DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GCS/ENG/90D-01

DETERMINING CONCURRENCY IN OBJECT-ORIENTED DESIGN OF REAL-TIME EMBEDDED SYSTEMS USING ADA

THESIS

Kenneth D. Baum
Captain, USAF

AFIT/GCS/ENG/90D-01

AFIT/GCS/ENG/90D-01

DETERMINING CONCURRENCY IN OBJECT-ORIENTED DESIGN OF REAL-TIME EMBEDDED SYSTEMS USING ADA

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(Computer Science)

Kenneth D. Baum, B.S.
Captain, USAF

December, 1990

Approved for public release; distribution unlimited

Acknowledgments

This thesis would not have been possible without the help and understanding of a number of people. Any merit this work has is a direct result of that support, and to experience that support has meant as much to me personally as the thesis experience has meant to me professionally.

This work was sponsored by the Air Force Office of Scientific Research, to whom I am grateful. I also want to thank Dr. Bo Sanden of George Mason University for spending time with me discussing my work. And to my advisor, Dr. David Umphress, thanks for putting up with me. I couldn't have completed the task without your help and encouragement.

My family has been an incredible support to me during this time. To Erich and Elise, thanks for laughing at all my stupid jokes and tales of eccentric AFIT professors; I know you didn't believe half of them. To Simon and Nathan, thanks for your relentless love and affection during a time when daddy wasn't there a lot. And to my wife, Margie, thanks for being patient and understanding; without you I would have reached escape velocity long ago. I love you.

Finally, I want to praise the Lord Jesus Christ for leading me and my family throughout this time. "For thou art great and doest wondrous things: thou art God alone." Ps 86:10

Kenneth D. Baum

Table of Contents

List of Figures

Figure		Page

Figure | Page

List of Tables

AFIT/GCS/ENG/90D-01

Abstract

One of the characteristics of real-time systems is concurrency. Designers of real-time systems have traditionally determined system concurrency at implementation time using the facilities of a cyclic executive. With the advent of programming language constructs for specifying concurrency, determining concurrency at design time has become a possibility.

Several design methods, all of which are extensions of either Structured Design or Jackson System Development, provide heuristics to help the designer make concurrency decisions. The object-oriented approach, however, has no corresponding heuristics to aid designers of real-time sytems.

The purpose of this thesis was to develop heuristics to help designers make concurrency decisions in developing object-oriented designs of real-time systems. This was accomplished by examining existing heuristics from other design methods and applying them to the object-oriented paradigm.

Four heuristics were developed, the first of which exploits the potential in object-oriented design to model the problem-space. The other three heuristics deal with concurrency which is not necessarily reflected in the problem-space, but must be implemented for practical reasons.

The heuristics were validated by applying them to a sample problem, then having the heuristics and the design of the sample problem evaluated by a group of software engineering experts. (KR)

DETERMINING CONCURRENCY IN OBJECT-ORIENTED DESIGN OF REAL-TIME EMBEDDED SYSTEMS USING ADA

I. Introduction

The design of embedded, real-time systems is considered one of the most complex software related activities[Levi and Agrawala 1987:3]. Journal articles and textbooks dealing with real-time software design have increased in number and frequency as researchers attempt to reduce complexity and help designers in their task. This thesis discusses the application of object-oriented design techniques to real-time systems.

1.1 Background

An embedded computer systems is one in which the computer is a critical part of a larger system[Scannell, *et al.* 1986:3]. These systems are usually large, complex, and subject to strict reliability and timing requirements[Booch 1987b:15].

A real-time software system is one which must respond to events or conditions in the external environment within a specified time period[IEEE 1983]. As this aspect of embedded systems leads directly to a consideration of concurrency in the system, this thesis focuses on real-time software design.

One of the primary characteristics of real-time systems is concurrency[Gomaa 1989b], which occurs when the execution of two or more processes is overlapped in time, *i.e.*, at least one process begins execution prior to the termination of some other process. These processes may be distributed on multiple processors or share a single processor.

Traditionally, concurrency in real-time systems has been handled via a cyclic executive, which is essentially a real-time extension to the operating system, providing facilities for creation, execution, and termination of concurrent processes[Sha and Goodenough:1]. Under a cyclic executive, each process is allotted a certain amount of execution time, at the end of which the process is suspended and another process scheduled. Handling concurrency then becomes strictly an implementation issue, since software modules that cannot execute within their time frame must then be decomposed into smaller components, not on the basis of design considerations, but on the basis of execution time.

The Ada programming language, introduced in the 1980's, provides language constructs for specifying concurrent processes without forcing the programmer to explicitly use a real-time executive. This enables the designer to make concurrency decisions at design time based on sound design principles, rather than at implementation time based on timing considerations.

The designer of real-time systems, therefore, must identify which processes in the software design are concurrent and which are not. Until recently, there has been little guidance for identifying concurrency, but several researchers have developed heuristics for determining when a process should be implemented as a concurrent process[Gomaa 1984, Nielsen and Shumate 1988, Sanden 1989]. These heuristics are presented in the context of Structured Design[Ward and Mellor 1985] or Jackson System Development[Jackson 1983]. One method that does not have comparable heuristics is object-oriented design[Kelly 1987:245].

Object-oriented design models the software as objects corresponding to entities in the real world[Booch 1987b:47]. Associated with each object is a set of operations which acts on the object. The software system is implemented by specifying the interaction of the objects via their operations.

The object-oriented method followed in this thesis is that described by Booch[Booch 1991], which is an iterative process of identifying objects and operations, determining the visibility and interfaces between objects, and then implementing the objects. As new objects are encountered during the design, the process is repeated. This continues until all objects are implemented. Chapter three contains a fuller discussion of object-oriented design and Booch's method.

1.2 Problem

At present, designers of object-oriented real-time systems have little guidance in determining concurrency in their designs[Kelly 1987]. The objective of this thesis is to develop heuristics for identifying concurrency in an object-oriented, real-time design.

Specifically, the objectives are as follows:

- Determine what heuristics exist for determining concurrency using other design methods.
- Define heuristics for determining concurrency using object-oriented design.
- Validate the heuristics by applying them to a sample problem and then having a panel of experts pass judgement on the validity of the heuristics.

1.3 Scope

This thesis concentrates on real-time systems implemented on single-processors. Concurrency in distributed, multi-processor systems depends on factors external to the design, such as the processor interconnection network, the communication mechanism, and the number of processors available. Assuming a single-processor environment allows the designer to focus on the design itself, independent of implementation platform. Even in a distributed environment there may be several processes executing on the same processor, so the single-processor heuristics apply in any case.

1.4 Assumptions

The design principles developed in this thesis are independent of implementation language. However, the language used to verify the principles is Ada. Accordingly, the benefits and constraints of the Ada tasking model have affected the resulting design.

1.5 Approach

The research to achieve the goals of this thesis was accomplished in the following stages:

1. Literature Survey. Over the past 25 years a vast amount of research concerning software system design has been done. A survey of this research was conducted, focusing on current developments in the design of real-time systems, and in determining concurrency in these designs. Specifically, three design paradigms were investigated: real-time extensions to Structured Analysis/Structured Design (SA/SD)[Ward and Mellor 1985], Jackson System Development[Jackson 1983], and object-oriented design[Booch 1991]. The results of this survey are in chapter two of this thesis.

2. Develop Design Heuristics. Based on the principles and heuristics examined in the literature survey, a set of heuristics specifically addressing concurrency in object-oriented real-time systems were developed. The heuristics are described in chapter three.

3. Validation of Heuristics. The validation of the concurrency heuristics took place in two stages. First, the heuristics were applied to a sample problem. An air traffic control simulation (ATC) was selected because it exhibited sufficient concurrency to demonstrate the heuristics, while being small enough to manage in an academic environment. The discussion of the ATC design is in chapter

four, and the object-oriented requirements analysis and the Ada specifications for the architectural design can be found in appendix A.

For the second stage of validation, the heuristics were distributed to several experts in software engineering whose opinions on various aspects of the heuristics were tabulated. Chapter five contains a detailed discussion of this effort and appendix B contains the validation package.

1.6 Thesis Organization

The thesis is organized to follow the stages of research outlined in the *Approach* section. Chapter two presents a review of current literature concerning concurrency in the design of real-time software systems. Chapter three outlines a set of heuristics which designers can apply to object-oriented design of real-time systems to determine concurrency in the system. Chapter four contains the results of applying these heuristics to a sample problem, an Air Traffic Control (ATC) simulation. Chapter five records the validation method and results for the heuristics. The thesis concludes with a chapter in which conclusions are drawn and recommendations for further work are given.

II. Literature Survey

2.1 Introduction

Real-time systems normally exhibit a high degree of concurrency[Gomaa 1989b]. Consequently, a real-time design method should provide guidance for designers to help identify and implement concurrency. This survey examines how current real-time design methods assist designers in making concurrency decisions. Five extensions to Yourdon's Structured Analysis are considered first: Structured Development for Real-time Systems, Design Approach for Real-time Systems (DARTS), Layered Virtual Machine/Object-Oriented Design (LVM/OOD), Ada-based Design Approach for Real-time systems (ADARTS), and Process Abstraction Method for Embedded Large Applications (PAMELA). Jackson System Development (JSD) is then examined, along with a related method, Entity-Life Modeling. The chapter concludes with a brief discussion of Object-Oriented Design.

2.2 Structured Development for Real-time Systems

Structured Design[Yourdon and Constantine 1979], a method of classical design in which the system under consideration is structured into transforms and data flows, has been popular with business data processing systems for a number of years. The design approach, though, addresses data manipulation mainly, and only peripherally touches on control and concurrency features characteristic of real-time and embedded systems[Ward and Mellor 1985]. Ward and Mellor introduced "... control considerations, through the use of state transition diagrams. A control transformation represents the execution of a state transition diagram"[Gomaa 1989b:9]. Thus, a state transition diagram may be associated with each control transform to represent the dynamic behavior of the system[Ward 1986:201].

The control and data transformations are graphically represented by a Data Flow Diagram (DFD). After the DFD is developed, the transforms are allocated to processors and the transforms on each processor are allocated to concurrent tasks. Structured Design is then iteratively applied to design the tasks[Gomaa 1989b:10]. Structured Design provides a method by which individual tasks can be designed, but little help is given in structuring the system into concurrent tasks. Gomaa notes that "...Structured Design is a program design method leading primarily to functional modules and does not address the issues of structuring a system into concurrent tasks"[Gomaa 1989b:11].

2.3 Design Approach for Real-time Systems (DARTS)

The DARTS method provides an approach for structuring a real-time system into concurrent tasks[Gomaa 1984]. Using a DFD, which is developed using Structured Design techniques, concurrency is identified by considering the nature of the transforms and grouping them according to the following task structuring criteria[Gomaa 1984:940].

- Dependency on Input/Output. A transform associated with an I/O device should be a separate task.
- Time-critical Functions. A transform which executes under tight time constraints needs to run at a high priority and should be a separate task.
- Computational Requirements. A transform which requires extensive calculation needs to run at a low priority (perhaps in background) and should be a separate task.
- Functional Cohesion. Two or more transforms that perform similar functions can be grouped into a single task.
- Temporal Cohesion. Two or more transforms that perform functions during the same time period can be grouped into a single task.
- Periodic Execution. Transforms that execute at regular intervals can be grouped into a single task.

Once the tasks are identified, the task interfaces are designed and the tasks are themselves designed, again using Structured Design techniques.

2.4 Layered Virtual Machine/Object-Oriented Design (LVM/OOD)

LVM/OOD is a data-flow based design method developed by Nielsen and Shumate[Nielsen and Shumate 1988].

> "The concept of LVM is used to create a top layer as a set of communicating sequential processes. Each process is a virtual machine that executes in parallel with the other processes (virtual machines). We combine the concepts of LVM and OOD (LVM/OOD) to decompose each process into a hierarchy of virtual machines (Ada subprograms) and objects (Ada packages, types, and operations on objects of the type)"[Nielsen and Shumate:33].

The method consists of ten steps[Nielsen and Shumate:211f]. The first three steps are concerned with producing a Structured Design, *i.e.*, the data flow diagram, data dictionary, etc. In the fourth step, the step in which concurrency is determined, process selection rules are applied to the DFD to combine transforms into concurrent processes[Nielsen and Shumate:212]. The first six process selection rules are identical to Gomaa's task structuring criteria[Gomaa 1984:940], listed above[Nielsen and Shumate:90-91]. Two rules have been added:

- Storage Limitations. If processes are too large, they will need to be split into smaller processes.
- Data Base Functions. Transforms needing access to shared data can be grouped in a single process to provide for mutual exclusion[Nielsen and Shumate:90-91].

2.5 Ada-based Design Approach for Real-time Systems (ADARTS)

In a recent article, Gomaa modified the original DARTS method to specifically address designing real-time systems using Ada, which he calls ADARTS[Gomaa 1989a].

In ADARTS, the task structuring criteria are expanded and reorganized as follows:

1. Event Dependency Criteria. These criteria are concerned with how and when a task is activated. Included in this category are the following:
   (a) Asynchronous Device I/O Dependency. This is the same as the DARTS *Dependency on I/O.*
   (b) Periodic Event. This is the same as the DARTS *Periodic Execution.*
   (c) Periodic I/O. The task activation is periodic, but is related to some I/O device.
   (d) Contr . Function. This is a function which may be represented by a state transition diagram.
   (e) Entity Modeling. This is a task which models concurrency in the problem environment.
   (f) User Interface Dependency. Sequential operations performed by the user can be grouped into a single task.
2. Task Cohesion Criteria. These criteria provide a basis for determining which functions can be combined into tasks.
   (a) Sequential Cohesion. Fucnctions that must be carried out sequentially can be grouped into a single task.
   (b) Temporal Cohesion. This is the same as the DARTS *Temporal Cohesion.*

(c) Functional Cohesion. This is the same as the DARTS *Functional Cohesion.*

3. Task Priority Criteria. The criteria are based on the priorities of the functions.

(a) Time Critical. This is the same as the DARTS *Time Critical Functions.*

(b) Computationally Intensive. This is the same as the DARTS *Computational Requirements.*

2.6 Process Abstraction Method for Embedded Large Applications (PAMELA)

PAMELA is an Ada-based design method developed by George Cherry[Cherry 1986]. Since most information on PAMELA is proprietary, the material in this section is taken from two articles comparing PAMELA with other methods[Kelly 1987][Boyd 1987].

PAMELA is a process-oriented method, i.e., the dynamic properties of the system under consideration are given priority over the static structure. These two views of the system are represented by process modules and procedure modules, respectively. Processes have "...one or more independent threads of control(run time stack)..."[Boyd 1987:4-69] and conserve local state. Procedure modules "...have no independent thread of control, and cannot conserve local state information"[Boyd 1987:4-69].

PAMELA is actually an extension of Structured Design. The top level of abstraction in a PAMELA design "...is essentially a data flow diagram of processes ..."[Kelly 1987:241]. Boyd states, "In effect, PAMELA supports a functional (procedural) decomposition ..."[Boyd 1987:4-69].

The heuristic PAMELA provides for determining these independent threads of control is to identify asynchronous processing in the system. According to Boyd[Boyd 1987:4-69], A guiding principle is to isolate (as much as possible) those interactions which require asynchronous handling in the highest regions of a system design; this leads to processes at the higher levels of the system. Sequential processing of information takes place at lower levels of the hierarchy, effectively isolated within the decomposition of asynchronous processes.

2.7 Jackson System Development(JSD)

Jackson System Development (JSD) incorporates a design method in which the real world is modeled"in terms of entities, actions they perform or suffer, and the orderings of those actions."[Jackson 1983:23] Thus the focus is not on a step-by-step progression of functions acting upon data.

A complete description of JSD can be found in [Jackson 1983]. The following discussion is drawn from [Cameron 1986] as it provides a concise overview of the method and discusses the relevant concurrency issues.

A JSD specification consists of a network of sequential processes communicating via message passing and access to the process's local, read-only data. This specification is produced by completing three phases:

1. Modeling phase[Cameron 1986:222]. This phase is concerned primarily with identifying the events or actions occurring in that portion of the real world which is to be modeled. Each action will be associated with one or more entities or objects. These action-entity associations are then grouped and ordered, producing a set of sequential processes. Each of these processes is then referred to as a process model.

2. Network phase[Cameron 1986:228]. The network phase determines the interconnections of the process models. Processes can communicate by two means, data streams and state vectors. A data stream is basically a first in, first-out (FIFO) message queue. The state vector consists of a process's local data which is available for inspection by other processes on a read-only basis.

3. Implementation phase[Cameron 1986:233f]. In this phase each of the process models is implemented in some programming language. This is the step in which concurrency decisions are made. Theoretically, every process model can be implemented as a concurrent process. This may not be desirable, especially in a single-processor system, as significant inefficiency may result. One way to alleviate this is to convert the processes to subroutines and combine the whole program into one process. Of course, these are the two extremes; the designer decides which processes are actually implemented concurrently and which are converted into subroutines. How the designer makes these decisions is not addressed.

2.8 Entity-Life Modeling

Entity-life Modeling is a JSD-based method developed by Sanden[Sanden 1989]. While JSD identifies many concurrent processes when applied to real-time problems, the goal of Entity-life Modeling (also called Object-life Modeling[Sanden 1989]) is to implement in software only those concurrent processes which model concurrency in the problem environment. "The aim is to pattern the software structure on structures found in the problem environment and minimizing the amount of extra material introduced for the administration of the software itself"[Sanden 1990:16].

The designer accomplishes this by identifying complex behavior patterns in the problem environment. "When using the approach, the analyst/designer starts by looking for complex, yet purely sequential, behavior patterns in the problem environment. The objective is to capture as much of the problem complexity as possible in as few behavior patterns as possible, and, generally, the more complexity that can be captured in a single sequential behavior pattern, the better"[Sanden 1989:1459]. This complex behavior is defined as "the timing and ordering of operations on various objects"[Sanden 1990:17].

Each of these behavior patterns is implemented as a concurrent task. Ideally, this is the minimum necessary concurrency, but practical considerations may require additional concurrency in the solution. For example, a task may be introduced to provide for mutual exclusion in a shared data store[Sanden 1990:298].

2.9 Object-Oriented Design

According to Booch, "Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design"[Booch 1991:37]. In object-oriented decomposition, the problem environment is viewed as a set of objects and the operations suffered by those objects. Design consists of identifying the objects and operations and specifying the interaction of the objects.

Concurrency in Object-Oriented Design is determined when the operations are identified, as this is when the dynamic behavior of the object is specified[Booch 1987b:337]. An object which exhibits significant dynamic behavior is said to represent an independent thread of control, and is called active[Booch 1991:66]. Thus, the world can be viewed "... as consisting of a set of cc perative objects, some of which are active and thus serve as centers of independent activity"[Booch 1991:66]. Chapter three of this thesis expands further on Object-Oriented Design and concurrency-related issues.

2.10 Conclusion

The Object-Oriented Design paradigm provides general guidance for determining which objects are concurrent, i.e., identifying active objects. The designer does not have specific criteria to aid in this determination, nor is the possibility of multiple concurrent operations on the same object addressed.

On the other hand, a designer applying Structured Design has specific criteria to apply to a DFD to determine concurrency, through DARTS, LVM/OOD, and

ADARTS. Entity-life Modeling also provides heuristics for identifying concurrency. Kelly claims that PAMELA's support of concurrency is very strong[Kelly 1987:245].

Chapter Three of this thesis provides heuristics which can be applied to an object-oriented design to determine concurrency. The heuristics are based on the work of Gomaa (DARTS, ADARTS), Nielsen and Shumate (LVM/OOD), and Sanden (Entity-life Modeling).

III. *Heuristics for Determining Concurrency in Object-Oriented Design*

This chapter details the heuristics a designer may use to determine concurrency in an object-oriented design. A discussion of object-oriented design is presented first, followed by a description of Booch's Object-Oriented Design method. The concurrency heuristics, which are based on the work surveyed in chapter two, are then given.

3.1 Object-Oriented Design

Object-oriented design is a design approach in which the problem environment is modeled as a collection of interacting objects and classes. The object interactions are referred to as messages or operations[Booch 1991:80].

The object-oriented paradigm is based on the concepts of abstraction and information hiding[Booch 1991:38]. Pressman states

> The unique nature of object-oriented design lies in its ability to build upon three important software design concepts: abstraction, information hiding, and modularity. All design methods strive for software that exhibits these fundamental characteristics, but only OOD provides a mechanism that enables the designer to achieve all three without complexity or compromise[Pressman 1987:334].

Application of these concepts produces a hierarchical object structure, where hierarchy is defined as "...a ranking or ordering of abstractions"[Booch 1991:54]. Booch defines two sets of hierarchies, the "kind of"/"part of" hierarchies, and the using/containing hierarchies[Booch 1991:54,88]. The "kind of"/"part of" hierarchies deal with objects which are instantiations of a class of objects and objects which are component parts of another object. These concepts apply directly to object-oriented programming languages and techniques, but are not crucial at design time.

Using/containing hierarchies, however, are important for object-oriented design. The using hierarchy demonstrates the relationships among objects which require services of other objects and objects which provide services to other objects. Booch calls the former "actor" objects and the latter "server" objects; objects which both require and provide services are called "agents"[Booch 1991:89].

The containing hierarchy demonstrates the relationships between objects which "enclose" other objects and the objects "within" the enclosing objects. In other words, some objects are completely hidden within another object.

Seidewitz calls the using and containing hierarchies the seniority and composition hierarchy, respectively. He states that 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 1989:97].

Consider, for example, the air traffic control simulation whose design is presented in chapter four. An example of the composition hierarchy would be the Console object and its related sub-objects, Display and Keyboard. Console contains those two objects (Figure 3.1). The ATC object however, does not contain the Console object, *i.e.*, it is not composed of Console; ATC does, however, use the services provided by the Console object (Figure 3.2). Note that although the ATC and Console objects are at the same level of abstraction, the ATC is a higher level virtual machine layer than Console, since ATC requires operations of Console, but Console requires no operations of ATC.

When an object is just one of several instantiations of the same type of object, the object type is referred to as a class of objects, which is "... a set of objects that share a common structure and a common behavior"[Booch 1991:93].

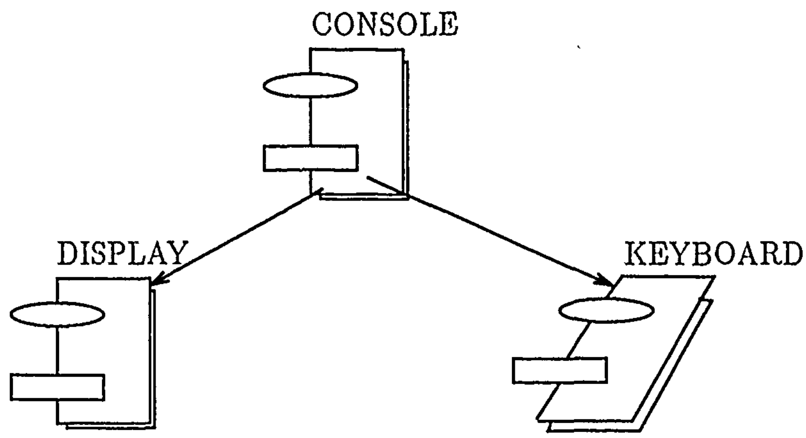


Figure 3.1. Composition Hierarchy for Console

In object-oriented programming languages, such as Smalltalk and C++, the class concept is related to the concept of inheritance. Inheritance is a relationship among objects where one object or class shares the structure of one or more objects or classes, *i.e.*, an object or class "inherits" the structure or behavior of another object or class. Ada does not directly support inheritance, so the class concept is not as important as in other languages. Consequently, this thesis does not consider inheritance in the design of systems.

The class concept is still useful in determining concurrency since a single concurrent object produces a different design and implementation from a concurrent class, which may have multiple concurrent instantiations. Also, identifying classes of objects is important from a reusability standpoint. If an object is a member of a previously implemented class, then that object need not be reimplemented.

3.2 Booch's Method

The object-oriented design method used in this thesis is Booch's Object-Oriented Design as presented in [Booch 1991:187-196]. The steps of the method are:

- Identify the classes and objects at a given level of abstraction.

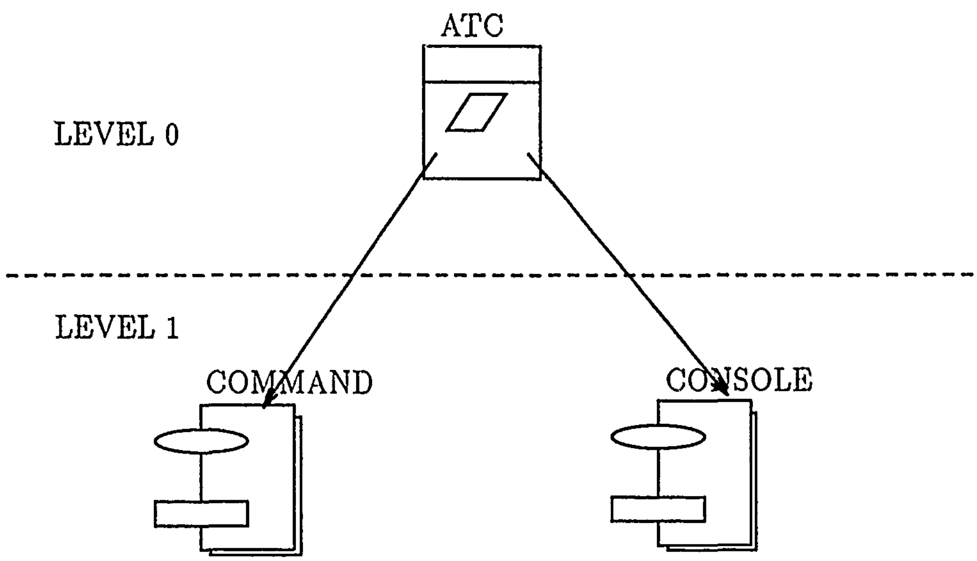


Figure 3.2. Seniority Hierarchy for ATC

- Identify the semantics of the classes and objects.
- Identify the relationships among the classes and objects.
- Implement the classes and objects.

The application of this method is not just a matter of mechanically performing the steps in sequence. Booch notes: "this is an incremental process: the identification of new classes and objects usually causes us to refine and improve upon the semantics of and relationships among existing classes and objects. It is also an iterative process: implementing classes and objects often leads us to the discovery or invention of new classes and objects whose presence simplifies and generalizes our designs"[Booch 1991:190].

Normally, software design is preceded by an analysis step in which the problem statement is analyzed and a requirements specification is produced[Fairley 1985:38]. Booch's method does not preclude this approach; object-oriented analysis is considered an "... ideal front end to object-oriented design"[Booch 1991:141]. However, when applying object-oriented analysis, the distinction between analysis and design is somewhat artificial and difficult to maintain[Sanden 1990:32]. Therefore, no attempt is made in this thesis to separate the two.

The analysis/design in this thesis will be accomplished using an Object-Class Specification, which is a combination of graphical and textual representation of and object or class. It shows an object's or class's components, operations, static and dynamic relationships, and other information pertinent to design and implementation. An example can be found in Figure 3.3.

3.2.1 Identify the classes and objects at a given level of abstraction. This step consists of "...two activities: the discovery of the key abstractions in the problem space (the significant classes and objects) and the invention of the important mechanisms that provide the behavior required of objects that work together to achieve some function"[Booch 1991:191]. Generally, the key abstractions are the classes and objects which correspond to the vocabulary of the problem domain[Booch 1991:123]. The mechanisms are structures through which the objects interact with one another to provide the required behavior[Booch 1991:123].

3.2.2 Identify the semantics of these classes and objects. This step "...involves one basic activity, that of establishing the meanings of the classes and objects identified from the previous step"[Booch 1991:192]. This entails determining what can be done to an object, and what things the object can do to other objects. According to Booch, "One useful technique to guide these activities involves writing a script for each object, which defines its life cycle from creation to destruction, including its characteristic behaviors"[Booch 1991:192].

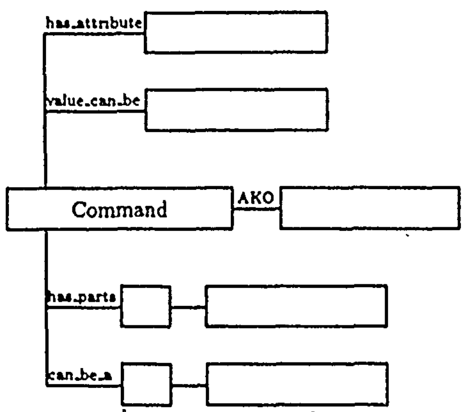
3.2.3 Identify the relationships among these classes and objects. This step establishes the interaction of things within the system. This is accomplished by performing two related activities: "First we must discover patterns: patterns among classes, which causes us to reorganize and simplify the system's class structure, and

CLASS SPECIFICATION

Class Name: Command

Description: Provides the command abstraction for the ATC simulation.

Static Relationships



Dynamic Relationships

Suffered Operations

	Descriptive Name
Selectors:	Get_ID
	Is_Status
	Is_Termination
	Is_COMMAND
Constructors:	Create_Command

Required Operations

	Name	Applied to
Selectors:		
Constructors:		

Exceptions

Name	Raised by
Invalid_Cmd	Command
Invalid_Aircraft	Command

QA

Initial:

Figure 3.3. Example Object-Class Specification

patterns among cooperative collections of objects which lead us to generalize the mechanisms already embodied in the design. ... Second, we must make visibility decisions: how do classes see one another, how do objects see one another, and, equally important, what classes and objects should not see one another"[Booch 1991:193].

3.2.4 Implement these classes and objects This step requires the designer to make "... design decisions concerning the representation of the classes and objects we have invented, and allocating classes and objects to modules, and programs to processors"[Booch 1991:195]. The result of this step is a complete system design. However, new abstractions and mechanisms are frequently discovered during this step. These abstractions usually belong to a lower level of abstraction, and they are designed by repeating the object-oriented design process. When no lower level abstractions or mechanisms remain to be designed, the design at higher levels can be completed, at which time the design is complete[Booch 1991:195].

3.3 Heuristics for determining concurrency.

As noted in chapter two, Booch's Object-Oriented Design method is weak in the area of determining concurrency. Kelly states that the designer is given "very little guidance on concurrent design ..."[Kelly 1987:245]. The purpose of this thesis is to provide this guidance.

Following are four heuristics which designers may use in determining concurrency in object-oriented designs. They are based on the heuristics used in the DARTS, LVM/OOD, ADARTS, and Entity-life Modeling methods. These methods and their heuristics are summarized in chapter two.

3.3.1 Problem-space concurrency. **An object which models concurrency in the problem environment should be implemented as a task.**

According to Fairley, "the software engineer creates models of physical situations in software"[Fairley 1985:3]. One of the strengths of the object-oriented paradigm is in allowing a designer to create these models of physical situations, *i.e.*, to directly model the problem-space, thus minimizing the "intellectual distance" between the model and the system being modeled[Fairley 1985:3]. Accordingly, if concurrency exists in the problem-domain, it should be modeled in the design.

Concurrency in the problem-domain can be determined by identifying behavior patterns, or sequences of events, in which the objects participate. These sequences of events are related to the timing and ordering of the operations on the problem-space objects. Sanden states, "while an object does not control the timing and ordering of the operations it suffers, the timing and ordering of operations on various objects can be described as behavior patterns in the reality"[Sanden 1990:17].

An object may exhibit a single pattern of behavior, multiple sequential patterns, or none at all. Note these patterns of behavior specify the timing and ordering of operations required of the object, *i.e.*, the behavior pattern expresses how an object uses other objects. Thus, objects with no suffered operations (an actor object in Booch's terminology[Booch 1987a:613]), or one that has suffered operations, but requires operations of other objects (an agent object in Booch's terminology[Booch 1987a:613]) are good candidates for problem-space concurrency. On the other hand, an object with no required operations (a server object in Booch's terminology[Booch 1987a:615]) will likely not exhibit problem-space concurrency, although it may or may not exhibit concurrency as determined by the remaining heuristics.

In general, no priority exists among the heuristics, *i.e.*, which heuristic is used to determine concurrency is not important as long as the necessary concurrency is identified. In a sense, however, problem-space concurrency is the most important of the heuristics, as it is really an extension of the object-oriented philosophy; problem-space concurrency goes to the heart of the modeling process. By examining the behavior of the objects specifically to identify concurrency, the designer not only determines problem-space concurrency, but gains a better understanding of the design overall.

An example of the application of this heuristic is in the air traffic control simulation (ATC) described in chapter four. The object representing the ATC simulation exhibits multiple behavior patterns, *i.e.*, more than one sequence of events. The object needs to update the position of the aircraft in the control space at periodic intervals, while concurrently polling the keyboard for asynchronously entered commands. The keyboard task cannot be placed within the update control space without forcing the keyboard task to be periodic. Thus the two sequential behavior patterns require two tasks to maintain the asynchronous nature of the polling routine.

A special case of problem space concurrency is when an actual hardware device is modeled as an object. In general, real-time systems interface to one or more hardware devices; these devices will likely be modeled as objects, with the operations corresponding to the input/output of the device.

Devices whose primary function is I/O, *e.g.*, printers and keyboards, have varying speeds and will generally have to be implemented as separate tasks to accommodate the differences in speed. In particular, if the I/O device must interface with another task, the only way to decouple the two is to make them separate tasks.

Those devices which perform other functions, such as sensors or control devices, may or may not be implemented as tasks, depending on how they interface with the rest of the system. If the device provides information to which the system must respond, but provides the information asynchronously, then a task should monitor the device rather than having the system poll the device.

To illustrate this point, consider a system that contains a temperature sensor. When the temperature exceeds some limit, the system must take action to reduce the temperature. If the system polls the sensor, system resources must be used to monitor a condition which may have a low probability of occurring, plus the polling interval may be too long for the system to provide adequate response. A better solution might be to have a task interface with the sensor, continuously monitoring the temperature. When the task detects an out-of-tolerance condition, it alerts the system. In this way, the system need not dedicate resources to a polling scheme, and the time in which the system is alerted will not be tied to a polling interval.

3.3.2 Time constraints. **An object whose behavior or operations are constrained by time requirements should be a task.**

One of the characteristics of real-time systems is the requirement for the system to meet time constraints. These time constraints can be periodic, *e.g.*, a certain operation needs to be performed at periodic intervals, or responsive, as when the system must respond to an event within a certain amount of time.

In an object-oriented design, an object's behavior may be constrained by time requirements, or one or more of its operations may be so constrained. When the designer encounters such objects or operations, the objects or operations should probably be implemented as tasks.

An example of a periodic constraint could be a temperature monitor which must sample a temperature sensor at regular intervals; this would most likely need to be a separate task. An example of a response constraint is an interrupt handler which must service an interrupt within a certain time. For example, in an elevator control system, an interrupt may be generated when an elevator arrives at a floor, and the system may have a short period in which to decide to stop the elevator at the floor or let it continue.

3.3.3 Computational requirements. **An object whose behavior or operations require substantial computational resources should be a task.**

Computational requirements may dictate that some operations or objects be implemented concurrently, probably as low priority, background tasks. Occasionally, an operation requires substantial computational resources. For example, in a satellite communication system, the satellite object may have an operation called Calculate Satellite Coordinates. To do this in real time requires the integration of a ninth-order polynomial. Depending on the resources available, this could be quite time consuming and processor intensive. This operation should be a separate task.

3.3.4 Solution-space objects. **An object introduced in the software solution to protect a shared data store, decouple two interacting tasks, or synchronize the behavior of two or more objects should be a task.**

Some solution-space objects may need to be implemented concurrently. This is a general heuristic which considers concurrency in software mechanisms belonging to the solution space.

One such mechanism is a shared data store modeled by an object. The only way to guarantee mutual exclusion in Ada is to use a task with a selective wait. In this case, the concurrency is forced by the language conventions.

Another mechanism is the use of intermediary tasks to control the coupling between two other tasks. In a simple Ada rendezvous the tasks are tightly coupled; neither task can continue until the rendezvous is complete. Oftentimes, especially when time constraints prevent a task from waiting, another task can be introduced to allow the other two to proceed. In [Nielsen and Shumate 1989:16[illegible]], several types of intermediaries are described, combinations of which allow the designer to achieve a range of coupling, from very loose coupling to very tight coupling. As a caveat, however, the looser the coupling, the greater the number of intermediaries needed; this could generate significant tasking overhead, particularly in a single-processor environment.

A third mechanism might be the synchronization of two objects or their operations. In this case, the synchronizing objects or operations need to be tasks, with a simple rendezvous accomplishing the synchronization.

3.4 Conclusion

The designer of object-oriented, real-time software systems has little more than general guidance for determining which objects and operations to make concurrent. This chapter provided four heuristics which designers can use to make concurrency decisions.

The next chapter provides an example of an object-oriented, real-time system design, an air traffic control simulation. The concurrency heuristics are applied to the design to determine the concurrency in the system.

IV. Application of Concurrency Heuristics

In this chapter, the heuristics for determining concurrency presented in the previous chapter are demonstrated by applying them to a sample problem.

4.1 Design Problem Description

The concurrency heuristics will be applied to the design of an air traffic control simulation, whose description appeared in *Creative Computing*, c.1980. For brevity, the system will be referred to as ATC. Following is a condensed statement of the problem:

> Air Traffic Control is a simulation which allows the user to play the part of an air traffic controller in charge of a 15x25 mile area from ground level to 9000 feet. In the area are 10 entry/exit fixes, 2 airports , and 2 navaids. During the simulation, 26 aircraft will become active, and it is the responsibility of the controller to safely direct these aircraft through his airspace.
>
> The controller communicates to the aircraft via the scope, issuing commands and status requests, receiving replies and reports, and noting the position of the aircraft on the map of the control space. The controller issues commands to change heading or altitude, to hold at a navaid, or clear for approach or landing. Each aircraft has a certain amount of fuel left, so the controller must see to it that the aircraft is dispositioned prior to fuel exhaustion. Also, the minimum separation rules must be followed, which state that no two aircraft may pass within three miles of each other at 1000 feet or less separation. The aircraft must enter and/or exit via one of the ten fixes. If an aircraft attempts to exit through a non-exit fix, a boundary error is generated. The controller may request a status report on each aircraft, which will display all information on the aircraft, including, fuel level, which is measured in minutes.
>
> The aircraft can be one of two types, a jet or a prop. The jets travel at 4 miles per minute, while the props travel at 2 miles per minute. This means the screen must updated every 15 seconds for a jet's course to be followed accross the screen.

The controller dispositions aircraft by giving commands which enable the aircraft to take off, land, hold at a navaid, assume a landing approach, turn, or change altitude. Take off is accomplished by ordering the aircraft to assume a certain altitude; there is no 'take off' command as such. Each of the airports has restrictions on heading for takeoff; these restrictions must be observed. Turns and altitude changes are effectively instantaneous, i.e., they are accomplished at the next mile marker. To land, the aircraft must be cleared for landing through the navigational beacon (navaid) assigned to the airport. Since there are two airports, there are two navaids. To land, the controller places the aircraft on a heading for a navaid and issues a clearance for approach command. Once the aircraft reaches the beacon, it automatically assumes the correct heading for the airport. The controller then issues a clearance to land command, and when the aircraft reaches the airport it lands (disappears from the screen). If the controller issues a hold command, the aircraft remains at the navaid until released.

The player initially specifies the length of the simulation, which may be between 16 and 99 minutes. The same number of aircraft will appear for each run, so the shorter the simulation, the more challenging. In any session, the last 15 minutes will be free of new aircraft. The simulation terminates when all aircraft have been successfully dispositioned, the timer runs out, the player requests termination, or one of three error conditions occurs:

- conflict error - separation rules were violated
- fuel exhaustion
- boundary error - the aircraft attempt to leave the control space via an unauthorized point.

Figure 4.1 contains the screen layout for the ATC simulation. The * symbol represents a navigational aid, the % and # are airports, and the numerals are entry-exit fixes. The aircraft are represented by an upper case letter followed by a number. The letter is the aircraft identifier and the number is the altitude of the aircraft in thousands of feet; *e.g.*, 'A4' indicates aircraft 'A' is at 4000 feet.

ATC commands consist of either three character directives or one character status requests. To request a status on a particular aircraft, a single character reprsenting the aircraft ID is entered. Table 4.1 contains a summary of the directive commands.

	A	L	R
0	clear to land	hold at navaid	continue straight ahead
1	ascend/descend to 1000'	turn left 45	turn right 45
2	ascend/descend to 2000'	turn left 90	turn right 90
3	ascend/descend to 3000'	turn left 135	turn right 135
4	ascend/descend to 4000'	turn left 180	turn right 180
5	ascend/descend to 5000'	clear for # approach	clear for % approach

Table 4.1. ATC Commands

```
. . . . . 1 . . . . 2 . . . . . . . . 3 . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
4 . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . % . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
0 . . . . * . . . . # . . . . . . . . * . . . . . 9
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . 8 . . . . . 5 . . . . . . . 6 . . . . . 7
                                               >60<
```

Figure 4.1. Airspace Display

4.2 Top Level of Abstraction

A cursory reading of the problem statement suggests several key abstractions: aircraft, airspace, display, commands, messages, *etc.* The initial focus of the design is determining which of these key abstractions belong at the top level of abstraction. This is admittedly a matter of designer judgement, but, in general, the top level should contain a minimal set of objects and classes, while still encompassing the entire system.

In some cases, the top level of abstraction may consist of a single object, such as

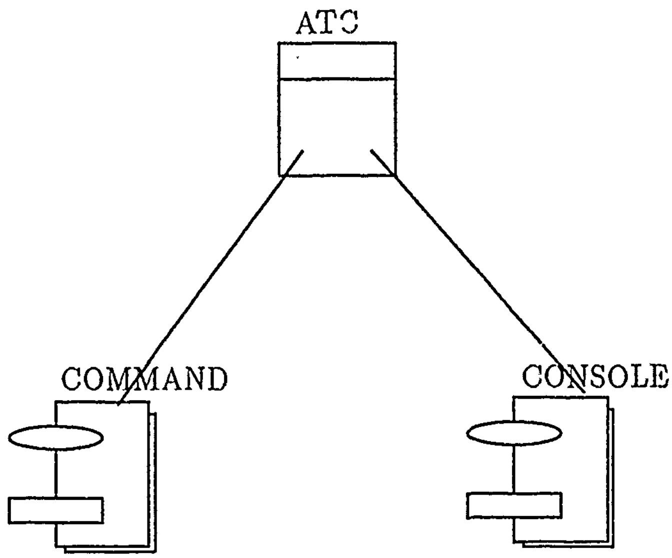


Figure 4.2. Initial Top Level Object Diagram

in Booch's example of a home heating system[Booch 1991:222-280]. In this instance, the top level of abstraction is the object *theHomeHeatingSystem.* It could be argued that the ATC system is similar, so the top level would contain only an ATC object. However, this is not a very useful structure, since it doesn't really say much about the ATC system or provide much guidance on what the next step may be. So for this design, the top level will contain more than one object.

4.2.1 Identify the classes and objects. As Figure 4.2 illustrates, the top level of abstraction consists of an ATC object, a console object, and a command class. This particular breakdown was chosen because the problem statement indicated two major activities of the system: periodic updating of the display screen to represent aircraft movement in the airspace, and responding to commands entered by the controller.

Figure 4.2 captures the essence of this activity: the ATC object does something with commands and does something with the operator's console. The lines connecting objects are at this time undirected; the arrows will be added when the relationships are established.

Notice that since there is a single instance of both the ATC abstraction and the console abstraction, they are specified as objects. There could, and most likely will, be many instances of the command abstraction, so it is specified as a class. This distinction is not reflected in the figures.

4.2.2 Identify the semantics of the classes and objects. This step involves specifying the behavior of the objects and classes at the current level of abstraction.

4.2.2.1 The command class. The command class is rather straightforward. It exports objects of type `Command` (whose representation is as yet unspecified) and two kinds of operations. The `Create_Command` operation accepts a character string and returns the command corresponding to the string. The other kind of operation, a set of selectors, accepts commands as input and then returns true if the command corresponds to that selector, and false otherwise. For example, if a *turn* command is passed to `Is_Turn`, then true will be returned, but if *change altitude* is passed to the same operation, false will be returned. Thus, the command class exhibits no dynamic behavior and can be implemented as a set of rather simple functions. The object-class specification for the Command class is shown in Figure 4.3.

4.2.2.2 The console object. Since the console is an object and not a class of objects, it does not export a type; it does, however, export operations on the console object.

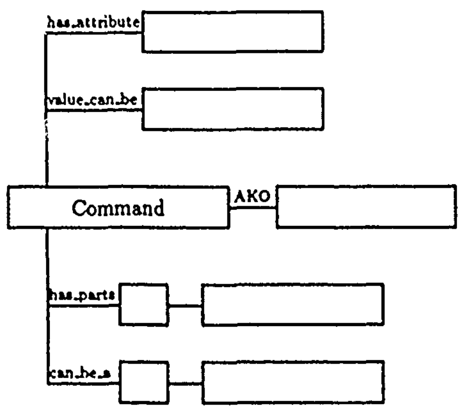
As with the command class, the console object does not exhibit significant dynamic behavior over time. This does not mean, however, that there is no concurrency withing the console object. At this level, the console displays messages and retrieves input from the user. Lower levels of abstraction may reveal concurrency which is not visible from the higher levels.

CLASS SPECIFICATION

Class Name: Command

Description: Provides the command abstraction for the ATC simulation.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Selectors: Get_ID

Is_Status

Is_Termination

Is_COMMAND

Constructors: Create_Command

Required Operations

Name Applied to

Selectors:

Constructors:

Exceptions

Name	Raised by
Invalid_Cmd	Command
Invalid_Aircraft	Command

QA

Initial:

Figure 4.3. Command Class Object-Class Specification

The following types of input/output from the console can be identified from the problem statement:

- Output the time remaining
- Output a map
- Output a preview message
- Output the string "Roger"
- Output the input string
- Input a string

Whether these operations can be combined into a smaller set or not cannot be determined at this time. The object-class specification for the console object is contained in Figure 4.4.

4.2.2.3 The ATC object. As with the console object, the ATC object exports no type, but neither does ATC export any operations. Since it is the very top level of the system, no object can call it, unless the user entering a "run" command via the operating system is considered an operation[Seidewitz 1989:99].

Since the ATC exports neither type nor operation, it must require operations from other objects (else it would not be much of an object). Thus to determine the behavior of the object entails identifying the time ordering and frequency of the required operations, and the threads of control. Even at this high level, the concurrency heuristics outlined in chapter three can be applied; however, any concurrency discovered here should be considered "candidate" concurrency, as further refinement could feasibly push the concurrency further down the hierarchy.

To elaborate the behavior of ATC, the life of the object will be modeled, as recommended by Booch[Booch 1991:192]. Since the user selects how long the simulation is to run, this information will have to be retrieved. In addition, the map will have to be initially drawn. These two items make up the initialization of the

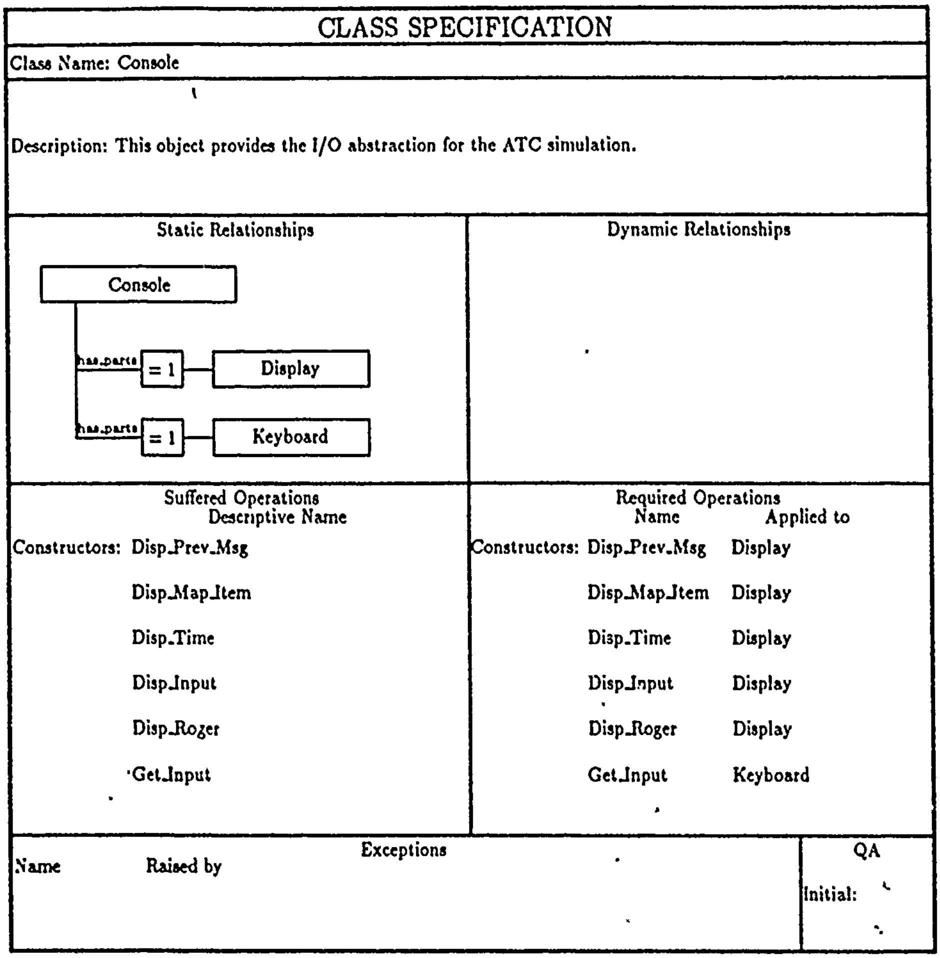


Figure 4.4. Console Object-Class Specification

system; once the initialization is complete, at least two independent threads of control are suggested by the problem statement. One thread handles the input of commands from the user and the execution of these commands; this is an asynchronous thread since the user can enter commands at any time. Another thread is a periodic update of the aircraft position in the airspace and the subsequent display of the updated map on the console. When either of these threads terminates, the simulation ends. No special clean-up operations are required other than displaying an appropriate termination message. The script of the ATC object is shown in Figure 2.5.

Applying the concurrency heuristics to these threads of control yields two tasks. The periodic update of the display fits the time constraint heuristic and thus should

```
                 Get Simulation Length
                   Draw Initial Map

loop                                  loop

  Get User Input                        Delay 15 Seconds

  If Termination Request Then           Get Airspace Updates

     Terminate Simulation               Display Airspace Updates

  End If                              end loop

  Create Command

  Process Command

end loop

                     Clean Up
```

Figure 4.5. Script for the ATC Object

be a separate task. The processing of commands is an asynchronous behavior pattern, indicating it should be contained in a concurrent task. The command processing function cannot be embedded within the periodic update task without forcing the command processing task to be periodic as well. Therefore, the command processing function should be a separate task under the problem-space heuristic. The object-class specification for the ATC object is in Figure 2.6

4.2.3 Identify the relationships among the classes and objects. This step identifies patterns of object interaction and visibility between objects and classes. The behavior specification from the previous step is used to determine the relationships. Examining the command class and the console object reveals no interaction, at least at this level of abstraction, with each other or with the ATC object. Thus, these two objects need see no other objects.

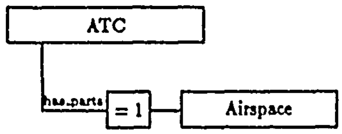
The ATC object, on the other hand, needs to have both command and console

CLASS SPECIFICATION

Class Name: ATC

Description: This is the main object of the simulation. It controls the interaction of the other objects.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Selectors:

Constructors:

Required Operations

	Name	Applied to
Selectors:	Get_ID	Command
	Is_Status	Command
	Is_Termination	Command
	Is_COMMAND	Command
Constructors:	Disp_Pre_Mge	Console
	Disp_Map_Item	Console
	Disp_Time	Console
	Disp_Input	Console
	Disp_Roger	Console
	Get_Input	Console
	Create_Cmd	Command

Exceptions

Name	Raised by
Time_Expired	ATC
Invalid_Cmd	Command
Invalid_Acft	Command
Fuel_Exhausted	Airspace
Conflict_Error	Airspace
Bdary_Error	Airspace

QA

Initial:

Figure 4.6. ATC Object-Class Specification

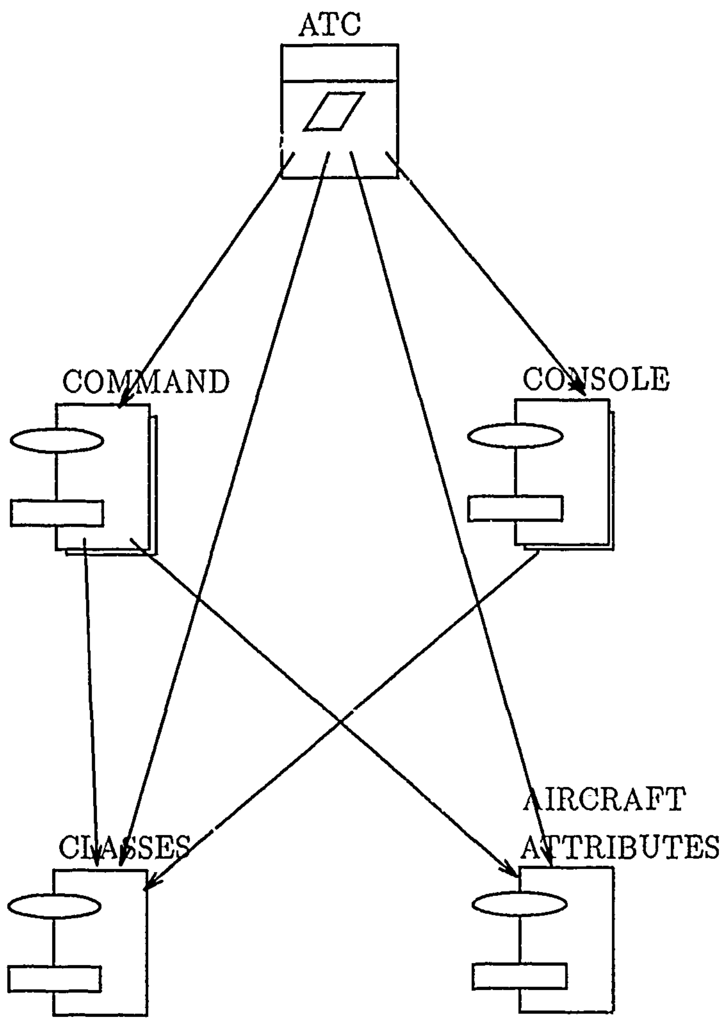


Figure 4.7. Final Top-level Object Diagram

visible. This is apparent from the script of the ATC behavior in Figure 4.5. Both console operations (get input string) and command operations (convert string to a command) are used. Other operations on objects not yet elaborated are also referenced, but they belong to lower levels of abstraction. The final top-level object diagram, with visibility indicated by directed lines, is in Figure 4.7. The update airspace task is indicated by the parallelogram within the ATC object.

4.2.4 Implement the classes and objects. It is at this time that representation decisions are made and the operations on each object are implemented. However, the implementation cannot be completed until all lower level abstractions are likewise implemented. This is a result of the iterative nature of object-oriented design: the same process is applied many times at different abstraction levels.

In implementing the ATC, console, and command objects, unspecified abstractions are encountered, necessitating suspension of the implementation while these new abstractions are designed. Once implemented, the suspended implemenations may resume.

Subsequent sections in this chapter detail this refinement for the ATC and console objects, but an example, which allows the command class to be completed, is given here for clarity.

The problem statement and ATC script both refer to input from the user which commands the aircraft or request status. As yet, there is not an input string object, and this needs to be specified before the command class can be completed. The input string class is deemed to be a component object of the console object, so this class appears beneath the console in the composition hierarchy. However, the command class and the ATC object need visibility into the input string object. Thus in the object diagram, shown in Figure 4.7, directed lines are drawn from ATC and command to input string. The command class may now be completed.

4.3 Refinement of the console object.

In the remainder of this chapter the discussion will be more informal than previous sections. The focus will be on concurrency in the ATC; the steps in the Object-Oriented Design method will be followed, but the design will not be documented to the level of detail of the previous section.

As stated previously, the console object has five output operations and one input operation. These operations could be implemented at this time, given a suitable display interface package, excepting that the problem statement places some restrictions on the format of the display. In effect, the display is divided into five distinct areas:

- Time area
- Preview area
- Map area
- Input area
- Response area

These areas can be treated as component objects of the console. They will consist of a location on the console and two operations: display message and clear area.

Since the output has been divided into five separate objects, the input operation will become an object to maintain separation of concerns. In light of this, it is appropriate to form two component objects of console: display and keyboard. The display areas mentioned earlier have now become component objects of the display object. This arrangement is shown in Figure 4.8.

At this level the concurrency heuristics can be applied to determine the keyboard object to be concurrent. It is a hardware device being modeled in software, and so should be implemented concurrently.

The implementation of the display areas must now be considered and a problem immediately poses itself. Should each area object write directly to the display or should each call a screen object which alone accesses the physical device? In the interests of encapsulation, the screen object option is chosen, although the resulting object diagram looks rather odd with the display being split into five component objects and then all five running back into one screen object.

The next concern is with concurrency. Since the screen is a hardware device being modeled in software, the screen object is concurrent.

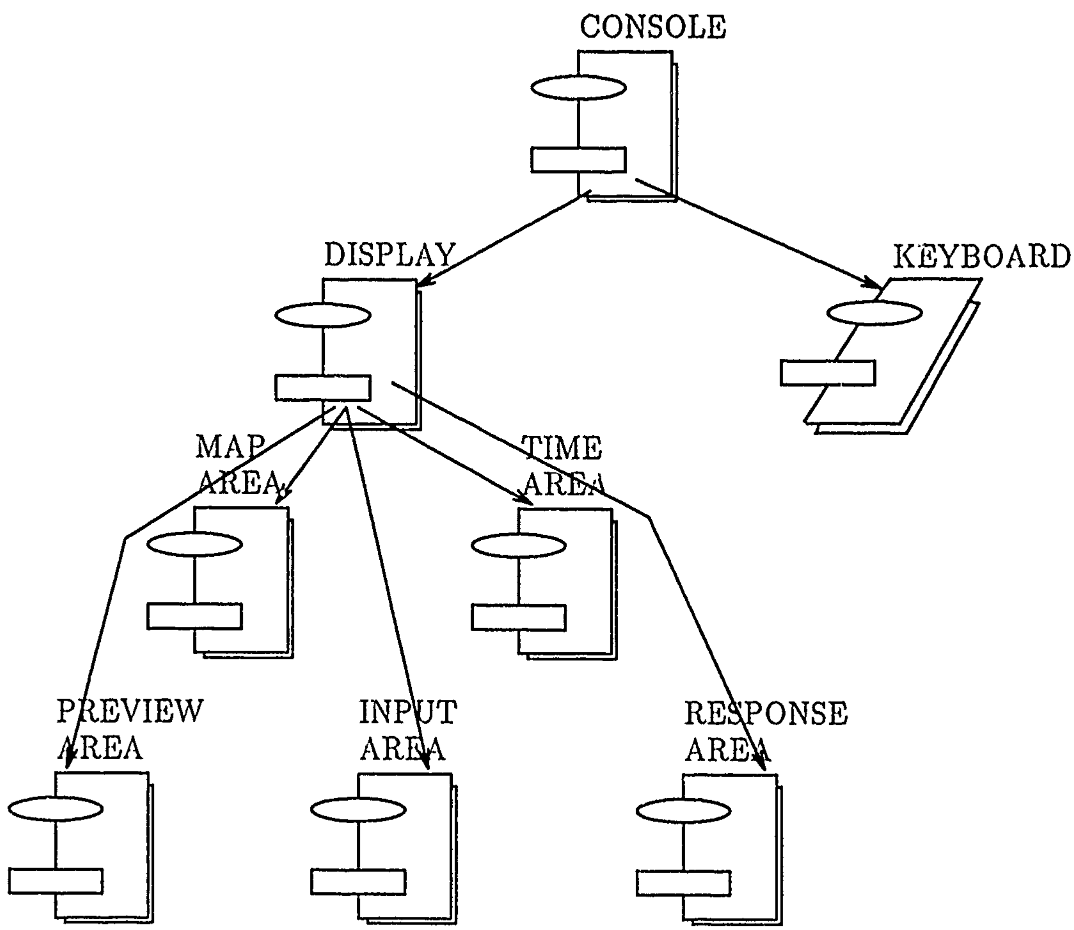


Figure 4.8. Console Object Refinement

The final console object diagram appears in Figure 4.9.

4.4 Refinement of the ATC object.

The top level design has considered mainly the user interface, *i.e.*, handling of input, display of output, and processing of commands. The Command class has been designed and the Console has been refined; this leaves the ATC object which is the heart of the simulation.

Examining the script from the ATC object in Figure 4.5 reveals references to the Airspace object. Thus, initially, the ATC object includes the Airspace object, as shown in Figure 4.10. The airspace is basically a 3-dimensional area through which aircraft fly and containing certain landmarks (navigational beacons, airports, entry/exit fixes). The operations on the Airspace object include setting and getting the position of landmarks, getting the position of a particular aircraft, and iterating through all the aircraft in the airspace to get their positions. It is possible to cast this last operation, iterating through all aircraft, as a task; however, by the first heuristic, the Airspace object has no discernible behavior pattern in the problem-space. It is rather a passive entity through which aircraft fly. So the decision at this point is to not make the Airspace or any of its operations concurrent.

One practical matter that arises is the communication between the ATC object and the Airspace object. The script of the ATC object indicates ATC "retrieves airspace updates" and then displays them. This implies the need for a 'solution-space' object, a list of aircraft updates which the Airspace returns to the ATC object. As this object is written only by Airspace and read only by ATC, it need not be a protected data store, and consequently should not be implemented as a task. This object, the `Update_Record_List` is shown in Figure 4.11.

The next step in refining the ATC object is to examine the component objects of the Airspace. The landmark objects are static, *i.e.*, they are initialized at the

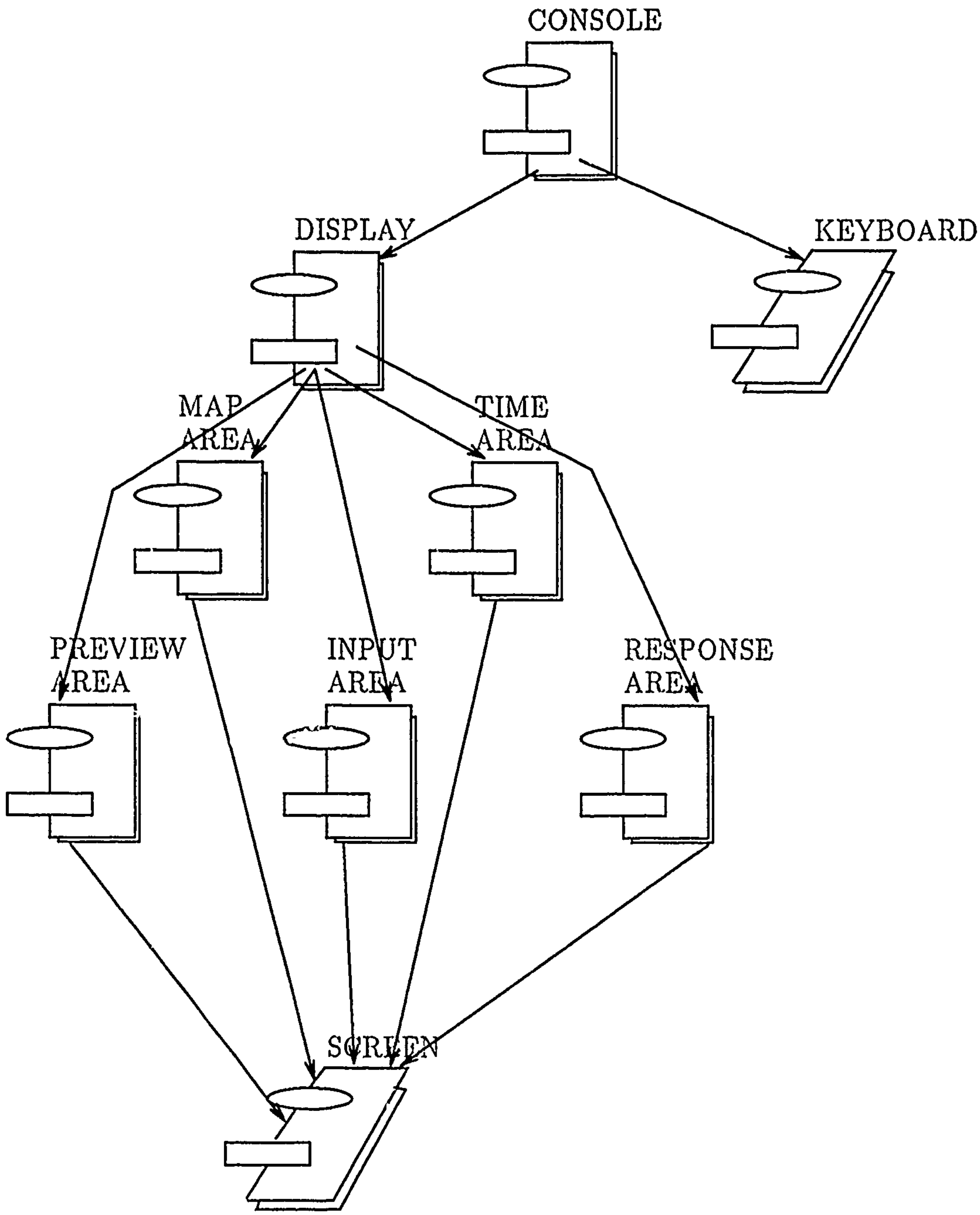


Figure 4.9. Final Console Object Refinement

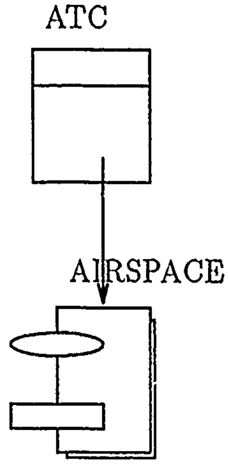


Figure 4.10. Initial ATC Object Refinement

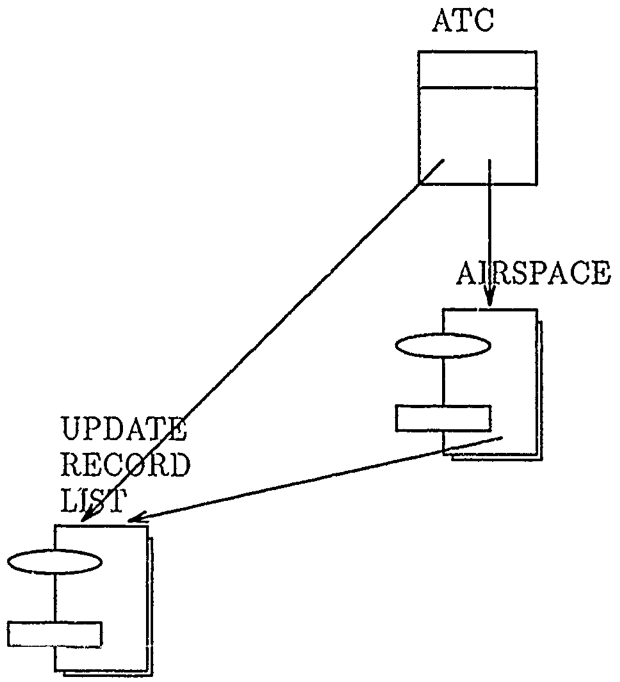


Figure 4.11. ATC with Update_Record_List

start of the simulation and never change location, so these objects are considered non-concurrent (Figure 4.12).

This leaves the Aircraft class. This class has a definable behavior pattern which changes over time. An aircraft is created, takes off or enters the airspace through a fix, makes changes to its altitude or course, and either lands or exits through a fix. Thus the Aircraft class, according to the first heuristic, exhibits problem-space concurrency and should be concurrent in the design.

An objection that may be raised at this point is that with twenty-six aircraft, this leads to massive concurrency which may not be feasible on a single processor system. This is a valid objection, but is really an implementation issue. The implementer may decide to limit the number of tasks in any way he or she chooses; the main concern for the designer is in modeling the problem-space and hence identifying concurrency.

The Aircraft class has a number of component classes and objects, but these are considered attributes of the aircraft and are thus not concurrent (Figure 4.13).

4.5 Summary

This chapter has applied the concurrency heuristics to an Air Traffic Control Simulation. Five concurrent tasks were identified: two in the ATC object, a keyboard task, a display task, and the Aircraft task.

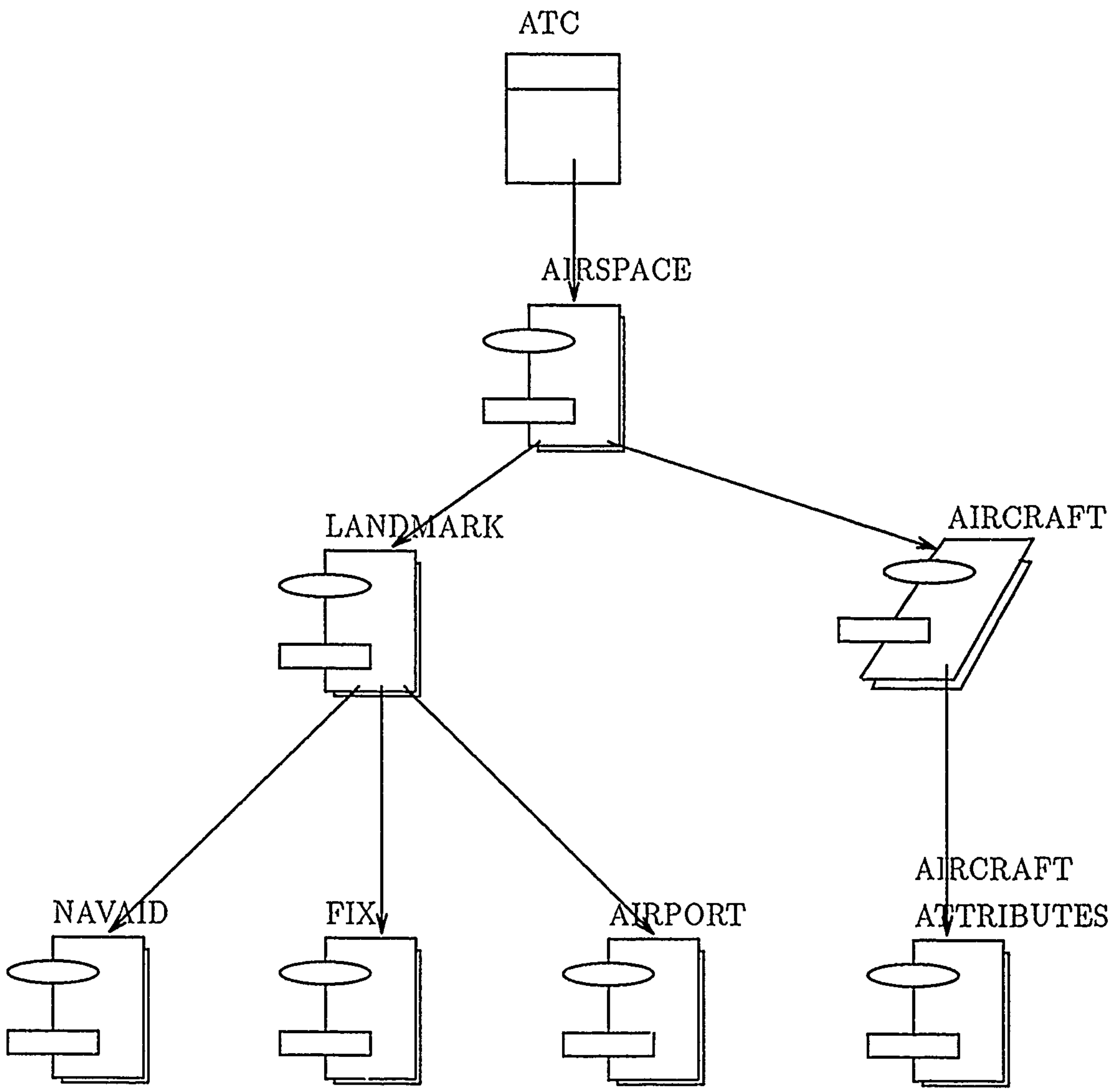


Figure 4.12. Final ATC Object Refinement

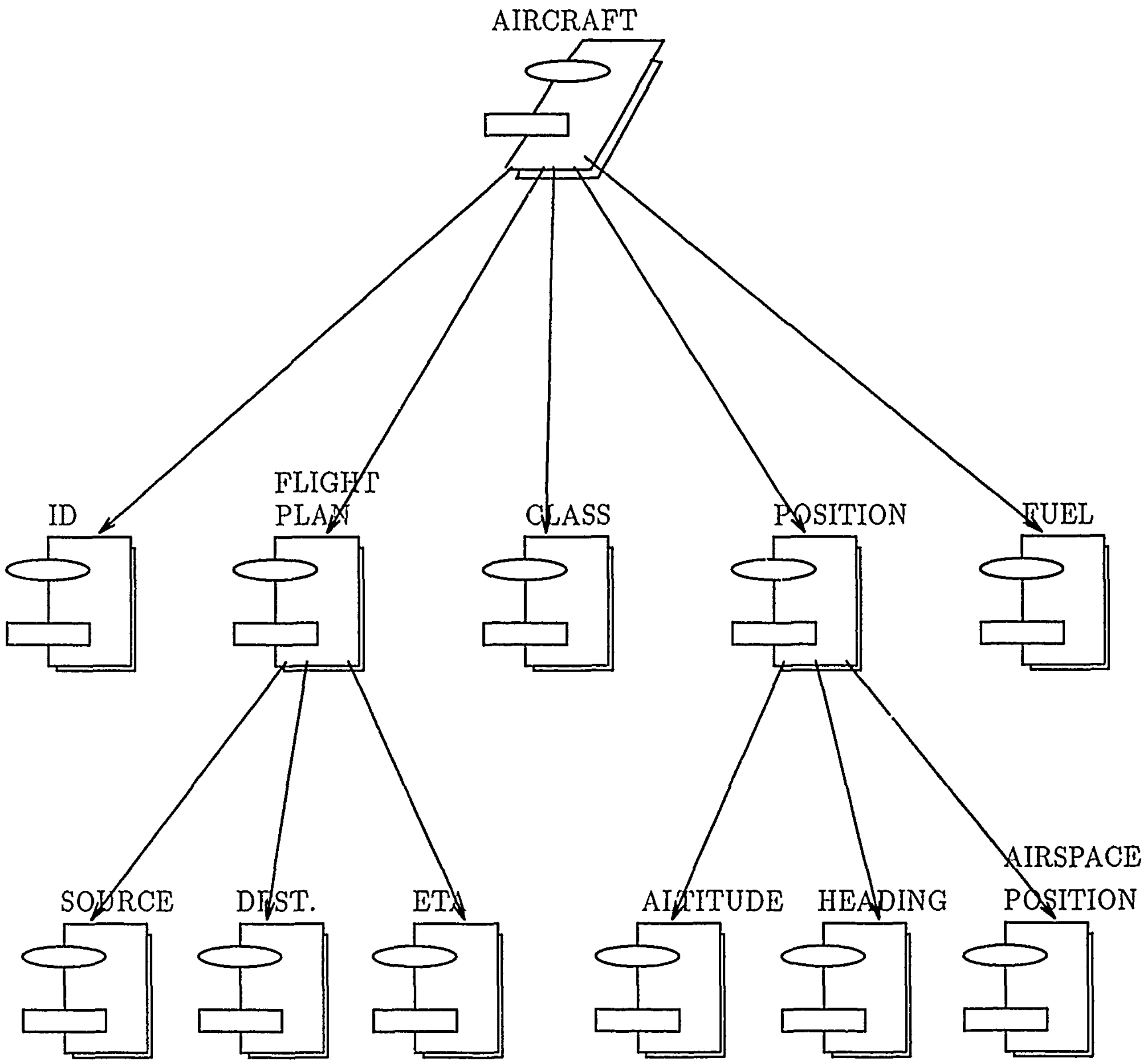


Figure 4.13. Aircraft and Attributes

V. Validation of Concurrency Heuristics

This chapter presents a plan for determining the validity of the concurrency heuristics developed in the first four chapters. A brief discussion of validation methods is given first, followed by a detailed description of the method used to validate the concurrency heuristics given in this thesis, concluding with the results of the validation.

5.1 Validation Methods

Research results may be validated by three methods: analytical, empirical, and expert opinion. Analytical validation seeks to establish the research results by proving the results follow from established principles or concepts, much the same as a mathematician proves a theorem using axioms, postulates, and previously proven theroems. While this method is the most rigorous of the three, it is also the most difficult to apply in the software engineering arena. The reason for this is that there are few, if any, widely accepted principles from which to prove further results. Those principles that do seem to be established, such as high cohesion, low coupling, information hiding, etc., have not themselves been proven analytically or empirically, but are rather accepted, or so it seems, based on expert opinion (the third method).

Empirical validation is based either on observation of naturally occurring phenomena, or on a controlled experiment designed to demonstrate the truth or falsehhod of a concept. Again, software engineering principles do not easily yield to empirical validation, mainly because the phenomena to be observed are usually intangible. For example, the superiority of a data compression algorithm may be demonstrated by implementing it and comparing its performance against other data compression algorithms. However, a theory of software modularization cannot be demonstrated simply by applying the theory in implementing a system, as the process of applying the theory is subjective–each designer will apply it a little differently in all but the most trivial cases. Consequently, implementing the ATC simulation described in chapter four using the concurrency heuristics says nothing about the validity of the heuristics.

The final validation method is expert opinion, in which experts in the field evaluate the research results and provide their considered opinion on the validity of the results. This is admittedly a subjective process, but it does provide some condidence in the research and is certainly better than no validation at all.

5.2 Validation Approach for Concurrency Heuristics

The validation method chosen for this thesis is expert opinion. The concurrency heuristics, a summary of the design of the ATC simulation, and a questionaire were distributed to fourteen experts. The experts were chosen based on their experience with object-oriented design. Of the fourteen, 11 responded. The validation package and the experts' responses appear in Appendix B.

5.3 Validation Results

For convenience the concurrency heuristics questionaire is shown in Figure 5.1, with the results summarized in Table 5.1. Each question in the questionaire will be discussed in turn.

Question	Mean	Std Dev	Ideal
1	4.1	.7	5.0
2	4.4	.8	5.0
3	1.5	.5	1.0
4	2.1	.8	1.0
5	1.2	.4	1.0

Table 5.1. Questionaire Results

Question one was necessary to ensure the experts understood what was being presented. Most of the experts felt the heuristics were understandable (average 4.1 out a possible 5.0), although some commented that heuristic one was rather vague.

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

4. Is there overlap among the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

Figure 5.1. Expert Opinion Questionaire

The first heuristic deals with building a model of real-world objects, which is a rather vague concept in itself, at least in application. The other heuristics seem to be more concrete, corresponding to concepts that are, for the most part, more familiar to designers.

Question two was probably the most important, at least to the author. The purpose of the thesis was to provide guidance to designers; this question gives an indication whether or not this purpose was realized. The average response of 4.4 out of 5.0 indicates the experts felt the heuristics were helpful to designers. However, several comments provided insight into the usefulness of the heuristics.

One comment concerned the amount of detail in the explanation of the heuristics, *i.e.*, more detail was needed to make the heuristic really useful. The validation package contained only a skeleton explanation of the heuristics (see Appendix B); more detail is contained in chapter three.

Another person noted that use of the wrong heuristic could lead to massive concurrency in the solution; for example, in the ATC problem, twenty- six tasks would be produced by applying heuristic one to the Aircraft object, whereas one aircraft manager task could be derived from heuristic three or four. In a single-processor system the massive concurrency could lead to excessive tasking overhead, in which case the second option, that of a single task managing the concurrency in all the aircraft, might be preferable. However, this is done at the sacrifice of the integrity of the model, since adhering to heuristic one more closely models the problem space than do the other heuristics. These sorts of tradeoffs are normal in software design; the heuristics allow the designer to identify the concurrency, and, consequently, the areas where these tradeoffs exist.

Questions three was a completeness question. To be usefule, a set of heuristics must be complete, *i.e.*, it must identify all possible concurrency situations. While none of the experts were willing to subscribe to such a strong statement, none came up with any situations not covered by the heuristics.

Question four addressed the issue of redundancy in the heuristics, whether more than one heuristic could apply to the same situation. All agreed there is some overlap among the heuristics, but not a great deal (average response of 2.1 with an ideal of 1.0). Some commented that redundancy is not a real problem; the important matter is that the concurrency be identified. This is probably true in general, but returning to the discussion on question three, there may be situations where the overlap is actually desirable. In determining concurrency in the Aircraft object, two heuristics were applied and resulted in different designs, the choice of which had significant impact on the conceptual integrity of the design as well as potentially affecting the performance of the implementation. In this case the overlap among the heuristic gave the designer more flexibility to make design tradeoffs.

Question five is important from an overall software engineering standpoint, since any heuristics which violate accepted practice will likely not be accepted. On this question the experts averaged a 1.2 with 1.0 being the ideal.

5.4 Conclusion

The results of the questionaire were very encouraging. The prevailing opinion among the experts was that the heuristics are helpful to designers, understandable, and complete. From this we may conclude that the heuristics appear to be sound.

VI. *Conclusions and Recommendations*

6.1 Summary

The purpose of this thesis was to develop heuristics for determining concurrency in object-oriented designs of real-time systems. This was accomplished by first investigating heuristics available to real-time designers using other paradigms (Structured Analysis, Jackson System Development) and then examining the object-oriented approach to see where these existing heuristics may apply. The survey of existing heuristics is contained in chapter two, and the heuristics for object-oriented design are in chapter three.

In real-time design using Structured Design techniques, the heuristics for determining concurrency are based on the functional decomposition of the system[Gomaa 1984]. Consequently, the heuristics consider such things as functional cohesion, temporal cohesion, process abstraction, *etc.*, which are not compatible with object-oriented design. However, some of the heuristics, in particular the ones dealing with periodic execution and response to events within time constraints, do apply to object-oriented design.

Jackson System Development (JSD) takes a modeling perspective in designing software systems, which is similar to the object-oriented approach[Cameron 1986]. JSD does not specifically address determining concurrency in real-time systems, but a derivative method, Entity-Life Modeling[Sanden 1989], does provide principles for determining concurrency. In Entity-Life Modeling the system is characterized as a set of sequential behavior patterns in which the entities or objects comprising the system participate. Each separate behavior pattern is then considered a concurrent task in the design.

Object-oriented design models the system under consideration as a set of objects and the operations on those objects. The system is implemented by specifying the interaction of the objects, *i.e.*, the timing and ordering of the operations. The method, as presented by Booch[Booch 1991], does not provide heuristics for determining concurrency. Booch's method is summarized in chapter three.

The heuristics of Gomaa[Gomaa 1984] and the Entity-Life Modeling principle[Sanden 1989] were applied to the object-oriented approach to produce a set of heuristics to guide designers in determining concurrency in the design of real-time systems. The four heuristics are:

1. **Problem-space concurrency.** An object which models concurrency in the problem environment should be implemented as a task. Concurrency in the problem-domain can be determined by identifying behavior patterns, or sequences of events, in which the objects participate. These sequences of events are related to the timing and ordering of the operations on the problem-space objects.

   This concept is closely related to the Entity-Life Modeling principle, the distinction being that object-oriented design focuses on individual objects and their operations, whereas Entity-Life Modeling concentrates on identifying behavior patterns in which any number of objects may participate. Thus, Entity-Life Modeling partitions the concurrency based on the behavior patterns, which may include any number of objects. The object-oriented approach partitions the concurrency according to the objects which contain the behavior patterns.

2. **Time constraints.** An object whose behavior or operations are constrained by time requirements should be a task. This heuristic combines the timing related heuristics of Gomaa[Gomaa 1984] and Nielsen and Shumate[Nielsen and Shumate 1989]. Thus an operation that is invoked at regular intervals is considered a separate task (in structured design these are periodic functions). Also, an operation which must respond to an event within a certain time period is a task, for example, an operation invoked in response to an interrupt.

3. **Computational requirements.** An object whose behavior or operations require substantial computational resources should be a task. These tasks would most likely run in background at a low priority.

4. **Solution-space objects.** An object introduced in the software solution to protect a shared data store, decouple two interacting tasks, or synchronize the behavior of two or more objects should be a task. Booch calls these 'mechanisms', *i.e.*, objects with no counterpart in the problem-space, but which are necessary to implement the system on a real machine. An example would be a shared data store implemented in Ada; a task must be used to guarantee mutual exclusion.

6.2 Conclusions

The concurrency heuristics are powerful tools for determining concurrency in object-oriented design of real-time systems. The set of heuristics is small enough to be easily remembered, yet general enough to determine concurrency in most cases. The heuristics are easy to understand and apply, and, in some cases, they allow the designer to determine concurrency from different perspectives, allowing the designer a range of choices in the implementation.

While the heuristics are referred to as 'design' heuristics, they actually can be useful during a broader portion of the development life-cycle than just the design phase. In object-oriented design, the analysis, design, and implementation stages are not rigidly delineated; rather, they are actually a continuum in which the software model progresses from a more abstract representation (analysis) to a more concrete representation (implementation). The heuristics may be applied at any point on the continuum. For example, in the ATC problem, concurrency was determined in the ATC object very early in the analysis; in fact it can be determined from the requirements definition. The concurrency in the console object, however, was determined after the object had been almost completely designed. The problem-space heuristic, by its very nature, does not determine concurrency until late in the process, perhaps not until detailed design.

Using object-oriented design, a designer seeks to build a model of the problem-space, *i.e.*, the structure of the solution should reflect the structure of the problem. This is a central concept in the object-oriented approach; consequently, the first heuristic is the most important from a pure modeling perspective and should be the first consideration in determining concurrency in a particular system. The remaining heuristics are important from a practical standpoint, since considerations unrelated to producing a model of the problem-space may force the designer to implement concurrency; for example, a periodic task, or a computationally intensive task, or a shared data store may not have corresponding objects in the problem-space, yet they require concurrency implementation nonetheless. To ensure the primacy of the model, however, the first heuristic should be considered first.

6.3 Recommendations

In this thesis the concurrency heuristics were applied to the ATC simulation, for which concurrency was rather easily determined. The ATC problem was a self-contained system which had a rather simple user interface and no external objects other than the keyboard and display. Also the ATC was not a 'hard' real-time system, *i.e.*, missing a timing constraint (display update) did not constitute a system failure. Another characteristic of the ATC problem as implemented in this thesis was that a single-processor system was assumed.

One possible area for further exploration is to see if the heuristics apply as well to other kinds of real-time systems. Do they work as well for more complex problems, ones with hard real-time requirements, or that require external files to be maintained in real-time? Systems which require a large number of interrupt handling routines would also be a good candidate.

Another area for further research is in applying the heuristics to distributed real-time systems. One of the assumptions of this thesis was that, even in a distributed system, there may be more than one process executing on a processor, so the heuristics apply at the processor or node level; the network or system level was not considered. At the network level, issues external to the system being designed must be considered, such as the processor interconnection network, the interprocessor message passing mechanism, and load balancing among the processors. It should be determined how the concurrency heuristics may be applied to these issues, or what heuristics must be added to the set to cover distributed systems.

Appendix A. *Air Traffic Control Simulation Object-Class Specifications and Ada Specifications*

This appendix contains the object-class specifications for the Air Traffic Control simulation introduced in chapter four, followed by the Ada package specifications for the major system objects.

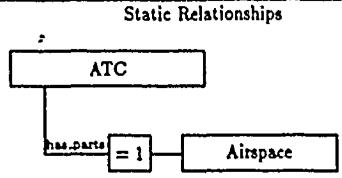
A.1 Object-Class Specifications

CLASS SPECIFICATION

Class Name: ATC

Description: This is the main object of the simulation. It controls the interaction of the other objects.

Static Relationships



Dynamic Relationships

Suffered Operations

	Descriptive Name
Selectors:	
Constructors:	

Required Operations

	Name	Applied to
Selectors:	Get_ID	Command
	Is_Status	Command
	Is_Termination	Command
	Is_COMMAND	Command
Constructors:	Disp_Pre_Mge	Console
	Disp_Map_Item	Console
	Disp_Time	Console
	Disp_Input	Console
	Disp_Roger	Console
	Get_Input	Console
	Create_Cmd	Command

Exceptions

Name	Raised by
Time_Expired	ATC
Invalid_Cmd	Command
Invalid_Acft	Command
Fuel_Exhausted	Airspace
Conflict_Error	Airspace
Bdary_Error	Airspace

QA

Initial:

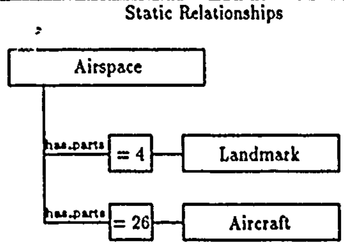
Figure A.1. ATC Object-Class Specification

CLASS SPECIFICATION

Class Name: Airspace

Description: Represents the airspace abstraction.

Static Relationships



Dynamic Relationships

Suffered Operations

	Descriptive Name
Selectors:	Is_Done
	Get_Landmark_Location
Constructors:	

Required Operations

	Name	Applied to
Selectors:	Get_ID	Command
	Is_Status	Command
	Is_Termination	Command
	Is_COMMAND	Command
	Get_Location	Landmark
	Get_ID	Aircraft
	Get_Source	Aircraft
	Get_Destination	Aircraft
	Get_ETA	Aircraft
	Get_Class	Aircraft
	Get_Heading	Aircraft
	Get_Fuel	Aircraft
	Get_Position	Aircraft
	Get_Altitude	Aircraft
Constructors:	Set_Location	Landmark
	Set_Landmark	Landmark
	Set_ID	Aircraft

Figure A.2. Airspace Object-Class Specification

CLASS SPECIFICATION			
Class Name: Airspace			(Continued)
Description: Represents the airspace abstraction.			

Suffered Operations Descriptive Name		Required Operations Name	Applied to
Constructors:		Constructors: Set_Flight_Plan	Aircraft
		Set_Class	Aircraft
		Set_Heading	Aircraft
		Set_Altitude	Aircraft
		Set_Fuel	Aircraft
		Set_Position	Aircraft
		Take_Off	Aircraft
		Hold_at_Navaid	Aircraft
		Clr_for_Apprch	Aircraft
		Clr_for_Ldg	Aircraft
		Continue_Straight	Aircraft
		Update_Position	Aircraft
Iterators:	Update_Airspace	Iterators:	

Exceptions Name	Raised by	QA
Fuel_Exhausted	Aircraft	
Conflict_Error	Airspace	Initial:
Bdary_Error	Airspace	

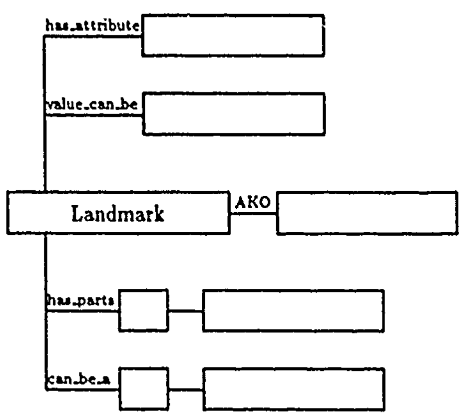
Figure A.3. Airspace Object-Class Specification(continued)

CLASS SPECIFICATION

Class Name: Landmark

Description: This class represents a landmark and its position within the airspace. A landmark can be one of three types: a navaid, an airport, or an entry/exit fix. Each of these has two or more possible values: 2 navaids, 2 airports, 10 entry/exit fixes.

Static Relationships



Dynamic Relationships

Suffered Operations

	Descriptive Name
Selectors:	Get_Location
Constructors:	Set_Location
	Set_Landmark

Required Operations

	Name	Applied to
Selectors:		
Constructors:		

Exceptions

Name	Raised by

QA

Initial:

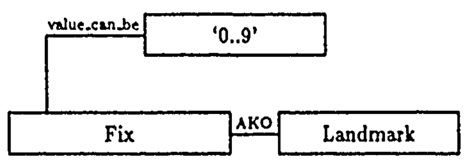
Figure A.4. Landmark Object-Class Specification

CLASS SPECIFICATION

Class Name: Fix

Description: This class represents an entry/exit fix which is a kind of landmark.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Selectors: Get_Location

Constructors: Set_Location

Set_Landmark

Required Operations

Name Applied to

Selectors:

Constructors:

Exceptions

Name Raised by

QA

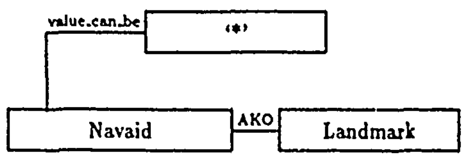
Initial:

Figure A.5. Fix Object-Class Specification

CLASS SPECIFICATION

Class Name: Navaid

Description: This class represents a navigational beacon which is a kind of landmark.

Static Relationships	Dynamic Relationships
	

Suffered Operations		Required Operations	
	Descriptive Name	Name	Applied to
Selectors:	Get_Location	Selectors:	
Constructors:	Set_Location	Constructors:	
	Set_Landmark		

Exceptions		QA
Name	Raised by	Initial:

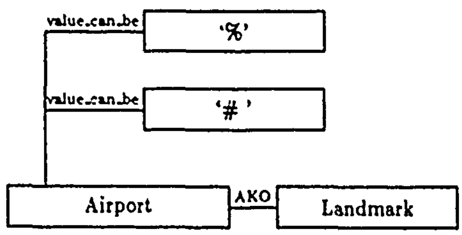
Figure A.6. Navaid Object-Class Specification

CLASS SPECIFICATION

Class Name: Airport

Description: This class represents an airport which is a kind of landmark.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Selectors: Get_Location

Constructors: Set_Location

Set_Landmark

Required Operations

Name Applied to

Selectors:

Constructors:

Exceptions

Name Raised by

QA

Initial:

Figure A.7. Airport Object-Class Specification

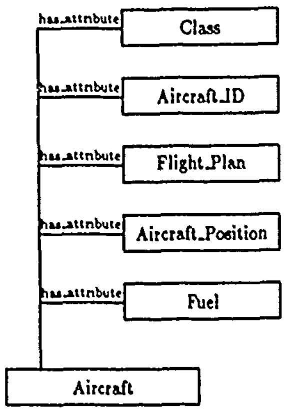
CLASS SPECIFICATION	
Class Name: Airspace_Location	
Description: Represents the location of an object in the airspace.	
Static Relationships Airspace_Location —AKO— record	**Dynamic Relationships**
Suffered Operations Descriptive Name Selectors: Constructors: Iterators:	**Required Operations** Name Applied to Selectors: Constructors: Iterators:
Exceptions Name Raised by	**QA** Initial:

Figure A.8. Airspace_Location Object-Class Specification

CLASS SPECIFICATION

Class Name: Aircraft

Description: Represents the aircraft abstraction.

Static Relationships	Dynamic Relationships
	

Suffered Operations

	Descriptive Name
Selectors:	Get_ID
	Get_Source
	Get_Destination
	Get_ETA
	Get_Class
	Get_Heading
	Get_Fuel
	Get_Position
	Get_Altitude
Constructors:	Set_ID
	Set_Flight_Plan
	Set_Class

Required Operations

	Name	Applied to
Selectors:	Get_Location	Aircraft_Position
	Get_Altitude	Aircraft_Position
	Get_Heading	Aircraft_Position
	Get_Source	Flight_Plan
	Get_Destination	Flight_Plan
	Get_Eta	Flight_Plan
Constructors:		

Figure A.9. Aircraft Object-Class Specification

CLASS SPECIFICATION				
Class Name: Aircraft				(Continued)
Description: Represents the aircraft abstraction.				
Suffered Operations		Required Operations		
	Descriptive Name		Name	Applied to
Constructors:	Set_Heading	Constructors:	Set_Location	Aircraft_Position
	Set_Altitude		Set_Heading	Aircraft_Position
	Set_Fuel		Set_Altitude	Aircraft_Position
	Set_Position		Set_Flight_Plan	Flight_Plan
	Take_Off			
	Hold_at_Navaid			
	Clear_for_Approach			
	Clear_for_Landing			
	Continue_Straight			
	Update_Position			

Exceptions		QA
Name	Raised by	
Fuel_Exhausted	Aircraft	Initial:

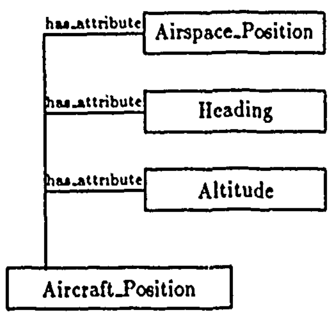
Figure A.10. Aircraft Object-Class Specification(continued)

CLASS SPECIFICATION

Class Name: Aircraft_Position

Description: Represents the position of an aircraft within the airspace.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Selectors: Get_Location

Get_Altitude

Get_Heading

Constructors: Set_Location

Set_Heading

Set_Altitude

Required Operations

Name Applied to

Selectors:

Constructors:

Exceptions

Name Raised by

QA

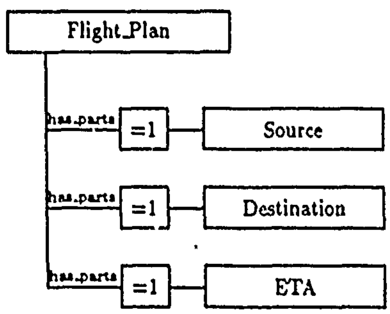
Initial:

Figure A.11. Aircraft_Position Object-Class Specification

CLASS SPECIFICATION

Class Name: Flight_Plan

Description: Represents the flight plan of a particular aircraft.

Static Relationships	Dynamic Relationships
	

Suffered Operations Descriptive Name		Required Operations Name	Applied to
Selectors:	Get_Source Get_Destination Get_Eta	Selectors:	
Constructors:	Set_Flight_Plan	Constructors:	

Exceptions		QA
Name	Raised by	Initial:

Figure A.12. Flight_Plan Object-Class Specification

CLASS SPECIFICATION	
Class Name: Fuel	
Description: Represents the fuel remaining in an aircraft.	
Static Relationships Fuel —AKO— integer	**Dynamic Relationships**
Suffered Operations Descriptive Name Selectors: Constructors: Iterators:	**Required Operations** Name Applied to Selectors: Constructors: Iterators:
Exceptions Name Raised by	**QA** Initial:

Figure A.13. Fuel Object-Class Specification

CLASS SPECIFICATION

Class Name: Altitude

Description: Represents the altitude of an aircraft.

Static Relationships	Dynamic Relationships
Altitude —AKO— integer	

Suffered Operations	Required Operations
Descriptive Name	Name / Applied to
Selectors:	Selectors:
Constructors:	Constructors:
Iterators:	Iterators:

Exceptions	QA
Name / Raised by	Initial:

Figure A.14. Altitude Object-Class Specification

CLASS SPECIFICATION	
Class Name: Heading	
Description: Represents the heading of an aircraft.	
Static Relationships Heading —AKO— enumeration type	**Dynamic Relationships**
Suffered Operations Descriptive Name Selectors: Constructors: Iterators:	**Required Operations** Name Applied to Selectors: Constructors: Iterators:
Exceptions Name Raised by	**QA** Initial:

Figure A.15. Heading Object-Class Specification

CLASS SPECIFICATION	
Class Name: ETA	
Description: Represents the estimated time which an aircraft will appear on the display.	

Static Relationships	Dynamic Relationships
ETA —AKO— integer	

Suffered Operations (Descriptive Name)	Required Operations (Name / Applied to)
Selectors:	Selectors:
Constructors:	Constructors:
Iterators:	Iterators:

Exceptions (Name / Raised by)	QA
	Initial:

Figure A.16. ETA Object-Class Specification

CLASS SPECIFICATION

Class Name: Source

Description: Represents the source of an aircrafts flight plan.

Static Relationships	Dynamic Relationships
Source —AKO— enumeration type	

Suffered Operations	Required Operations	
Descriptive Name	Name	Applied to
Selectors:	Selectors:	
Constructors:	Constructors:	
Iterators:	Iterators:	

Exceptions		QA
Name	Raised by	Initial:

Figure A.17. Source Object-Class Specification

<table>
<tr><td colspan="3">CLASS SPECIFICATION</td></tr>
<tr><td colspan="3">Class Name: Destination</td></tr>
<tr><td colspan="3">Description: Represents the destination of an aircrafts flight plan.</td></tr>
<tr><td>Static Relationships
Destination —AKO— enumeration type</td><td colspan="2">Dynamic Relationships</td></tr>
<tr><td>Suffered Operations
Descriptive Name
Selectors:
Constructors:
Iterators:</td><td colspan="2">Required Operations
Name Applied to
Selectors:
Constructors:
Iterators:</td></tr>
<tr><td colspan="2">Exceptions
Name Raised by</td><td>QA
Initial:</td></tr>
</table>

Figure A.18. Destination Object-Class Specification

CLASS SPECIFICATION

Class Name: Aircraft_ID

Description: Represents the tail number on an aircraft.

Static Relationships

Dynamic Relationships

Suffered Operations

Descriptive Name

Selectors:

Constructors:

Iterators:

Required Operations

Name Applied to

Selectors:

Constructors:

Iterators:

Exceptions

Name Raised by

QA

Initial:

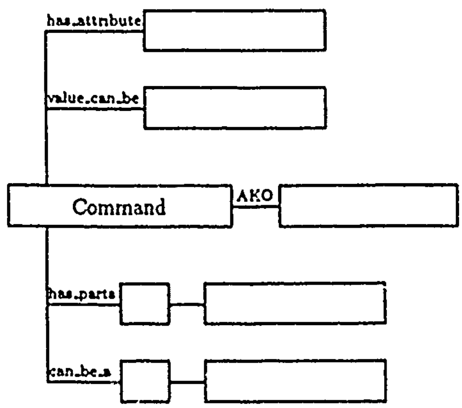
Figure A.19. Aircraft_ID Object-Class Specification

CLASS SPECIFICATION

Class Name: Command

Description: Provides the command abstraction for the ATC simulation.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Selectors: Get_ID

Is_Status

Is_Termination

Is_COMMAND

Constructors: Create_Command

Required Operations

Name Applied to

Selectors:

Constructors:

Exceptions

Name	Raised by
Invalid_Cmd	Command
Invalid_Aircraft	Command

QA

Initial:

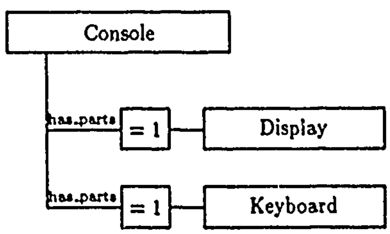
Figure A.20. Command Object-Class Specification

CLASS SPECIFICATION

Class Name: Console

Description: This object provides the I/O abstraction for the ATC simulation.

Static Relationships



Dynamic Relationships

Suffered Operations

	Descriptive Name
Constructors:	Disp_Prev_Msg
	Disp_Map_Item
	Disp_Time
	Disp_Input
	Disp_Roger
	Get_Input

Required Operations

	Name	Applied to
Constructors:	Disp_Prev_Msg	Display
	Disp_Map_Item	Display
	Disp_Time	Display
	Disp_Input	Display
	Disp_Roger	Display
	Get_Input	Keyboard

Exceptions

Name	Raised by

QA

Initial:

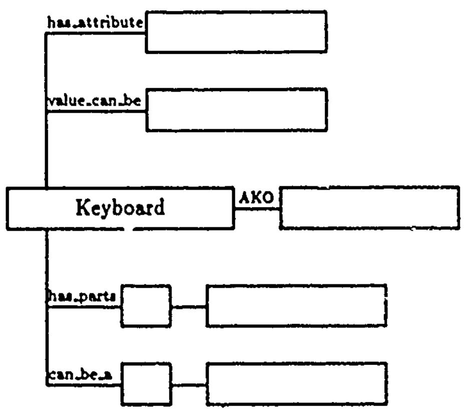
Figure A.21. Console Object-Class Specification

CLASS SPECIFICATION

Class Name: Keyboard

Description: Provides the interface to the physical keyboard.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Constructors: Get_Input

Required Operations

Name | Applied to

Constructors:

Exceptions

Name | Raised by

QA

Initial:

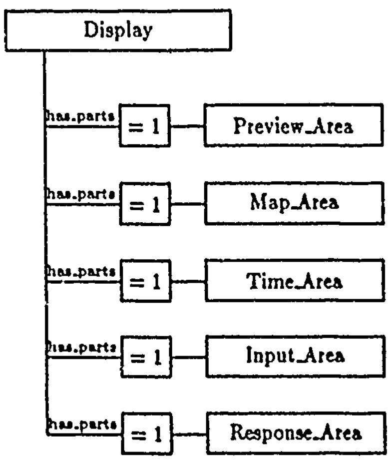
Figure A.22. Keyboard Object-Class Specification

CLASS SPECIFICATION

Class Name: Display

Description: Provides the output for the simulation.

Static Relationships



Dynamic Relationships

Suffered Operations

	Descriptive Name
Constructors:	Disp_Prev_Msg
	Disp_Map_Item
	Disp_Time
	Disp_Input
	Disp_Roger

Required Operations

	Name	Applied to
Constructors:	Disp_Prev_Msg	Preview_Area
	Disp_Map_Item	Map_Area
	Disp_Time	Time_Area
	Disp_Input	Input_Area
	Disp_Roger	Response_Area

Exceptions

Name	Raised by

QA

Initial:

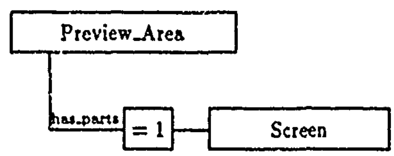
Figure A.23. Display Object-Class Specification

CLASS SPECIFICATION

Class Name: Preview_Area

Description: Represents the area of the display where the preview messages are shown.

Static Relationships



Dynamic Relationships

Suffered Operations	Required Operations	
Descriptive Name	Name	Applied to
Constructors: Disp_Prev_Msg	Constructors: Disp_Prev_Msg	Screen

Exceptions

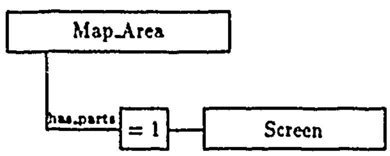
Name	Raised by	QA
		Initial:

Figure A.24. Preview_Area Object-Class Specification

CLASS SPECIFICATION

Class Name: Map_Area

Description: Represents the area of the display where the map of the control space is displayed.

Static Relationships	Dynamic Relationships
	

Suffered Operations Descriptive Name	Required Operations Name / Applied to
Constructors: Disp_Map_Item	Constructors: Disp_Map_Item / Screen

Exceptions		QA
Name	Raised by	Initial:

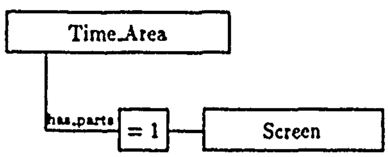
Figure A.25. Map_Area Object-Class Specification

CLASS SPECIFICATION

Class Name: Time_Area

Description: Represents the area of the display where the time remaining is displayed.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Constructors: Disp_Time

Required Operations

Name	Applied to
Constructors: Disp_Time	Screen

Exceptions

Name	Raised by

QA

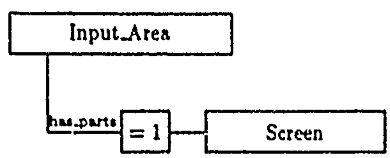
Initial:

Figure A.26. Time_Area Object-Class Specification

CLASS SPECIFICATION

Class Name: Input_Area

Description: Represents the area of the display where the input is echoed.

Static Relationships	Dynamic Relationships
	

Suffered Operations	Required Operations	
Descriptive Name	Name	Applied to
Constructors: Disp_Input	Constructors: Disp_Input	Screen

Exceptions		QA
Name	Raised by	Initial:

Figure A.27. Input_Area Object-Class Specification

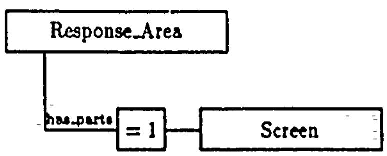
CLASS SPECIFICATION		
Class Name: Response_Area		
Description: Represents the area of the display where the system response is displayed.		
Static Relationships 	Dynamic Relationships	
Suffered Operations Descriptive Name Constructors: Disp_Roger	Required Operations Name Applied to Constructors: Disp_Response Screen	
Exceptions Name Raised by		QA Initial:

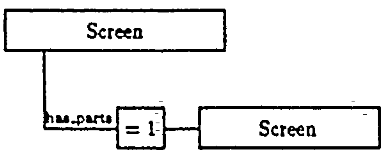
Figure A.28. Response_Area Object-Class Specification

CLASS SPECIFICATION

Class Name: Screen

Description: Provides the interface to the physical screen.

Static Relationships



Dynamic Relationships

Suffered Operations

Descriptive Name

Constructors: Disp_Prev_Msg

Disp_Map_Item

Disp_Time

Disp_Input

Disp_Response

Required Operations

Name	Applied to

Constructors:

Exceptions

Name	Raised by

QA

Initial:

Figure A.29: Screen Object-Class Specification

CLASS SPECIFICATION		
Class Name: Simulation_Time		
Description: Represents the time remaining in the simulation.		
Static Relationships Simulation_Time —AKO— integer	**Dynamic Relationships**	
Suffered Operations Descriptive Name Selectors: Constructors: Iterators:	**Required Operations** Name Applied to Selectors: Constructors: Iterators:	
Exceptions Name: Raised by		**QA** Initial:

Figure A.30. Simulation_Time Object-Class Specification

CLASS SPECIFICATION

Class Name: Map_Item

Description: Represents an item to be placed in the map display.

Static Relationships	Dynamic Relationships
Map_Item —AKO— record	

Suffered Operations Descriptive Name	Required Operations Name Applied to
Selectors:	Selectors:
Constructors:	Constructors:
Iterators:	Iterators:

Exceptions Name Raised by	QA Initial:

Figure A.31. Map_Item Object-Class Specification

CLASS SPECIFICATION

Class Name: Preview_Message_Count

Description: Represents the number of the current preview message.

Static Relationships	Dynamic Relationships
Preview_Message_Count —AKO— integer	

Suffered Operations	Required Operations
Descriptive Name	Name / Applied to
Selectors:	Selectors:
Constructors:	Constructors:
Iterators:	Iterators:

Exceptions		QA
Name	Raised by	Initial:

Figure A.32. Preview_Message_Count Object-Class Specification

CLASS SPECIFICATION	
Class Name: Preview_Message	
Description: Represents the string into which the preview message is placed.	
Static Relationships Preview_Message —AKO— string	**Dynamic Relationships**
Suffered Operations Descriptive Name Selectors: Constructors: Iterators:	**Required Operations** Name Applied to Selectors: Constructors: Iterators:
Exceptions Name Raised by	**QA** Initial:

Figure A.33. Preview_Message Object-Class Specification

CLASS SPECIFICATION		
Class Name: Input_String		
Description: Represents the string into which the user input is placed.		
Static Relationships Input_String —AKO— string	**Dynamic Relationships**	
Suffered Operations Descriptive Name Selectors: Constructors: Iterators:	**Required Operations** Name Applied to Selectors: Constructors: Iterators:	
Exceptions Name Raised by		**QA** Initial:

Figure A.34. Input_String Object-Class Specification

A.2 Ada Specifications

Following are the Ada package and subprogram specifications for the ATC problem. Only the major objects are included.

A.2.0.1 ATC Object

```
with Calendar;
with Text_IO;
use  Text_IO;
with Command_PKG;
use  Command_PKG;
with Console_PKG;
with Classes_PKG;
use  Classes_PKG;
with airspace_PKG;
with Aircraft_Attributes_PKG;
procedure ATC is
--*********************************************************************
--          CLASS:    ATC
--  REPRESENTATION:   none
--        USED BY:    none
--           USES:    Command,  Classes, Airspace, Console,
--                    Aircraft_Attributes, Calenda , Text_IO
--      OPERATIONS:   none
--
--  PURPOSE: This object represents the the entire air traffic contol
--           simulation.
--*********************************************************************

   Simulation_Length: Classes_PKG.Simulation_Time;
   This_Command: Command_PKG.Command;
   Controller_Input: Classes_PKG.Input_String;
   Time_Expired:exception;
   package Time_IO is new integer_io(Classes_PKG.Simulation_Time);

   task Update_Airspace is
     entry Start(Simulation_Length: in  Classes_PKG.Simulation_Time);
     entry Stop;
   end Update_Airspace;

   use  Calendar;
```

```
task body Update_Airspace is
  Time_Left:Classes_PKG.Simulation_Time;
  Minute_Counter: integer:=1;
  Time_Expired:exception;
  Next_Update:Calendar.Time;
  Update_Interval:duration:=15.0;
begin
  accept Start(Simulation_Length: in Classes_PKG.Simulation_Time) do
       Time_Left:=Simulation_Length;
  end Start;
  Console_PKG.Display_Time(Time_Left);
  Next_Update:=Calendar.clock;
  loop
    Next_Update:=Next_Update + Update_Interval;
    delay Next_Update - Calendar.clock;
    -- retrieve airspace updates
    -- display airspace updates
    if Minute_Counter=4 then
       Time_Left:=Time_Left-1;
       Console_PKG.Display_Time(Time_Left);
       if Time_Left=0 then
         raise Time_Expired;
       end if;
       Minute_Counter:=1;
    else
       Minute_Counter:=Minute_Counter+1;
    end if;
    select
      accept Stop;
      exit;
    else
      null;
    end select;
  end loop;
end Update_Airspace;

begin
  put("Enter the simulation length: ");
  Time_IO.get(Simulation_Length);
  -- Draw_Initial_Map;
  Update_Airspace.Start(Simulation_Length);
  delay 1.0;
  loop
```

```
    Controller_Input:=Console_PKG.get_input;
    Console_PKG.Display_Input(Controller_Input);
    If Command_PKG.Is_Termination_Request(Controller_Input) then
        Console_PKG.Display_Input("Terminating simulation.");
        -- Terminate_Simulation;
        Update_Airspace.Stop;
        exit;
    end if;
    begin
      This_Command:= Command_PKG.Create_Command(Controller_Input);
    exception
      when Invalid_Command    =>
        Console_PKG.Display_Input("Invalid command.");
      when Invalid_Aircraft   =>
        Console_PKG.Display_Input("Invalid aircraft.");
      when others             =>
        Console_PKG.Display_Input("Something else went wrong.");
    end;
    if not Command_PKG.Is_Status(This_Command) then
       Console_PKG.Display_Roger;
       -- Execute Command
    else
       -- Get Status
       -- Display Status
      null;
    end if;
    delay 1.0;
  end loop;
  exception
    when Time_Expired=>
      put_line("You ran out of time!!!!!");
    when others =>
      put_line("Something bad went wrong.");
end ATC;

with Aircraft_PKG;
with Aircraft_Attributes_PKG;
with Command;
with Landmark_PKG;
with Classes_PKG;
package Airspace_PKG is
--*********************************************************************
--          CLASS:    Airspace
```

```
--  REPRESENTATION:    none
--         USED BY:    ATC
--            USES:    Command, Landmark, Classes, Aircraft Attributes
--      OPERATIONS:    Initialize_Airspace - sets the location of all
--                                  landmarks in the airspace and
--                                  passes it back to ATC for
--                                  display
--                     Update_Airspace  -  gets the position updates
--                                  of the aircraft, checks for
--                                  errors, and passes the updates
--                                  back to ATC
--                     Execute_Command  -  performs the specified command
--                                  on the specified aircraft
--                     Is_Done          -  returns true if 26 aircraft have
--                                  been dispositioned
--                     Get_Landmark_Location - returns the location of the
--                                  specified landmark.
--                                   based on the heading, speed, etc.
--
--  PURPOSE: This class represents the airspace.
--*****************************************************************

  package Update_Record_List is new ???????(Update_Record_PKG.Update_Record);
  procedure Initialize_Airspace (Update_List: out Update_Record_List);
  procedure Update_Airspace (Update_List: out Update_Record_List);
  procedure Execute_Command (This_Command: in Command_PKG.Command;
                         This_Aircraft:Aircraft_Attributes_PKG.Aircraft_ID);
  function Is_Done return Boolean;
  function Get_Landmark_Location(This_Landmark: in Landmark_PKG.Landmark)
                             return Classes_PKG.Airspace_Position;
end Airspace_PKG;

with Aircraft_Position_PKG;
with Flight_Plan_PKG;
with Aircraft_Attributes_PKG;
package Aircraft_PKG is
--*****************************************************************
--           CLASS:    Aircraft
--  REPRESENTATION:    record
--         USED BY:    Airspace
--            USES:    Aircraft_Attributes, Flight_Plan, Aircraft_Position
--      OPERATIONS:    Get_ID         -  returns the ID of the aircraft
--                     Get_Source     -  returns the source of the
```

```
--                                 aircraft
--                     Get_Destination  -  returns the destination of the
--                                         aircraft
--                     Get_ETA        -  returns the ETA of the aircraft,
--                                       the time the aircraft will appear
--                                       on the display.
--                     Get_Class      -  returns the class of aircraft,
--                                       whether it is a jet  r prop.
--                     Get_Heading    -  returns the heading of the aircraft
--                     Get_Fuel       -  returns the fuel level of the
--                                       aircraft
--                     Get_Position   -  returns the 3 dimensional position
--                                       of the aircraft
--                     Get_Altitude   -  returns the altitude of the aircraft
--
--                     Set_ID         -  assigns an ID to the aircraft
--                     Set_Flight_Plan - assigns a flight plan to the
--                                        aircraft
--                     Set_Class      -  assigns a class to the aircraft
--                     Set_Heading    -  assigns a heading to the aircraft
--                     Set_Altitude   -  assigns an altitude to the aircraft
--                     Set_Fuel       -  assigns a fuel level to the aircraft
--                     Set_Position   -  assigns a position to the aircraft
--                     Take_Off       -  sets the Take_Off flag to true
--                     Hold_at_Navaid-  sets the Hold_at_Navaid flag to true
--                     Clear_for_Approach - sets the Clear_for_Approach flag
--                                           to true
--                     Clear_for_Landing  - sets the Clear_for_Landing flag
--                                           to true
--                     Continue_Straight  - does nothing
--                     Update_Position - sets the new position of the aircraft
--                                        based on the heading, speed, etc.
--
-- PURPOSE: This class represents an aircraft in the airspace
--*******************************************************************

  type Aircraft is private;
  function Get_ID (This_Plan: in Aircraft)
                              return Aircraft_Attributes_PKG.Aircraft_ID;
  function Get_Source (This_Plan: in Aircraft)
                              return Aircraft_Attributes_PKG.Source;
  function Get_Destination (This_Plan: in Aircraft)
                              return Aircraft_Attributes_PKG.Destination;
```

```
function Get_ETA (This_Plan: in Aircraft)
                              return Aircraft_Attributes_PKG.ETA_Type;
function Get_Class (This_Plan: in Aircraft)
                              return Aircraft_Attributes_PKG.Class;
function Get_Heading (This_Plan: in Aircraft)
                              return Aircraft_Attributes_PKG.Heading_Type;
function Get_Fuel (This_Plan: in Aircraft)
                              return Aircraft_Attributes_PKG.Fuel;
function Get_Position (This_Plan: in Aircraft)
                              return Aircraft_Position_PKG.Aircraft_Position;
function Get_Altitude (This_Plan: in Aircraft)
                              return Aircraft_Attributes_PKG.Altitude_Type;
procedure Set_ID (This_ID: in Aircraft_Attributes_PKG.Aircraft_ID;
                  This_Plane: out Aircraft);
procedure Set_Flight_Plan (This_Src: in Aircraft_Attributes_PKG.Source;
                           This_DST: in Aircraft_Attributes_PKG.Destination;
                           This_ETA: in Aircraft_Attributes_PKG.ETA_Type;
                           This_Plane : out Aircraft);
procedure Set_Class (This_Class: in Aircraft_Attributes_PKG.Class;
                           This_Plane : out Aircraft);
procedure Set_Heading (This_Heading: in Aircraft_Attributes_PKG.Heading_Type;
                           This_Plane : out Aircraft);
procedure Set_Altitude (This_Altitude: in Aircraft_Attributes_PKG.Altitude_Type;
                           This_Plane : out Aircraft);
procedure Set_Fuel (This_Fuel: in Aircraft_Attributes_PKG.Fuel;
                           This_Plane : out Aircraft);
procedure Set_Position (This_Position: in Aircraft_Position_PKG.Aircraft_Position;
                           This_Plane : out Aircraft);
procedure Take_Off (This_Position: out Aircraft);
procedure Hold_at_Navaid (This_Position: out Aircraft);
procedure Clear_for_Approach (This_Position: out Aircraft);
procedure Clear_for_Landing (This_Position: out Aircraft);
procedure Continue_Straight (This_Position: out Aircraft);
procedure Update_Position (This_Position: out Aircraft);
private
  type Aircraft is
    record
      ID: Aircraft_Attributes_PKG.Aircraft_ID;
      Active: Boolean:=false;
      Flight_Plan: Flight_Plan_PKG.Flight_Plan;
      Class: Aircraft_Attributes_PKG.Class;
      Fuel_Level: Aircraft_Attributes_PKG.Fuel;
      Position: Aircraft_Position_PKG.Aircraft_Position;
```

```
      Approach: Boolean:=false;
      Landing : Boolean:=false;
      Hold    : Boolean:=false;
    end record;
end Aircraft_PKG;
```

A.2.0.2 Console Object

```
--*********************************************************************
--          OBJECT:    Console
--  RFPRESENTATION:    Subobjects - display, keyboard
--         USED BY:    ATC
--            USES:    Display_PKG, Keyboard_PKG, Classes_PKG
--      OPERATIONS:    Display_Preview_Message - displays a preview
--                              message in the preview area
--                     Display_Map_Item - displays a single map item
--                              in the map area
--                     Display_Time - displays the time remaining in
--                                    the simulation in the time area
--                     Display_Input - echos the input to the screen
--                     Get_Input - gets input from the keyboard
--                     Display_Roger - displays a "ROGER" message in
--                             the response area
--                     Clean_Up - kills all tasks at the termination
--                                of the simulation
--
--         PURPOSE:    Provides the I/O to the ATC simulation.
--
--*********************************************************************
with Classes_PKG;
package Console_PKG is
  procedure Display_Preview_Message
                       (Next_Message: in Classes_PKG.Preview_Message;
                        Msg_Num: in Classes_PKG.Preview_Message_Count);
  procedure Display_Map_Item(This_Item: in Classes_PKG.Map_Item);
  procedure Display_Time(New_Time: in Classes_PKG.Simulation_Time);
  procedure Display_Input(This_Input: in String);
  function Get_Input return String;
  procedure Display_Roger;
  procedure Clean_Up;
end Console_PKG;


--*********************************************************************
--          OBJECT:    Keyboard
--  REPPESENTATION:    Subobjects - none
--         USED BY:    Console
--            USES:    ?
--      OPERATIONS:    Get_Input - gets a string from the use
--                     Clean_Up - kills the task at the termination
```

```
--                                  of the simulation
--
--         PURPOSE:    Provides the input from the physical keyboard.
--
--*******************************************************************
package Keyboard_PKG is
  function Get_Input return String;
  procedure Clean_Up;
end Keyboard_PKG;

with Classes_PKG;
package Display_PKG is
--*******************************************************************
--          OBJECT:    Display
--  REPRESENTATION:    Subobjects - areas: preview, map, response,
--                                         input, time
--         USED BY:    Console
--            USES:    Preview_Area_PKG, Response_Area_PKG,
--                     Map_Area_PKG, Time_Area_PKG, Input_Area_PKG,
--                     Classes_PKG, Screen_PKG
--      OPERATIONS:    Display_Preview_Message - displays a preview
--                               message in the preview area
--                     Display_Map_Item - displays a single map item
--                               in the map area
--                     Display_Time - displays the time remaining in
--                                    the simulation in the time area
--                     Display_Input - echos the input to the screen
--                     Get_Input - gets input from the keyboard
--                     Display_Roger - displays a "ROGER" message in
--                             the response area
--                     Clean_Up - cleans up the screen. NOTICE: This
--                                operation directly manipulates the
--                                screen object, which is not a
--                                component object of the display, but
--                                is a component of display's component
--                                objects. This is done to prevent the
--                                the invention of a component object
--                                of display called 'Clean_Up_PKG' or
--                                something like that.
--                                of the simulation
--
--         PURPOSE:    Provides the output for the ATC simulation.
--
```

```
--*******************************************************************
  procedure Display_Preview_Message
                  (Next_Message: in Classes_PKG.Preview_Message;
                   Msg_Num: in Classes_PKG.Preview_Message_Count);
  procedure Display_Map_Item(This_Item: in Classes_PKG.Map_Item);
  procedure Display_Time(New_Time: in Classes_PKG.Simulation_Time);
  procedure Display_Input(This_Input: in String);
  procedure Display_Roger;
  procedure Clean_Up;
end Display_PKG;

with Classes_PKG;
package Preview_Area_PKG is
--*******************************************************************
--          OBJECT:   Preview_Area
--  REPRESENTATION:   Sub-objects - screen
--         USED BY:   Display
--            USES:   Screen
--      OPERATIONS:   Display_Preview_Message - displays a preview
--                             message in the preview area
--         PURPOSE:   Displays a preview message in the preview area.
--
--*******************************************************************
  procedure Display_Preview_Message
                  (Next_Message: in Classes_PKG.Preview_Message;
                   Msg_Num: in Classes_PKG.Preview_Message_Count);
end Preview_Area_PKG;

with Screen_PKG;
package body Preview_Area_PKG is
  Area_x:constant integer:=1;
  Area_y:constant integer:=65;
  procedure Display_Preview_Message
                  (Next_Message: in Classes_PKG.Preview_Message;
                   Msg_Num: in Classes_PKG.Preview_Message_Count) is
    x,y:integer;
  begin
     -- The message number determines which line the preview message
     -- is printed on. This prevents messages from being over-
     -- written by new messages.
     x:=integer(Msg_Num);
     y:=Area_y;
     Screen_PKG.Display_Preview_Message(x,y,Next_Message);
```

```
  end Display_Preview_Message;
end Preview_Area_PKG;


package Screen_PKG is
--*****************************************************************
--           OBJECT:    Screen
--  REPRESENTATION:    Subobjects - none
--          USED BY:    Display, Preview_Area, Map_Area, Time_Area,
--                      Input_Area, Response_Area
--             USES:    ?
--       OPERATIONS:    Display_Preview_Message - displays a preview
--                                message in the preview area
--                      Display_Map_Item - displays a single map item
--                                in the map area
--                      Display_Time - displays the time remaining in
--                                      the simulation in the time area
--                      Display_Input - echos the input to the screen
--                      Display_Response - displays a message in
--                               the response area
--                      Clean_Up - Kills the task upon termination
--                                 of the simulation
--
--          PURPOSE:    Provides the interface to the physical screen.
--
--*****************************************************************
  procedure Display_Preview_Message(x,y:in integer;
                                    Next_Message: in String);
  procedure Display_Map_Item(x,y:in integer;
                             Item: in character);
  procedure Display_Time(x,y,New_Time: in integer);
  procedure Display_Input(x,y: in integer;
                          This_Input: in String);
  procedure Display_Response(x,y: in integer;
                             This_Response:in String);
  procedure Clean_Up;
end Screen_PKG;
```

A.2.0.3 Command Object

```
with Aircraft_Attributes_PKG;
with Classes_PKG;
package Command_PKG is
--*****************************************************************
--           CLASS:    Command
--  REPRESENTATION:    Record
--         USED BY:    ATC
--            USES:    NONE
--      OPERATIONS:    Create_Command - builds a command from an
--                                      input string
--                     Get_ID         - returns the ID of the aircraft
--                                      specified in the passed command.
--                     Is_Status      - returns true if the passed
--                                      command is a status request,
--                                      false otherwise.
--                     Is_Termination - returns true if the passed
--                                      command is a termination
--                                      request, false otherwise.
--                     Is_<command>   - returns true if the passed
--                                      command = <command>, false
--                                      otherwise. There will be
--                                      one of these for each
--                                      different command.
--      EXCEPTIONS:    Invalid_Command- this exception is raised
--                                      when the Direction or
--                                      Amount parts of the command
--                                      are illegal values. The
--                                      exception is propagated to
--                                      the ATC object.
--                     Invalid_Aircraft-this exception is raised
--                                      when an invalid Aircraft_ID
--                                      is detected. The exception
--                                      is propagated to the ATC
--                                      object.
--
--         PURPOSE:    Represents the commands used in the ATC
--                     simulation.
--*****************************************************************
  type  Command  is private;
  function Create_Command (This_String: in string)
                                                  return Command;
```

```
  function Get_ID (This_Command: in Command)
                                   return Aircraft_Attributes_PKG.Aircraft_ID;
  function Is_Status (This_Command: in Command) return boolean;
  function Is_Termination_Request (This_String: in Classes_PKG.Input_String)
                                                return boolean;
  function Is_Clear_to_Land (This_Command: in Command)
                                                return boolean;
  function Is_Turn_Left_45 (This_Command: in Command)
                                                return boolean;

  Invalid_Command   : exception;
  Invalid_Aircraft  : exception;
  private
     -- Command is a record containing the following:
     -- 1. aircraft_ID of the aircraft being commanded
     -- 2. the direction character which determines which
     --    direction the aircraft should go.
     --    L - left
     --    R - right
     --    A - ascend/descend
     -- 3. the amount character which specifies how far the
     --    the aircraft should turn/ascend/descend
     --    0 - clear to land, hold at navaid, continue
     --    1 - 1000'/45 degrees
     --    2 - 2000'/90 degrees
     --    3 - 3000'/135 degrees
     --    4 - 4000'/180 degrees
     --    5 - 5000'/clear for approach
     -- 4. a boolean which flags the command as a status request or
     --    a directive command
     type Command
        is record
              Aircraft_ID   : Aircraft_Attributes_PKG.Aircraft_ID;
              Direction     : character;
              Amount        : character;
              Is_a_Command  : boolean:=true;
        end record;
end Command_PKG;

package body Command_PKG is
  -- This function converts a string into a command
  function Create_Command (This_String: in string)
                                              return Command   is
```

```
subtype Upper_Case is character range 'A'..'Z';
Temp_Command: Command;
Ch          : character;
-- Function to convert lower case characters to upper case.
function upper (Ch: in character) return character is
  subtype Lower_Case is character range 'a'..'z';
begin
  if Ch in Lower_Case then
    return character'val(character'pos(Ch)-character'pos(' '));
  end if;
  return Ch;
end upper;
begin
  -- Check to make sure the aircraft id is valid
  Ch:=upper(This_String(1));
  if not (Ch in Upper_Case) then
     raise Invalid_Aircraft;
  end if;
  -- Check for status message
  if This_String'length = 1 then
     Temp_Command.Is_a_Command:=false;
     Temp_Command.Aircraft_ID := Ch;
  -- Must be a command
  elsif This_String'length = 3 then
     Temp_Command.Is_a_Command:=true;
     Temp_Command.Aircraft_ID := Ch;
     -- Check for valid direction character
     -- If valid, assign it
     Ch:=upper(This_String(2));
     if (Ch='A') or (Ch='L') or (Ch='R') then
       Temp_Command.Direction:=Ch;
     else
       raise Invalid_Command;
     end if;
     -- Check for valid amount character
     -- If valid, assign it
     Ch:=This_String(3);
     if Ch in '0'..'5' then
       Temp_Command.Amount:=Ch;
     else
       raise Invalid_Command;
     end if;
  end if;
```

```
    return Temp_Command;
end Create_Command;

-- This function returns the ID of the aircraft specified in the command
function Get_ID (This_Command: in Command)
                          return   Aircraft_Attributes_PKG.Aircraft_ID is
begin
  return This_Command.Aircraft_ID;
end Get_ID;

-- This function returns true if the command is a status request,
-- false otherwise
function Is_Status (This_Command: in Command) return boolean is
begin
  if not This_Command.Is_a_Command then
     return true;
  else
     return false;
  end if;
end Is_Status;

-- This function returns true if the command is a Clear_to_Land
-- command, false otherwise
function Is_Clear_to_Land (This_Command: in Command)
                                        return boolean        is
begin
  if (This_Command.Direction='A') and (This_Command.Amount='0') then
     return true;
  else
     return false;
  end if;
end Is_Clear_to_Land;

-- This function returns true if the command is a turn left 45
-- degrees command, false otherwise
function Is_Turn_Left_45 (This_Command: in Command)
                                      return boolean        is
begin
  if (This_Command.Direction='L') and (This_Command.Amount='1') then
     return true;
  else
     return false;
  end if;
```

```
  end Is_Turn_Left_45;

  -- This function returns true if the string input at the keyboard is
  -- a termination request, false otherwise
  function Is_Termination_Request (This_String: in Classes_PKG.Input_String)
                                                   return boolean is
  begin
    if This_String = "TER" then
      return true;
    else
      return false;
    end if;
  end Is_Termination_Request;

end Command_PKG;
```

Appendix B. *Validation Package*

This chapter contains the validation package used to validate the research presented in the thesis. Also included are the list of experts consulted and the individual responses of the experts.

B.1 The Package

The validation package consists of a discussion of the concurrency heuristics and a questionaire. The questionaire is reproduced in Figure 5.1. The remainder of this section contains the textual portion of the package.

B.1.1 Heuristics for determining concurrency. Following are four heuristics which designers may use in determining concurrency in object-oriented designs. They are based on the heuristics used in the DARTS[Gomaa 1984], LVM/OOD[Nielsen and Shumate 1989], ADARTS[Gomaa 1989a], and Entity-life Modeling[Sanden 1989] methods.

B.1.1.1 Problem-space concurrency. **An object which models concurrency in the problem environment should be implemented as a task.**

Concurrency in the problem-domain can be determined by identifying behavior patterns, or sequences of events, in which the objects participate. The objects themselves may represent physical entities to which the system interfaces, or logical entities, such as an air traffic control system.

B.1.1.2 Time constraints. **An object whose behavior or operations are constrained by time requirements should probably be a task.**

These may be periodic constraints, such as an operation which must be performed at set intervals, or responsive constraints, such as responding to an interrupt.

B.1.1.3 Computational requirements. **An object whose behavior or operations require substantial computational resources should probably be a task.**

For example, in a satellite communication system, the satellite object may have an operation called Calculate Satellite Coordinates. To do this in real time requires the integration of a ninth-order polynomial. Depending on the resources available, this could be quite time consuming and processor intensive. This operation should be a separate task.

B.1.1.4 Solution-space objects. **An object introduced in the software solution to protect a shared data store, decouple two interacting tasks, or synchronize the behavior of two or more objects should be a task.**

B.1.2 Application of the Heuristics to the ATC Problem The heuristics were applied to an Air Traffic Control (ATC) simulation . This section contains a description of the problem followed by a discussion of the concurrency identified via the heuristics and concludes with a discussion of the overall design of the system. For reference the Booch diagrams and Ada package specifications are also included.

B.1.2.1 ATC Description

Air Traffic Control is a simulation which allows the user to play the part of an air traffic controller in charge of a 15x25 mile area from ground level to 9000 feet. In the area are 10 entry/exit fixes, 2 airports , and 2 navaids. During the simulation, 26 aircraft will become active, and it is the responsibility of the controller to safely direct these aircraft through the airspace.

The controller communicates to the aircraft via the scope, issuing commands and status requests, receiving replies and reports, and noting the position of the aircraft on the map of the control space. The controller issues commands to change heading or altitude, to hold at a navaid, or clear for approach or landing. Each aircraft has a certain amount of fuel left, so the controller must see to it that the aircraft is dispositioned prior to fuel exhaustion. Also, the minimum separation rules must be followed, which state that no two aircraft may pass within three miles of each other at 1000' or less separation. The aircraft must enter and/or exit via one of the ten fixes. If an aircraft attempts to exit through a non-exit fix, a boundary error is generated. The controller may request a status report on each aircraft, which will display all information on the aircraft, including fuel level, which is measured in minutes.

The aircraft can be one of two types, a jet or a prop. The jets travel at 4 miles per minute, while the props travel at 2 miles per minute. This means the screen must updated every 15 seconds for a jet's course to be followed accross the screen.

The controller dispositions aircraft by giving commands which enable the aircraft to take off, land, hold at a navaid, assume a landing approach, turn, or change altitude. Take off is accomplished by ordering the aircraft to assume a certain altitude; there is no 'take off' command as such. Each of the airports has restrictions on heading for takeoff; these restrictions must be observed. Turns and altitude changes are effectively instantaneous, i.e., they are accomplished at the next mile marker. To land, the aircraft must be cleared for landing through the navigational beacon (navaid) assigned to the airport. Since there are two airports, there are two navaids. To land, the controller places the aircraft on a heading for a navaid and issues a clearance for approach command. Once the aircraft reaches the beacon, it automatically assumes the correct heading for the airport. The controller then issues a clearance to land command, and when the aircraft reaches the airport it lands (disappears from the screen). If the controller issues a hold command, the aircraft remains at the navaid until released.

The player initially specifies the length of the game, which may be between 16 and 99 minutes. The same number of aircraft will appear for each game, so the shorter the simulation, the more challenging. In any session, the last 15 minutes will be free of new aircraft. The simulation terminates when all aircraft have been successfully dispositioned, the timer runs out, the player requests termination, or one of three error conditions occurs:

- conflict error - separation rules were violated
- fuel exhaustion
- boundary error - the aircraft attempt to leave the control space via an unauthorized point.

B.1.2.2 ATC Design This section contains a summary of the ATC design in general. The main objects are discussed briefly, the Ada package specifications for the main objecs are listed, and the Booch diagrams for the design are given. sectionConcurrency in ATC

In the ATC simulation, three objects contain concurrency: the ATC object, the Console object, and the Aircraft class.

- ATC. The first and second heuristics were used to identify concurrency in the ATC object. Examining the ATC problem description reveals two separate patterns of behavior. The first is the periodic updating of the ATC display. This is a task under the second heuristic, an object behavior constrained by time. The second pattern of behavior is the asynchronous processing of user-entered commands. The pattern is a follows: the user enters a command, the system responds with a message, and the command is executed. The asynchronous nature of this pattern precludes it being embedded within the periodic update of the display.

- Console. The first heuristic identified two behavior patterns within the console object, one corresponding to input (Keyboard), the other corresponding to output (Screen). These objects happen to model physical devices.

- Aircraft. The first heuristic was used to identify the Aircraft class as concurrent. Although the actual physical airplane need not be modeled (flaps, engines, etc.), the behavior of the aircraft flying throught the airspace is an identifiable behavior pattern, which should be modeled as a task.

Two heuristics were not used. No computationally intensive objects or operations were identified; nor were any concurrent solution-space objects encountered.

B.1.2.3 ATC Design This section contains a summary of the ATC design in general. The main objects are discussed briefly, the Ada package specifications for the main objects are listed, and the Booch diagrams for the design are given.

Main Objects The main objects in the ATC system are ATC, Console, Command, Airspace, and Aircraft.

- ATC. The ATC object is the primary object of the system. It controls the interaction of other objects. As previously mentioned, it has two threads of control, command processing and display update.

- Console. The Console object handles the system I/O.

- Command. The Command class defines the representation of a command, and provides operations to create a command, determine whether a command is a status request or a directive, and identifies which particular command a command variable contains.

- Airspace. The Airspace object represents the airspace, which contains landmarks (navigational beacons, airports, and entry/exit fixes) and aircraft. It tracks the location of the aircraft, determines when proximity errors occur, and supervises the execution of commands.
- Aircraft. The Aircraft class represents aircraft as they pass through the airspace, and contains operations which query the status of the aircraft and change the state of the aircraft.

B.1.2.4 Ada Code

ATC Object

```
with Calendar;
with Text'IO;
use  Text'IO;
with Command'PKG;
use  Command'PKG;
with Console'PKG;
with Classes'PKG;
use  Classes'PKG;
with Aircraft'Attributes'PKG;
procedure ATC is

  Simulation'Length: Classes'PKG.Simulation'Time;
  This'Command: Command'PKG.Command;
  Controller'Input: Classes'PKG.Input'String;
  Time'Expired:exception;
  package Time'IO is new integer'io(Classes'PKG.Simulation'Time);

  task Update'Airspace is
    entry Start(Simulation'Length: in  Classes'PKG.Simulation'Time);
    entry Stop;
  end Update'Airspace;

  use  Calendar;
  task body Update'Airspace is

    Time'Left:Classes'PKG.Simulation'Time;
    Minute'Counter: integer:=1;
    Time'Expired:exception;
    Next'Update:Calendar.Time;
    Update'Interval:duration:=15.0;

  begin
    accept Start(Simulation'Length: in Classes'PKG.Simulation'Time) do
        Time'Left:=Simulation'Length;
    end Start;
    Console'PKG.Display'Time(Time'Left);
    Next'Update:=Calendar.clock;
```

```
    loop
      Next'Update:=Next'Update + Update'Interval;
      delay Next'Update - Calendar.clock;
      -- retrieve airspace updates
      -- display airspace updates
      if Minute'Counter=4 then
         Time'Left:=Time'Left-1;
         Console'PKG.Display'Time(Time'Left);
         if Time'Left=0 then
           raise Time'Expired;
         end if;
         Minute'Counter:=1;
      else
         Minute'Counter:=Minute'Counter+1;
      end if;
      select
        accept Stop;
        exit;
      else
        null;
      end select;
    end loop;
  end Update'Airspace;

begin
  put("Enter the simulation length: ");
  Time'IO.get(Simulation'Length);
  -- Draw'Initial'Map;
  Update'Airspace.Start(Simulation'Length);
  delay 1.0;
  loop
    Controller'Input:=Console'PKG.get'input;
    Console'PKG.Display'Input(Controller'Input);
    If Command'PKG.Is'Termination'Request(Controller'Input) then
       Console'PKG.Display'Input("Terminating simulation.");
       -- Terminate'Simulation;
       Update'Airspace.Stop;
       exit;
    end if;
    begin
      This'Command:= Command'PKG.Create'Command(Controller'Input);
    exception
      when Invalid'Command    =>
        Console'PKG.Display'Input("Invalid command.");
      when Invalid'Aircraft   =>
        Console'PKG.Display'Input("Invalid aircraft.");
      when others             =>
        Console'PKG.Display'Input("Something else went wrong.");
    end;
    if not Command'PKG.Is'Status(This'Command) then
       Console'PKG.Display'Roger;
       -- Execute Command
    else
       -- Get Status
       -- Display Status
      null;
    end if;
    delay 1.0;
```

```
  end loop;
  exception
    when Time'Expired=¿
      put'line("You ran out of time!!!!!");
    when others =¿
      put'line("Something bad went wrong.");

end ATC;
```

Console Object

```
--*****************************************************************
--         OBJECT:    Console
--  REPRESENTATION:   Subobjects - display, keyboard
--         USED BY:   ATC
--           USES:   Display'PKG, Keyboard'PKG, Classes'PKG
--      OPERATIONS:    Display'Preview'Message - displays a preview
--                        message in the preview area
--                 Display'Map'Item - displays a single map item
--                        in the map area
--                 Display'Time - displays the time remaining in
--                              the simulation in the time area
                   Display'Input - echos the input to the screen
--                 Get'Input - gets input from the keyboard
--                 Display'Roger - displays a "ROGER" message in
--                        the response area
--                 Clean'Up - kills all tasks at the termination
--                          of the simulation
--
--         PURPOSE:    Provides the I/O to the ATC simulation.
--
--*****************************************************************
with Classes'PKG;
package Console'PKG is

  procedure Display'Preview'Message
                  (Next'Message: in Classes'PKG.Preview'Message;
                   Msg'Num: in Classes'PKG.Preview'Message'Count);
  procedure Display'Map'Item(This'Item: in Classes'PKG.Map'Item);
  procedure Display'Time(New'Time: in Classes'PKG.Simulation'Time);
  procedure Display'Input(This'Input: in String);
  function Get'Input return String;
  procedure Display'Roger;
  procedure Clean'Up;

end Console'PKG;
```

Command Class

```
--*****************************************************************
--          CLASS:    Command
--  REPRESENTATION:    Record
--         USED BY:   ATC
--           USES:   NONE
--      OPERATIONS:    Create'Command - builds a command from an
```

```
--                      input string
--              Get'ID        - returns the ID of the aircraft
--                      specified in the passed command.
--              Is'Status     - returns true if the passed
--                      command is a status request,
--                      false otherwise.
--              Is'Termination - returns true if the passed
--                      command is a termination
--                      request, false otherwise.
--              Is'¡command¿   - returns true if the passed
--                      command = ¡command¿, false
--                      otherwise. There will be
--                      one of these for each
--                      different command.
--      EXCEPTIONS:   Invalid'Command- this exception is raised
--                      when the Direction or
--                      Amount parts of the command
--                      are illegal values. The
--                      exception is propagated to
--                      the ATC object.
--              Invalid'Aircraft-this exception is raised
--                      when an invalid Aircraft'ID
--                      is detected. The exception
--                      is propagated to the ATC
--                      object.
--
--      PURPOSE:   Represents the commands used in the ATC
--              simulation.
--*****************************************************************
with Aircraft'Attributes'PKG;
with Classes'PKG;
package Command'PKG is

  type  Command  is private;
  function Create'Command (This'String: in string)
                                  return Command;
  function Get'ID (This'Command: in Command)
                          return Aircraft'Attributes'PKG.Aircraft'ID;
  function Is'Status (This'Command: in Command) return boolean;
  function Is'Termination'Request (This'String: in Classes'PKG.Input'String)
                                  return boolean;
  function Is'Clear'to'Land (This'Command: in Command)
                                  return boolean;
  function Is'Turn'Left'45 (This'Command: in Command)
                                  return boolean;
  Invalid'Command   : exception;
  Invalid'Aircraft  : exception;

  private
     -- Command is a record containing the following:
     -- 1. aircraft'ID of the aircraft being commanded
     -- 2. the direction character which determines which
          direction the aircraft should go.
     --   L - left
     --   R - right
     --   A - ascend/descend
     -- 3. the amount character which specifies how far the
     --   the aircraft should turn/ascend/descend
```

```
-   0 - clear to land, hold at navaid, continue
-   1 - 1000'/45 degrees
-   2 - 2000'/90 degrees
-   3 - 3000'/135 degrees
-   4 - 4000'/180 degrees
-   5 - 5000'/clear for approach
- 4. a boolean which flags the command as a status request or
-   a directive command
type Command
   is record
        Aircraft'ID  : Aircraft'Attributes'PKG.Aircraft'ID;
        Direction    : character;
        Amount       : character;
        Is'a'Command : boolean:=true;
   end record;

end Command'PKG;
```

Airspace Object

```
with Aircraft'Attributes'PKG;
with Command;
with Landmark'PKG;
with Classes'PKG;
package Airspace'PKG is
_*********************************************************************
-         CLASS:   Airspace
- REPRESENTATION:   none
-       USED BY:   ATC
-          USES:   Command, Landmark, Classes, Aircraft Attributes
-    OPERATIONS:   Initialize'Airspace - sets the location of all
-                           landmarks in the airspace and
-                           passes it back to ATC for
-                           display
-                  Update'Airspace  -  gets the position updates
-                           of the aircraft, checks for
-                           errors, and passes the updates
-                           back to ATC
-                  Execute'Command  -  performs the specified command
-                           on the specified aircraft
-                  Is'Done          -  returns true if 26 aircraft have
-                           been dispositioned
-                  Get'Landmark'Location - returns the location of the
-                           specified landmark.
-                           based on the heading, speed, etc.
-
- PURPOSE: This class represents the airspace.
_*********************************************************************

  package Update'Record'List is new ???????(Update'Record'PKG.Update'Record);
  procedure Initialize'Airspace (Update'List: out Update'Record'List);
  procedure Update'Airspace (Update'List: out Update'Record'List);
  procedure Execute'Command (This'Command: in Command'PKG.Command;
                      This'Aircraft:Aircraft'Attributes'PKG.Aircraft'ID);
  function Is'Done return Boolean;
  function Get'Landmark'Location(This'Landmark: in Landmark'PKG.Landmark)
                      return Classes'PKG.Airspace'Position;
```

```
end Airspace'PKG;
```

Aircraft Class

```
with Aircraft'Position'PKG;
with Flight'Plan'PKG;
with Aircraft'Attributes'PKG;
package Aircraft'PKG is
--*****************************************************************
--       CLASS:   Aircraft
-- REPRESENTATION:   record
--       USED BY:   Airspace
--          USES:   Aircraft'Attributes, Flight'Plan, Aircraft'Position
--    OPERATIONS:   Get'ID       - returns the ID of the aircraft
--                  Get'Source   - returns the source of the
--                                 aircraft
--                  Get'Destination - returns the destination of the
--                                 aircraft
--                  Get'ETA      - returns the ETA of the aircraft,
--                                 the time the aircraft will appear
--                                 on the display.
--                  Get'Class    - returns the class of aircraft,
--                                 whether it is a jet or prop.
--                  Get'Heading  - returns the heading of the aircraft
--                  Get'Fuel     - returns the fuel level of the
--                                 aircraft
--                  Get'Position - returns the 3 dimensional position
--                                 of the aircraft
--                  Get'Altitude - returns the altitude of the aircraft
--
--                  Set'ID       - assigns an ID to the aircraft
--                  Set'Flight'Plan - assigns a flight plan to the
--                                 aircraft
--                  Set'Class    - assigns a class to the aircraft
--                  Set'Heading  - assigns a heading to the aircraft
--                  Set'Altitude - assigns an altitude to the aircraft
--                  Set'Fuel     - assigns a fuel level to the aircraft
--                  Set'Position - assigns a position to the aircraft
--                  Take'Off     - sets the Take'Off flag to true
--                  Hold'at'Navaid- sets the Hold'at'Navaid flag to true
--                  Clear'for'Approach - sets the Clear'for'Approach flag
--                                 to true
--                  Clear'for'Landing - sets the Clear'for'Landing flag
--                                 to true
--                  Continue'Straight - does nothing
--                  Update'Position - sets the new position of the aircraft
--                                 based on the heading, speed, etc.
--
-- PURPOSE: This class represents an aircraft in the airspace
--*****************************************************************

  type Aircraft is private;
  function Get'ID (This'Plan: in Aircraft)
                        return Aircraft'Attributes'PKG.Aircraft'ID;
  function Get'Source (This'Plan: in Aircraft)
                        return Aircraft'Attributes'PKG.Source;
```

```
function Get'Destination (This'Plan: in Aircraft)
                    return Aircraft'Attributes'PKG.Destination;
function Get'ETA (This'Plan: in Aircraft)
                    return Aircraft'Attributes'PKG.ETA'Type;
function Get'Class (This'Plan: in Aircraft)
                    return Aircraft'Attributes'PKG.Class;
function Get'Heading (This'Plan: in Aircraft)
                    return Aircraft'Attributes'PKG.Heading'Type;
function Get'Fuel (This'Plan: in Aircraft)
                    return Aircraft'Attributes'PKG.Fuel;
function Get'Position (This'Plan: in Aircraft)
                    return Aircraft'Position'PKG.Aircraft'Position;
function Get'Altitude (This'Plan: in Aircraft)
                    return Aircraft'Attributes'PKG.Altitude'Type;
procedure Set'ID (This'ID: in Aircraft'Attributes'PKG.Aircraft'ID;
          This'Plane: out Aircraft);
procedure Set'Flight'Plan (This'Src: in Aircraft'Attributes'PKG.Source;
                This'DST: in Aircraft'Attributes'PKG.Destination;
                This'ETA: in Aircraft'Attributes'PKG.ETA'Type;
                This'Plane : out Aircraft);
procedure Set'Class (This'Class: in Aircraft'Attributes'PKG.Class;
                This'Plane : out Aircraft);
procedure Set'Heading (This'Heading: in Aircraft'Attributes'PKG.Heading'Type;
                This'Plane : out Aircraft);
procedure Set'Altitude (This'Altitude: in Aircraft'Attributes'PKG.Altitude'Type;
                This'Plane : out Aircraft);
procedure Set'Fuel (This'Fuel: in Aircraft'Attributes'PKG.Fuel;
                This'Plane : out Aircraft);
procedure Set'Position (This'Position: in Aircraft'Position'PKG.Aircraft'Position;
                This'Plane : out Aircraft);
procedure Take'Off (This'Position: out Aircraft);
procedure Hold'at'Navaid (This'Position: out Aircraft);
procedure Clear'for'Approach (This'Position: out Aircraft);
procedure Clear'for'Landing (This'Position: out Aircraft);
procedure Continue'Straight (This'Position: out Aircraft);
procedure Update'Position (This'Position: out Aircraft);

private
 type Aircraft is
  record
   ID: Aircraft'Attributes'PKG.Aircraft'ID;
   Active: Boolean:=false;
   Flight'Plan: Flight'Plan'PKG.Flight'Plan;
   Class: Aircraft'Attributes'PKG.Class;
   Fuel'Level: Aircraft'Attributes'PKG.Fuel;
   Position: Aircraft'Position'PKG.Aircraft'Position;
   Approach: Boolean:=false;
   Landing : Boolean:=false;
   Hold    : Boolean:=false;
  end record;

end Aircraft'PKG;
```

B.1.2.5 Booch Diagrams Following are the Booch diagrams for the higher levels of the design. The lower level objects and classes are included only when relevant for concurrency.

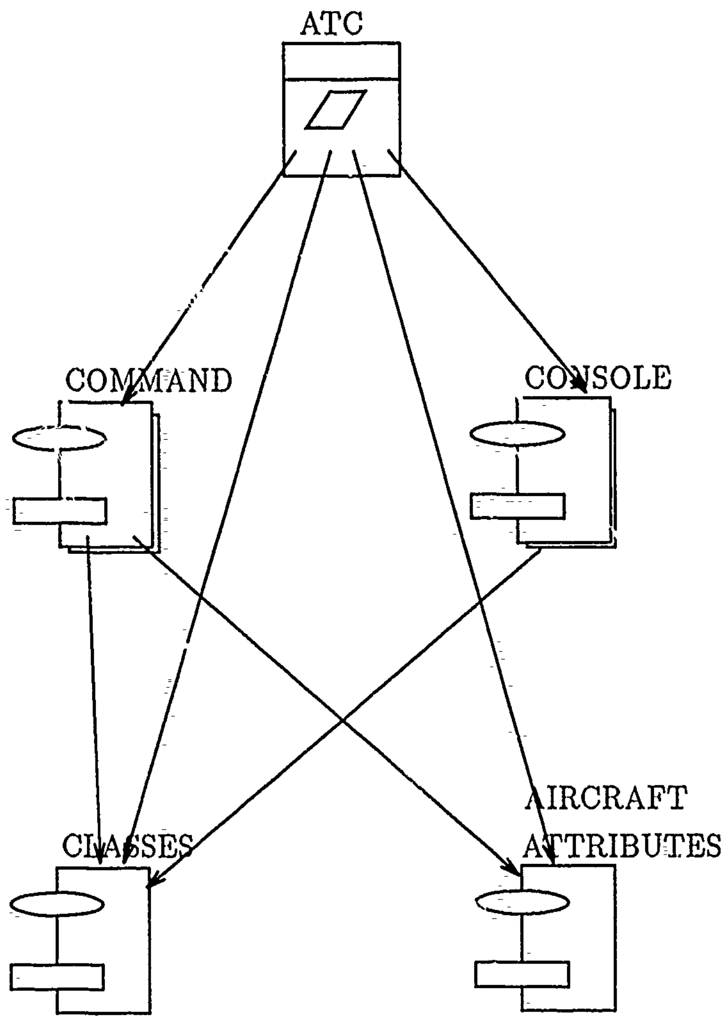


Figure B.1. Top Level Design

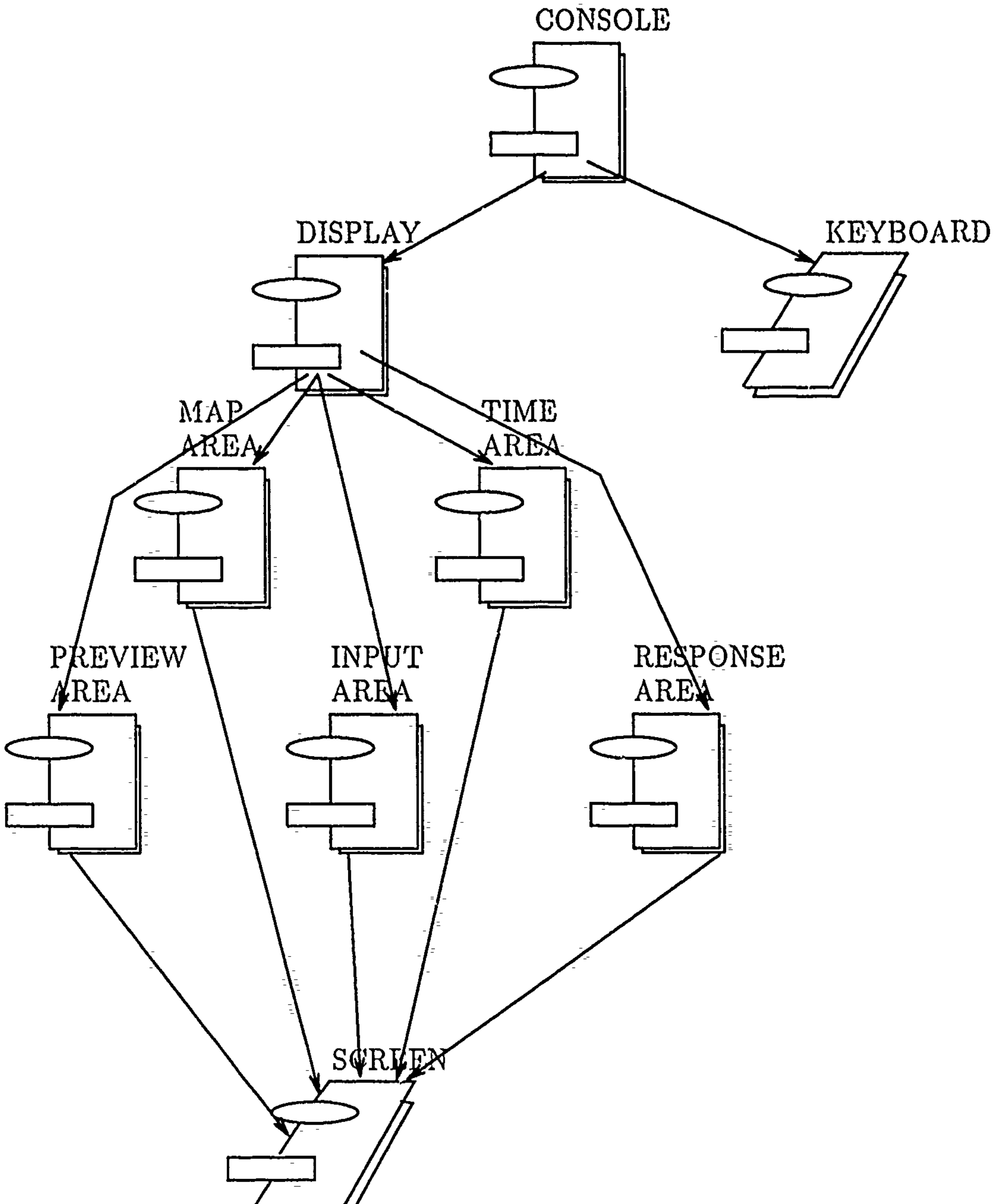


Figure B.2. Console Object Refinement

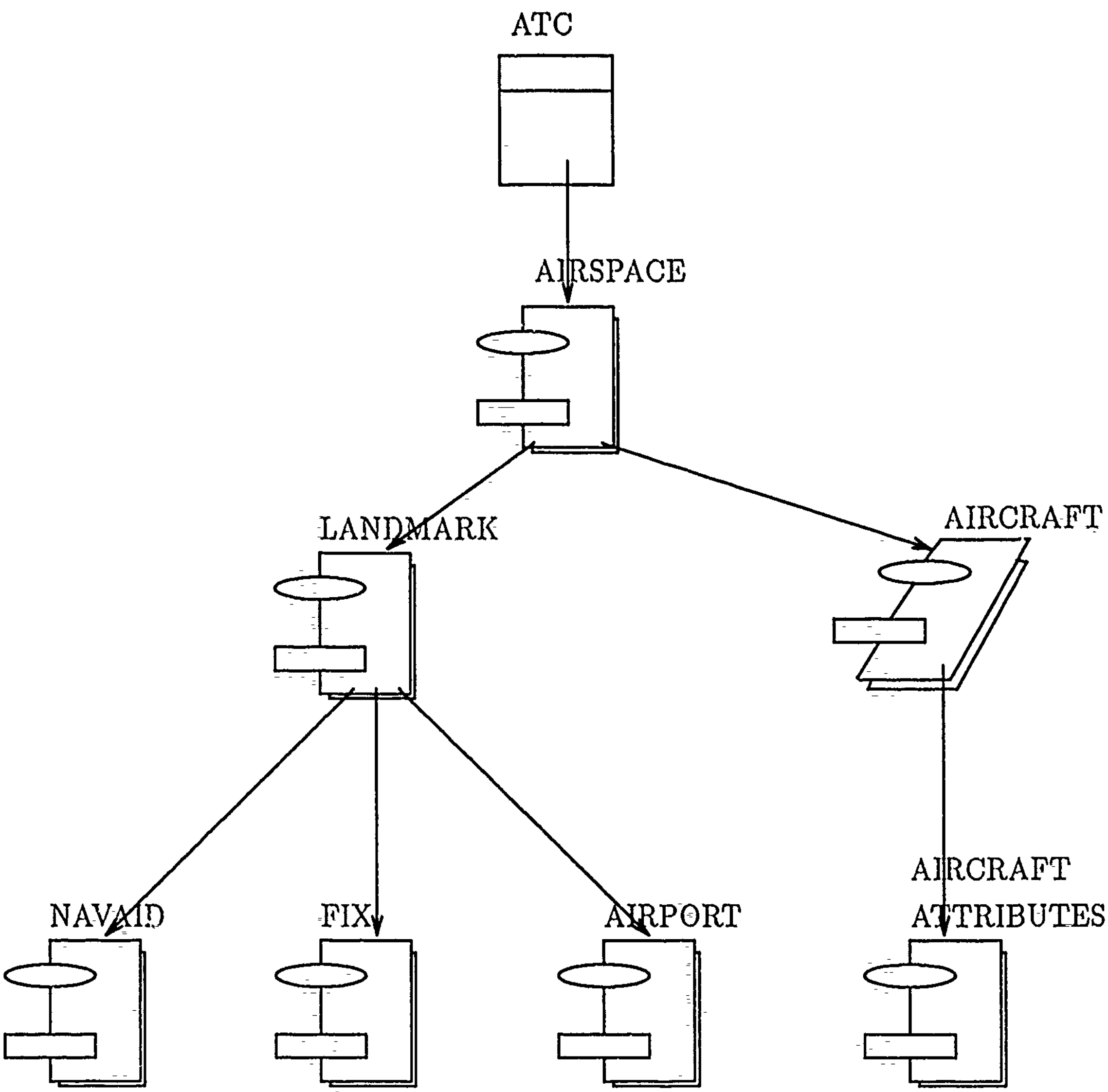


Figure B.3. ATC Object Refinement

B.2 *The Experts*

The following table lists the software engineering experts who participated in evaluating the research contained in this thesis:

NAME	ORGANIZATION
Karyl Adams	Contractor
Capt Paul Hardy	Air Force Institute of Technology (AFIT)
Dr James Howatt	AFIT
Capt Terry Kitchen	AFIT
Dr Patricia Lawlis	AFIT
Capt James Marr	AFIT
Capt Gene Place	AFIT
Dr Bo Sanden	George Mason University
Capt Kelly Spicer	AFIT
Dr Marty Stytz	AFIT
Capt Jay Tevis	AFIT

B.3 *The Responses*

The following pages contain copies of the experts' responses to the questionaire.

4221

Adams + Lawlis

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

The explanation for 1.3 does not make clear the issue that Nielsen & Shumate focus on which is that those computationally expensive algorithms which are not time critical could become low priority tasks in order to manage time performance more appropriately.

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

With this level of detail in the explanation the ~~[illegible]~~ individual entries are of marginal value. The heuristics are good but more detail is necessary to support the designer's decision making process.

Done a nice job in folding some of the Nielson/Shumate guidelines together and still covered the essentials.

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

Can't think of any — you seem to have covered those situations of current concern — but I wouldn't go so far as to say you've covered all possible situations

4. Is there overlap among the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

Possible overlaps with problem-space concurrency, time constraints, and computational requirements — but not in every case, so there is no redundancy here.

Basically agree with this comment Overlap is not detrimental as you have stated the heuristics

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

No inherent inconsistencies

Not if the designer follows the "rules" of any good design — HOWEVER

1.2 Time constraints — provides the latitude to develop temporally cohesive objects which are not particularly robust

1.3 Computational Reqs — can become a "grab bag" again leading to poor cohesion and potentially poor encapsulation of a single object

The key is not that the basic heuristic is weak, it's OK — more guidance, or refinement, to explain how to apply the heuristic is needed to keep the designer on the "straight and narrow"

Hardy

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

4. Is there overlap among the heuristics? Which?

1	1.5	2	3	4	5
NONE			SOME		MANY

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

FROM: Paul R. Hardy, Capt 6 September 1990

SUBJECT: RE: Evaluation of Design Heuristics, 4 Sep 90, Ltr

TO: Capt Ken Baum

Below are comments requested in the subject letter:

Question 1: No additional comment.

Questions 2 and 3: The comments I'm providing stradle issues raised in questions 2 and 3. First, not evident in the write up for evaluation was a mapping from traditional object oriented analysis and design tools (concept map, class specification, etc) to identification of possible tasks. This may be part of a more extensive presentation. This proposed mapping would be useful to the designer in applying the heuristics. Second, since it appears that the dynamic characteristics of an object are the predominant factors in deciding concurrency is there a classification of objects based upon this dynamic behavior which could facilitate identification of a task-oriented object? For instance, an actor object could be a candidate for a Task. (This is just an example.) Have you found it to be true that objects which essential are similar to abstract data types, that is, have operations which change state values (boolean, numerical, etc) and inspector operations for state values, do not need to be tasks? As opposed to objects that change the state of the system, physical or logical, which map into task?

Question 4: It is probaby important to include "Time Constraints" as a characteristic of a candidate task object. This attribute can be overlooked. I would tend to believe, though, that time restrictions are an attribute of a physical or logical entity, for example, ATC must update the airspace every few clock cycles. If not an attribute of the object, most likely, computationally complex processing is the driving determiner. In either case, time constraints may be implicitly embedded within the other heuristics. (I say may be because these are just comments and I don't have to support any issues I raise!)

Question 5: No comment.

Questions on application of heuristics:

In application of the heuristics to the ATC, it appeared that the Airspace Object decouples the Aircraft and ATC objects. Was I correct in this observation? If so, was there an explicit design decision made to not follow the "Solution-space" heuristic? Are there heuristics for making this sign decision?

Paul R Hardy

Paul R. Hardy, Capt, USAF
AFIT/ENG/GCS-90D

1 Atch
Questionaire

Howatt

1. Arê the heuristics understandable?

1 2 3 4 5
NO FAIRLY YES

See my comments

2. Do the heuristics help the designer in making concurrency decisions?

1 2 3 4 5
NO SOME YES

3. Are there concurrency situations not covered by the heuristics? Which?

1 2 3 4 5
NONE SOME MANY

Can't think of any.

4. Is there overlap among the heuristics? Which?

1 2 3 4 5
NONE SOME MANY

The heuristics don't overlap, per se. However, the code which is put in a task may satisfy two or more of the heuristics.

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1 2 3 4 5
NONE SOME MANY

The heuristics do not violate the principles per se; ~~however, the result of applying~~ and, assuming a "good" problem modularization before applying the heuristics, application of the heuristics should not cause violation of the principles

Hwatt

1 Heuristics for determining concurrency.

Following are five heuristics which designers may use in determining concurrency in object-oriented designs. The are based on the heuristics used in the DARTS[1], LVM/OOD[3] ADARTS[2], and Entity-life Modeling[4] methods.

1.1 Problem-space concurrency.

An object which ~~models~~ exhibits concurrency in the problem environment should be implemented as a task.

Concurrency in the problem-domain can be determined by identifying behavior patterns, or sequences of events, in which the objects participate. The objects themselves may represent physical entities to which the system interfaces, or logical entities, such as an air traffic conrol system.

1.2 Time constraints.

An object whose behavior or operations are constrained by time requirements should probably be a task.

These may be periodic constraints, such as an operation which must be performed at set intervals, or responsive constraints, such as responding to an interrupt.

1.3 Computational requirements.

An object whose behavior or operations require substantial computational resources should probably be a task. Assuming other computations must be done at about the same time?

For example, in a satellite communication system, the satellite object may have an operation called Calculate Satellite Coordinates. To do this in real time requires the integration of a ninth-order polynomial. Depending on the resources available, this could be quite time consuming and processor intensive. This operation should be a separate task.

1.4 Solution-space objects.

An object introduced in the software solution to protect a shared data store, decouple two interacting tasks, or synchronize the behavior of two or more objects should be a task.

Since I don't claim to be an expert in anything really, I'm kitchey answering the below to the best of my knowledge.

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

2, 3, 4 - easy.
1 - little vague, but probably unavoidably so.

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

There may or may not be; thus, I don't feel confident to circle "1".

4. Is there overlap among the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

Theoretically, I believe each could overlap with the others except for "1" and "4".

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

I wish I could answer this one intelligently. I don't think they violate any, but I don't feel confident enough to give it a "1".

(OVER)

Ken,

9/8

Here are some comments on the questionnaire:

1) Sentence 1 ⇒ "4" or "5" heuristics. In my current state of mind, this threw me for a loop right away. Am I missing a page perhaps?

2) You may want to briefly define what you consider to be the "line in the sand" between problem space and solution space. Some authors treat these terms differently.

3) You explained in ~~Section~~ 2.2 how the 3 objects ~~contain~~ concurrency. For "completeness", consider explaining why concurrency is not found in the other objects. In this example, it may be obvious, but it helps to understand your example better by explaining why an object does not exhibit concurrency.

4) Does your heuristic require that the OOD be completed first? I wasn't sure ~~[illegible]~~ if the heuristics were applied during the OOD process or strictly after.

Terry

Marr

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

DEPENDS HOW THE HEURISTICS ARE USED AND AT WHAT STAGE OF DESIGN. DO YOU HAVE ADDITIONAL GUIDANCE AS TO HOW THESE ARE USED? IE. I RECALL DARTS REQUIRED AN INITIAL DATA FLOW DIAGRAM BEFORE THE "CONCURRENT" BOXES WERE DRAWN....

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

AGAIN, DRAWING ON MY DARTS EXPOSURE (POWER OF SUGGESTION!) GOMAA INCLUDED I/O DEPENDANCY. I SUPPOSE THAT MIGHT FALL UNDER ONE OF YOUR OTHER CATEGORIES BUT SINCE IT IS EASILY RECOGNIZED, PERHAPS IT SHOULD BE MENTIONED EXPLICITLY. OVERALL I THINK YOUR 4 HEURISTICS CAN JUST ABOUT COVER ANY SITUATION.

4. Is there overlap among the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

CLEARLY, AN ENTITY MAY HAVE MULTIPLE REASONS FOR CONTAINING CONCURRENCY. BUT DOES IT REALLY MATTER? IN THE END WE DON'T CARE WHY A TASK WAS CREATED, WE WANT THE HEURISTICS TO HELP US IDENTIFY THE TASKS.

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

IN MY "HUMBLE OPINION" NO THE HEURISTICS HELP IDENTIFY THOSE PROCESSES THAT SHOULD RUN AS AN INDEPENDENT TASK. THE DESIGNER MUST STILL DETERMINE THE BEST WAY TO ENCAPSULATE OBJECTS ETC. IE. IN ATC YOU HAVE 4 TASKS (ACCORDING TO BOOCH DIAGRAMS) WHICH WERE IDENTIFIED BY USING THE HEURISTICS - BUT THE WAY THEY WERE PACKAGED AND GIVEN VISIBILITY WITHIN THE IMPLEMENTATION IS STILL ANOTHER DESIGN ISSUE.

Place

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

Seems like using the wrong heuristic can be counterproductive. By using #1, each plane must have a separate task which could possibly result in excessive context switching. Using #3 or #2, however, on a data structure containing info for all planes (since all planes are updated simultaneously?) would use only one task and would therefore allow more cpu time to be devoted to real work.

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

4. Is there overlap among the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

Application dependendent. Data decomposition can effect choice of heuristics as shown above.

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

SEP. 19 1990

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

4. Is there overlap among the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

Bo Sanden

over

Comments

SEP. 19 1990

1.1 Does the object "model concurrency"? This could perhaps be rephrased to indicate that the object represents a behavior pattern that is concurrent with those of other objects

Is "air traffic control system" a good example? Wouldn't "aircraft" be a better one?

1.2 This is somewhat hard to understand at first. There may be 2 issues: 1) The object that ~~has~~ takes regular action and 2) The object that responds to interrupts

1.3 This seems to hold if the computation can be carried out independently. If another ~~task~~ object is waiting for the result, having 2 different tasks won't help.

1.4 This is true in Ada but not necessarily other environments with built-in semaphores or "monitors".

References: ~~The~~ My CACM paper(s) March and December 89 are better refs than the Tech Report. (The *) book, now called Software Systems Construction can also be referenced if you wish)

*) forthcoming

Sanden

Spicer

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

See my comments in your text.

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

Ada's only method of accepting an interrupt is using a task entry. ∴ All hardware monitors should probably be tasks. This is probably covered under the first heuristic, but maybe this should be more clear. However, heuristic 2 mentions something about responding to interrupts. Which one is for monitoring HW?

4. Is there overlap among the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

As mentioned above.

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

I'm a little confused about H #3. How do tasks help you with a long calculation, except perhaps to:

1. Allow you to interrupt the computation to do other work in the case of uni-processor.
2. Allow computation to be put-off to another processor in the case of multi-processors and tasks map to processors.

For uni-processor, tasks slow things down due to context switching overhead.

Spicer

1 Heuristics for determining concurrency.

Following are five heuristics which designers may use in determining concurrency in object-oriented designs. The are based on the heuristics used in the DARTS[1], LVM/OOD[3] ADARTS[2], and Entity-life Modeling[4] methods.

1.1 Problem-space concurrency.

An object which models concurrency in the problem environment should be implemented as a task.

Concurrency in the problem-domain can be determined by identifying behavior patterns, or sequences of events, in which the objects participate. The objects themselves may represent physical entities to which the system interfaces, or logical entities, such as an air traffic conrol system.

1.2 Time constraints.

An object whose behavior or operations are constrained by time requirements should probably be a task.

These may be periodic constraints, such as an operation which must be performed at set intervals, or responsive constraints, such as responding to an interrupt.

1.3 Computational requirements.

An object whose behavior or operations require substantial computational resources should probably be a task.

For example, in a satellite communication system, the satellite object may have an operation called Calculate Satellite Coordinates. To do this in real time requires the integration of a ninth-order polynomial. Depending on the resources available, this could be quite time consuming and processor intensive. This operation should be a separate task.

1.4 Solution-space objects.

An object introduced in the software solution to protect a shared data store, decouple two interacting tasks, or synchronize the behavior of two or more objects should be a task.

5+3T2

1. Are the heuristics understandable?

1 2 3 4 5

NO FAIRLY YES

I is confusing

2. Do the heuristics help the designer in making concurrency decisions?

1 2 3 4 5

NO SOME YES

3. Are there concurrency situations not covered by the heuristics? Which?

1 2 3 4 5

NONE SOME MANY

none which come to mind

4. Is there overlap among the heuristics? Which?

1 2 3 4 5

NONE SOME MANY

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1 2 3 4 5

NONE SOME MANY

Tevis

1. Are the heuristics understandable?

1	2	3	4	5
NO		FAIRLY		YES

- see notes on that page

2. Do the heuristics help the designer in making concurrency decisions?

1	2	3	4	5
NO		SOME		YES

- Brainstorming is still needed but the heuristics keep your mind on the right track.

3. Are there concurrency situations not covered by the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

- I am sure someone somewhere has got another, but the areas I think of are time, space, and mutually exclusive actions.

4. Is there overlap among the heuristics? Which?

1	2	3	4	5
NONE		SOME		MANY

5. Do the heuristics violate established principles of software engineering (coupling, cohesion, encapsulation, information hiding, etc.)? Which?

1	2	3	4	5
NONE		SOME		MANY

Tevis

1 Heuristics for determining concurrency.

Following are five heuristics which designers may use in determining concurrency in object-oriented designs. The are based on the heuristics used in the DARTS[1], LVM/OOD[3] ADARTS[2], and Entity-life Modeling[4] methods.

are these all the same place?

1.1 Problem-space concurrency.

An object which models concurrency in the problem environment should be implemented as a task.

This is vague

Concurrency in the problem-domain can be determined by identifying behavior patterns, or sequences of events, in which the objects participate. The objects themselves may represent physical entities to which the system interfaces, or logical entities, such as an air traffic conrol system.

I agree

1.2 Time constraints.

An object whose behavior or operations are constrained by time requirements should probably be a task.

These may be periodic constraints, such as an operation which must be performed at set intervals, or responsive constraints, such as responding to an interrupt.

1.3 Computational requirements.

I agree

An object whose behavior or operations require substantial computational resources should probably be a task.

For example, in a satellite communication system, the satellite object may have an operation called Calculate Satellite Coordinates. To do this in real time requires the integration of a ninth-order polynomial. Depending on the resources available, this could be quite time consuming and processor intensive. This operation should be a separate task.

1.4 Solution-space objects.

This name really denotes a lot more than whats listed in the paragraph below.

I agree

An object introduced in the software solution to protect a shared data store, decouple two interacting tasks, or synchronize the behavior of two or more objects should be a task.

Bibliography

1. Booch, Grady. "Object-oriented development," *IEEE Transactions on Software Engineering*, *SE-12*(2) (February 1986).
2. Booch, Grady. *Software Components with Ada*. Menlo Park, CA: Benjamin/Cummings, 1987.
3. Booch, Grady. *Software Engineering with Ada* (Second Edition). Menlo Park, CA: Benjamin/Cummings, 1987.
4. Booch, Grady. *Object-Oriented Design with Applications*. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc., 1991.
5. Boyd, Stowe. "Object-oriented design and PAMELA," *Ada Letters*, *VII*(4) (July, August 1987).
6. Cameron, John R. "An overview of JSD," *IEEE Transactions on Software Engineering*, *SE-12*:222–240 (February 1986).
7. Cherry, George W. *The PAMELA Designer's Handbook*, Volume I,II. Thought Tools, 1986.
8. Fairley, Richard. *Software Engineering Concepts*. New York: McGraw Hill Book Company, 1985.
9. Gomaa, Hassan. "A software design method for real-time systems," *Communi-[illegible]s of the ACM*, *27*(9) (September 1984).
10. [illegible], Hassan. "Structuring Criteria for Real-time System Design." In *Proceedings 11th International Conference on Software Engineering*, pages 290–301, Washington, D.C.: IEEE Computer Society Press, May 1989.
11. Gommaa, Hassan. *Software Design Methods for Real-Time Systems*. SEI Curriculum Module SEI-CM-22-1.0, Carnegie Mellon University: Software Engineering Institute, December 1989.
12. IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. New York: IEEE, 1983.
13. Jackson, Michael. *System Design*. New Jersey: Prentice-Hall International, 1983.
14. Kelly, John C. "A comparison of four design methods for real-time systems." In *Proceedings, Ninth International Conference on Software Engineering*, pages 238–252, IEEE Computer Society, March 1987.
15. L.C. Scannell, et al. *Design Impacts of Using Ada For Real-Time Embedded Systems*. Technical Report AD-B105 227, Bedford, MA: The Mitre Corporation, August 1986. Contract F19628-84-C-0001.

16. Levi, Shem-Tov and Ashok K. Agrawala. *Real Time Programs: Design Implementation and Validation.* Computer Science Technical Report Series CS-TR-1837, University of Maryland, April 1987.

17. Nielsen, Kjell and Ken Shumate. *Designing Large Real-time Systems with Ada.* New York: McGraw-Hill, 1988.

18. Pressman, Roger S. *Software Engineering: A Practitioner's Approach.* New York, NY 10020: McGraw-Hill Book Company, 1987.

19. Sanden, Bo. "Entity-life modeling and structured analysis in real-time software design - a comparison," *CACM*, *32*(12) (December 1989).

20. Sanden, Bo. *Software Construction in Ada.* George Mason University, Va: George Mason University, 1990.

21. Seidewitz, Ed. "General Object-Oriented Software Development: Background and Experience," *Journal of Systems and Software*, *9*:95–108 (1989).

22. Sha, Lui and John B. Goodenough. *Real-Time Scehduling Theory and Ada.* Technical Report CMU/SEI-89-TR-14, Software Engineering Institute, April 1989.

23. Ward, Paul T. "The transformation schema: an extension of the data flow diagram to represent control and timing," *IEEE Transactions on Software Engineering*, *SE-12*:198–210 (February 1986).

24. Ward, Paul T. and Stephen J. Mellor. *Structured Development for Real-time Systems*, Volume 1. New York: Yourdon Press, 1985.

25. Yourdon, E. and L. Constantine. *Structured Design: Fundamentals of A Discipline of Computer Program and Systems Design.* Englewood Cliffs, N.J.: Prentice-Hall, 1979.

Vita

Capt Kenneth D. Baum was born on September 24, 1953 in Gary, Indiana. He attended Hobart High School in Hobart, Indiana, graduating in 1971.

Capt Baum enlisted in the United States Air Force in 1975. He was selected for the Airman Education and Commissioning Program in 1982 and attended the University of Maryland, College Park in College Park, Maryland where he was awarded a Bachelor of Science degree in Computer Science in December, 1984.

Capt Baum then attended Officer's Training School and was commissioned an Air Force officer in June of 1985. He was assigned to the 4th Satellite Communications Squadron, Mobile(Holloman AFB, NM) where he served as Computer Operations Officer. Capt Baum entered the Air Force Institute of Technology, School of Engineering, in May of 1989.

Permanent address: 5455 Cobb Drive
Dayton, Ohio 45431

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE
Determining Concurrency In Object-Oriented Design Of Real-Time Embedded Systems Using Ada

5. FUNDING NUMBERS

6. AUTHOR(S)
Kenneth D. Baum, Capt, 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-01

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*) One of the characteristics of real-time systems is concurrency. Designers of real-time systems have traditionally determined system concurrency at implementation time using the facilities of a cyclic executive. With the advent of programming language constructs for specifying concurrency, determining concurrency at design time has become a possibility. Several design methods, all of which are extensions of either Structured Design or Jackson System Development, provide heuristics to help the designer make concurrency decisions. The object-oriented approach, however, has no corresponding heuristics to aid designers of real-time systems. The purpose of this thesis was to develop heuristics to help designers make concurrency decisions in developing object-oriented designs of real-time systems. This was accomplished by examining existing heuristics from other design methods and applying them to the object-oriented paradigm. Four heuristics were developed, the first of which exploits the potential in object-oriented design to model the problem space. The other three heuristics deal with concurrency which is not necessarily reflected in the problem-space, but must be implemented for practical reasons. The heuristics were validated by a group of software engineering experts.

14. SUBJECT TERMS
Object-Oriented Design, Real-Time Systems, Embedded Systems

15. NUMBER OF PAGES
154

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL