986.

Umphress, 1990. Umphress, David A. "OOA vs OOD." Posted in the comp.object newgroup of a public bulletin board, Message-ID: 1675@blackbird.afit.af.mil, 1990.

Vita

Captain Kelly L. Spicer was born on 19 June 1955 in Alexandria, Virginia. He graduated from Palo Verde High School in Tucson, Arizona, in 1973. He graduated with a Bachelor of Science degree in Renewable Natural Resources from the University of Arizona in 1982. He graduated from the Air Force Officer Training School in 1984. He graduated from Central State University in Oklahoma with a second Bachelor of Science degree (computer science) in 1988.

Captain Spicer was assigned to the Command and Control Systems Office (CCSO) (Air Force Communications Command) at Tinker AFB, OK, in 1985. While there, he served as part of an Ada software development team who developed the software for the Standard Automated Remote to AUTODIN Host (SARAH) message preparation and communication system. Captain Spicer entered the Air Force Institute of Technology, School of Engineering, in May of 1989.

Permanent address: 8100 Calle Potrero
Tucson, Arizona 85715

REPORT DOCUMENTATION PAGE	Form Approved OMB No 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE MAPPING AN OBJECT-ORIENTED REQUIREMENTS ANALYSIS TO A DESIGN ARCHITECTURE THAT SUPPORTS DESIGN AND COMPONENT REUSE	5. FUNDING NUMBERS
6. AUTHOR(S) Kelly L. Spicer, Captain, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/90D-13
9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Design reuse has more potential for increasing the productivity of software development and maintenance than do traditional approaches to software reuse. Current software development methods do not promote design reuse. Reusable designs should apply within some application domain, have a consistent structure, provide a method for instantiating the design, avoid object nesting, and promote reuse of smaller components. A design mapping method from an object-oriented requirements analysis to a design adhering to the foregoing principles is presented. The method involves two transformation steps and introduces four representation tools for conducting the transformations. The second step produces Ada specifications. Design templates are used. The method is applied to two problems and one is implemented.

14. SUBJECT TERMS Software Design Reuse, Software Reuse, Software Engineering, Ada Programming Language, Object-Oriented, Object-Oriented Requirements Analysis	15. NUMBER OF PAGES 211
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL