AFIT/GCE/ENG/92D-02

OBJECT-ORIENTED ANALYSIS, DESIGN, AND IMPLEMENTATION OF THE SABER WARGAME

THESIS

David Scott Douglass
First Lieutenant, USAF

AFIT/GCE/ENG/92D-02

AFIT/GCE/ENG/92D-02

OBJECT-ORIENTED ANALYSIS, DESIGN, AND IMPLEMENTATION OF THE SABER WARGAME

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

David Scott Douglass, B.S.E.E

First Lieutenant, USAF

December, 1992

Acknowledgements

There are several people I would like to thank, for without them this thesis would not have been possible. First, I would like to thank my thesis advisor, Major Mark A. Roth for his technical guidance. I also extend thanks to my committee members, Major Eric Christensen and retired Major Michael Garrambone for their expertise and their time spent reviewing my thesis draft. I also would like to thank my counterpart Donald R. Moore for his support and company throughout my thesis effort. Above all, I would like to thank my family, especially my mother and father, for their love.

David Scott Douglass

Table of Contents

Page

List of Figures

List of Tables

AFIT/GCE/ENG/92D-02

Abstract

Saber is a two-sided, air and land war game that simulates decisions made by commanders at the theater-level. It is being developed by the Air Force Institute of Technology for the Air Force Wargaming center at Maxwell AFB, Alabama. Saber models conventional, chemical, and nuclear warfare between aggregated air and land forces. It also protrays the effects of logistics, satellites, weather, terrain, and intelligence which add to the realism of the Saber war game.

The Saber war game has three main components, the preprocessor, which is responsible for scenario development and pre-game acitvities, the simulation, the guts of the war game that provides execution of missions and conflict resolution, and the postprocessor, which provides detailed reports and an animated graphical output of troop movement.

This thesis documents the object-oriented analysis, design and implementation of Saber simulation. During the analysis and design phase, a five step process was used. These steps included identifying the objects along with their attributes and operations and encapsulating them within Ada Packages. During implementation, sound object-oriented principles were used to ensure a system that could be easily understood, modified, and enhanced.

OBJECT-ORIENTED ANALYSIS, DESIGN, AND IMPLEMENTATION OF THE SABER WARGAME

I. Introduction

1.1 Overview

Saber is a new computerized war game being designed and implemented by several students and faculty members at the Air Force Institute of Technology(AFIT). The effort of this thesis was to continue this design and development process, bringing Saber a step closer to its final production. Ultimately, Saber will be delivered to the Air Force Wargaming Center at Maxwell AFB, Alabama, where it will be utilized as an educational tool in the art of Army and Air Force Doctrine.

Saber is a two-sided, air and land war game that simulates decisions made of commanders at the theater-level. The major components simulated include "stochastic attrition between aircraft, ground forces and theater air defenses, formation of aircraft packages, logistics, intelligence, and nuclear and chemical warfare."(8:1)

The Saber war game is composed of three main parts: 1) the pre-processor, 2) the post-processor, and 3) the simulation itself. Specifically, this thesis concentrates on the object-oriented analysis, design and implementation of the simulation.

1.2 Background

The history of Saber can be traced back to another computerized war game, the Theater War Exercise (TWX) or its PC based version, Agile. TWX has been used by the Air Force Wargaming Center since 1977, and it is the role of Saber to replace this outdated model. Although TWX and Agile have served their purpose, they lack several key aspects of modeling todays "real world" combat. One shortfall of these models are their inability of the air battle to realistically affect the land battle. The air battle only resulted in the "slowing down of land units". Considering the recent events of Desert Storm, it is evident that air power plays a much larger role in battle than just a "slowing down of land units".

The first concept of a new land battle was initiated in 1990 by Caption Marlin Ness. The Ada programming language was used. The land battle was modeled using a two dimensional hexagonal grid structure on which land units performed missions such as move, attack, defend, withdraw, and support(12). Each hex is divided into six pie pieces where terrain type features are modeled (vegetation, water, mountains, etc.). Obstacles, such as broken bridges and mine fields are associated with a hex-side and must be negotiated before land units can advance from one hex to the next.

Following Ness, Captain William Mann developed a conceptual model integrating Air Force doctrine into a redefined version of Ness' land battle. This new airland model is now known as Saber. Mann's air battle included adding features such as bases, depots, and the formation of aircraft packages which conduct area, support, and strike missions against enemy targets. The formulas and algorithms used to perform stochastic attrition between forces were also part of Mann's thesis effort. The basic air structure is modeled by placing six layers of "mega hexes"(15) on top of the ground hexes where each "mega hex" encloses a single ground hex and its six surrounding hexes. The first "mega hex" layer represents treetop level; the sixth layer represents space where satellites are modeled.

The overall structure of Saber looks very similar to the typical combat model presented in (8), and is shown in Figure 1.

Since Ness and Mann's work, four additional theses have been devoted to the overall structure of the Saber war game. Three were devoted to further model development, and the fourth to an assessment of the model's fundamental concepts. The three developmental theses were performed by Captains Andrew Horton(7), Gary Klabunde(12), and Chris Sherry(22) and the assessment by Captain David Scagliola(21).

Horton's efforts concentrated on the preprocessor resulting in the design and implementation of a relational database using the Oracle database management system to

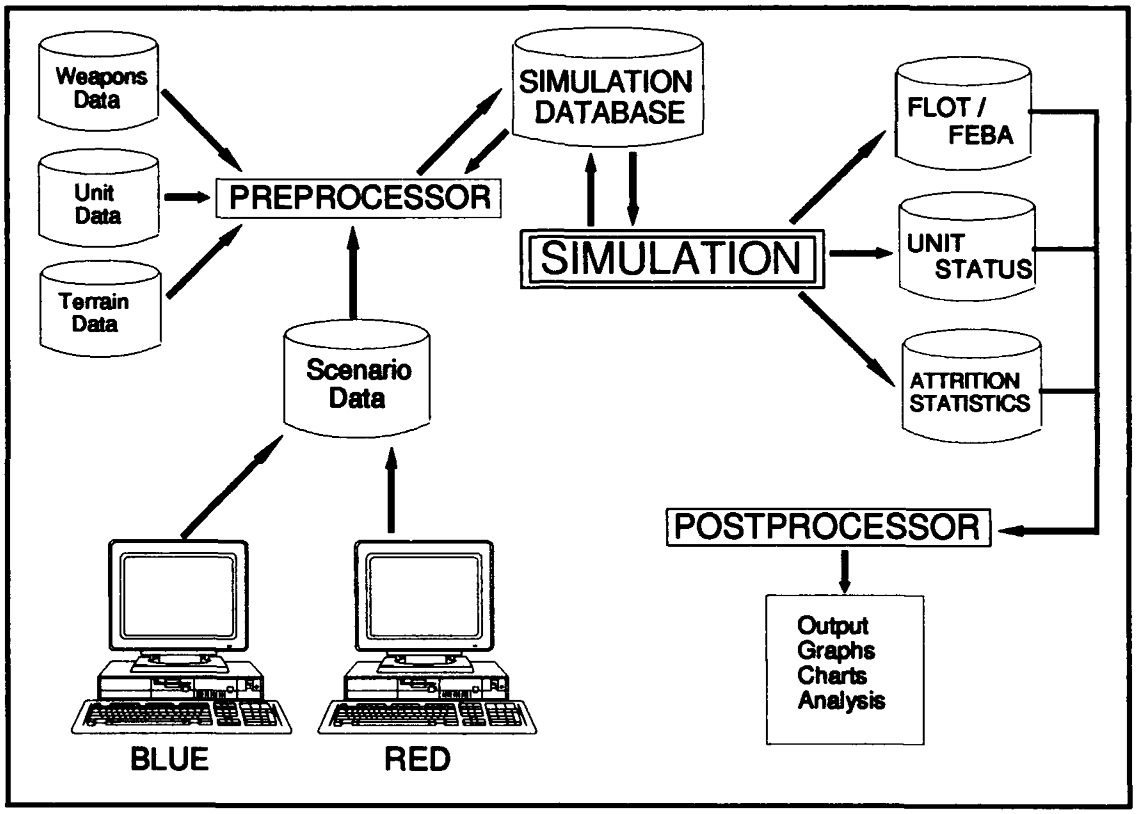

Figure 1. Typical Combat Model (8)

handle the enormous amount of data required to run the simulation. He also layed out the framework for a graphical user interface in which players could enter game missions.

Klabunde's main effort was in the development of the postprocessor. The X Window System along with the Open Software Foundation's (OSF) (12) Motif widget set was used to "provide an animated, graphical postprocessor and report generator for the Saber war game that provided the participants with force status information necessary to plan and execute a theater level air war"(12).

Sherry developed an object-oriented design of the air war using Mann's conceptual model. During these theses efforts, Scagliola conducted an overall assessment of the model. A complete verification and validation was not possible as Saber still had several portions that were not implemented and, at the time, lacked a working simulation.

1.3 The Problem

The Air Force Wargaming Center needs a new and improved computer war game to replace existing outdated models being used as training tools by students of Air War College. Saber is intended to fulfill this need. The underlying concepts of the Saber War game have been formulated, however, several portions remain to be implemented. The two major areas of implementation needed include the integration of the land and air battle into a single simulation and the completion of the user interface. The purpose of this thesis is to design an object-oriented model of the airland battle, implementing the design incorporating any new functionality layed out in any of the previous theses, plus any new functionality added by the current Saber working group. Captain Donald R. Moore conducted his thesis effort in the concurrently worked on the completion of the user interface(16).

1.4 Objectives

As mentioned, the effort of this thesis was to develop the overall airland battle simulation. To produce this result, the following objectives were set forth:

1. Conduct an object-oriented design of the airland battle to determine the objects, and their relationships with one another, that are required for a complete working model of the simulation.
2. Research the data structures currently used in the completed portions of the battle and replace them where appropriate, with data structures giving rise to faster simulation run times and reduced memory requirements.
3. Develop a history file that will be used by Klabunde's graphical user interface. This history file will act as the source in which animation of the last simulation's activities can be presented to the game players.
4. Using the object-oriented design, complete coding of the simulation using object-oriented programming techniques.

1.5 Assumptions

The following assumptions were made during the completion of this thesis effort:

1. The Ada programming language would be used for all software coding, and must be compatible with the Verdix Ada version. It is preferable that the simulation will be compatible with any version of Ada.
2. The simulation must be executed on Sun Sparc Station II or compatible workstation.
3. The simulation must have a maximum run time of less than four hours between input and analysis(21).
4. The conceptual models of both Ness and Mann adequately represent air and land warfare to the level desired for the educational requirement.
5. The algorithms and equations developed for combat processes are correct.

1.6 Methodology

The approach to the development and implementation of the simulation followed these basic steps:

1. Gained an understanding of the basic elements and functionality requirements of the Saber simulation. This included a comprehensive review of the conceptual land and air models designed by Ness and Mann, as well as modifications made during later theses efforts.
2. Completed a literature review of object-oriented design and object-oriented programming. A proper object-oriented design and implementation of that design supplies a number of advantages described in Chapter II.
3. Decided on a methodology in which to implement the object-oriented design. This included selecting data structures in which tradeoffs between run time and memory requirements were considered.
4. Developed a mini scenario in which test cases could be run. The mini scenario was instrumental in speeding up the testing process and allowed for quicker scenario modifications.

5. Created a history file by determining what data requirements were necessary for the graphical user animation postprocessor.

1.7 Materials and Equipment

All development efforts of thesis were conducted on a Sun Sparc II workstation at AFIT. Version 6 of Verdix Ada was the primary development language used, however, the software was also tested with Meridian Ada. Developed software is executable on a Sun Sparc Station II or compatible workstation.

1.8 Thesis Overview

Chapter II is a literature review of object-oriented design methods and the use of these methods in developing simulations with the Ada programming language. Chapter III describes the requirements analysis and design. Chapter IV discusses the modifications and enhancements that were added to Saber. Chapter V presents the implementation details of the design, and finally Chapter VI contains a summary and conclusion as well as recommendations for future work.

II. Literature Review

2.1 Introduction

In order to conduct an object-oriented design of the Saber war game, it was necessary to understand the basic concepts behind the object-oriented methodology. The purpose of this chapter is to review some of the literature on object-oriented modeling and design. Basic object-oriented modeling and design definitions and concepts are discussed. There were several authors that presented varying object-oriented techniques, however, only two are presented. The first is the object modeling technique (OMT) of Rumbaugh, *et al.*(20) and the secend is Booch's five steps to object-oriented analysis and design(2). The advantages of object oriented modeling and design are then discussed, and finally, the ability of the Ada programming language to support object-oriented design is discussed.

2.2 Object-Oriented Modeling and Design

Object-oriented modeling and design is a new way of thinking about problems using models to represent "real world" concepts. The basic entity of an object-oriented system is the object which encapsulates both data structure and behavior. This differs from the traditional view of software systems which are composed of a collections of data that represent some information and a set of procedures that manipulate the data. Object-orientation, as it is integrated into the fundamental components of software development, is to the 1990's what structured programming was to the 1970's and 1980's: a new and important paradigm for improving software construction, maintenance, and use (26).

2.3 Definitions and Concepts

There are a number of key definitions and concepts that must be understood before conducting any type of object-oriented project. Many authors define these key definitions and concepts differently, as well as use different names that represent the same concept. For example, the basic concept of an object is termed "object" by some authors and "instance" by others. For purposes of consistency, the five definitions and concepts as described by Korson and McGregor (13) are presented. These concepts, described in the following sections, are objects, classes, inheritance, polymorphism, and dynamic binding.

2.3.1 Objects. An object is a "concept, abstraction, or thing with crisp boundaries and meaning"(20:21) that makes sense in an application context. Booch defines an object as "something you can do something to. An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable"(3:77). An object's state encompasses all of the properties of the object plus the current values of each of these properties(20:78). The "behavior of an object is how it acts and reacts, in terms of its state changes and message passing"(3:80). Message passing is the basis for communication among objects and it is these messages that initiate object operations(25:204). Objects can be either concrete or conceptual. An example of a concrete object is a file in a file system. A scheduling policy in a multiprocessing operating system is an example of a conceptual object. Objects serve two purposes: "They promote understanding of the real world and provide a practical basis for computer implementation"(20:21).

"Objects are the basic run-time entities in an object-oriented system. Objects take up space in memory and have an associated address"(13:42). Every object is associated with a set of procedures and functions that define operations on that object.

2.3.2 Classes. A class "describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics"(20:22). A single object is simply an instance of a class. "From the point of view of a strongly typed language, a class is a construct for implementing a user-defined type"(13:42).

Ideally, "a class is an implementation of an abstract data type (ADT)"(13:42). This means that the implementation details of the class are private to the class(13:42). "Abstraction gives object-orientation its power and ability to generalize from a few specific cases to a host of similar cases"(20:22). The programming language Ada has the package construct for creating ADTs. The package construct of Ada differs from a class in that a "package encapsulates the type but is not the type itself. It results in a weaker connection between state and behavior as well as the syntactic burden of an additional parameter to most of the package's procedures"(13:42).

2.3.3 Inheritance. Inheritance is "a relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a 'kind of' hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of its superclasses"(3:514). With single inheritance a subclass may inherit data and methods from a single class as well as adding or subtracting behavior on its own. Multiple inheritance refers to the ability of a subclass to acquire data and methods from more than one class. Multiple inheritance is useful in building composite behavior from more than one branch of a class hierarchy(26:34). "Inheritance is the most promising concept we have to help us realize the goal of constructing software systems from reusable parts rather than hand coding every system from scratch"(13:43).

2.3.4 Polymorphism. Polymorphism is "a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways"(3:517). In simpler terms, it is the ability of the objects to take on more than one form. "In an object-oriented language, a polymorphic reference is one that can, over time, refer to instances of more than one class"(13:45). Because of this ability, a polymorphic reference has both a dynamic and static type associated with it. The dynamic type of a polymorphic reference may change from instant to instant during the program execution and the static type is determined from the declaration of the entity in the program text(13:45).

2.3.5 Dynamic Binding. Dynamic binding "denotes the association of a name (such as a variable declaration) with a class; dynamic binding is a binding in which the name/class association is not made until the object designated by the name is created (at execution time)"(3:513). "Dynamic binding means the code associated with a given procedure call is not known until the moment of the call at runtime"(13:46). "Dynamic binding is associated with polymorphism and inheritance in that a procedure call associated with a polymorphism reference may depend on the dynamic type of that reference"(13:46).

2.4 Object-Oriented Methodology

According to (5), experience of the industrial use of object-oriented technology indicates that a disciplined software process is the essential factor determining success. Key components of a software process are systematic analysis and design techniques and recently there have been a number of object-oriented analysis and design techniques developed (4, 9, 20, 23). All of these these techniques are very similar since they use entity-relationship models, state machines and data flow diagrams to build different views of the problem domain, or software in the case of design.

As just mentioned, the key components of the object-oriented methodology are object-oriented analysis and design, each of which are discussed further in the following sections.

2.4.1 Object-Oriented Analysis. Object-Oriented Analysis (OOA), the first step of the object-oriented methodology, is primarily concerned with gaining a precise understanding of the application, then modeling the application in the domain of its intended use. The analysis phase is an iterative process in which the customer and developer work together to produce models that represent the "real world". In order for these models to be successful, they must state what needs to be accomplished, without specifying any restriction as to how it is to be accomplished, and without diving into any implementation details. Other requirements of these models are for them to be:

- *Unambiguous* - A model should have a single discernible meaning. Ambiguity leads to confusion and can cause the "wrong" problem being analyzed and solved.
- *Abstract* - A model should not be cluttered with unnecessary details that can prejudice the design and implementation phases.

- *Consistent* - It should be possible to check if different models of the same system conflict. Inconsistency causes the same problems as ambiguity.

Rumbaugh, *et al.* uses a method called the Object Modeling Technique (OMT) in which the analysis phase is broken down into six steps. The following sections are a summary of the material in (20) except where indicated.

2.4.1.1 Step 1 - Statement of Requirements. Before anything can be developed the requirements of the system must be stated, usually in the form of a problem statement. For the Department of Defense projects, the problem statement may come from an A-Spec, B-Spec, Statement of Work, Request for Proposal, System Specification, and a number of other sources(24:62). The problem statement should scope the problem by stating what is needed, describing the context in which the application will be used, listing any assumptions and stating any specific performance needs. The problem statement should not attempt to give a general approach to solving the problem, such as specific algorithms or data structures to use, architecture, or any optimizations. These should be left to the design and implementation phases.

2.4.1.2 Step 2 - Construction of Object Models. Once the requirements are understood, the next step is to construct an object model also known as a entity-relationship model. The purpose of the object model is to show the static data structure of the real-world system and to partition the system into workable pieces. It describes real-world object classes and their relationships with each other. These relationships are shown by drawing arcs between the object classes. The cardinality of the relationship can also be specified. The OMT technique uses an empty circle to indicate a zero or one relationship while a filled in circle denotes a zero or many relationship. Methods for constructing the object model vary slightly from author to author, but most have the basic steps as described by Rumbaugh, *et al.* The steps are:

- *Identify objects and classes.* During this first step all the relevant object classes are extracted from the application domain. Relevant object classes include physical entities such as aircraft and tanks, as well as concepts, such as trajectories. The object classes can usually be found by "extracting all nouns, pronouns, and noun clauses"(9) from the requirements statement.

- *Prepare a data dictionary.* Words often do not have precise meanings, therefore a data dictionary should be prepared describing all of the modeled entities to include object classes, associations, attributes, and operations.
- *Identify associations (including aggregations) between objects.* An association exists when two or more classes depend on each other, or when one class references another. The associations can usually be extracted from the stative verbs or verb phrases of the problem statement.
- *Identity attributes of objects and links.* The next step is to identify the attributes of an object. Attributes are properties of an object and can be found in the problem statement as nouns followed by possessive phrases, such as "maximum speed of the aircraft". Many times, attributes are not fully described in the problem statement. In this case, the developers must rely on their knowledge of the application domain to extract the attributes.
- *Organize and simplify object classes using inheritance.* The next step is to organize classes to share a common structure. This is done by using inheritance. "Inheritance can be added in two directions: by generalizing common aspects of existing classes into a superclass (bottom up) or by refining existing classes into specialized subclasses (top down)"(20:163).
- *Iterate and refine the model.* Rarely will the object model be correct after a single pass. Because the complete software development process is an iterative process, changes are inevitable, requiring modifications to the object model.
- *Group classes into modules.* The last step for constructing an object model is to group classes into sheets and modules. Classes are grouped into sheets for purposes of drawing, printing and viewing, while they are grouped into modules that capture some logical subset of the entire model.

2.4.1.3 Step 3 - Construction of Dynamic Models. The purpose of the dynamic model is to show the time-dependent behavior of the system and the objects in it.

The dynamic analysis starts by determining what events will take place in the system. These include any stimuli or responses. Once the events have been determined, a state diagram is developed showing the events and the objects they are associated with. The dynamic model is insignificant for systems that are completely static, such as a database. The major steps in developing the dynamic model are (20):

- *Prepare scenarios of typical interaction.* These prepared scenarios are typical dialogs between the user and the system and are used to get a feel for the system behavior. This allows the designer to gain an understanding of the major interactions, external display formats, and information exchanges.
- *Identify events between objects.* Once the scenarios are prepared they are examined to identify all of the external events. These events include all signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices.
- *Prepare an event trace for each scenario.* From the scenarios generated an event trace is constructed which is an ordered list of events between different objects assigned to columns in a table. By looking down a single column of the table, you can determine all of the events that affect a particular object. The events of the event trace are then used to build the state diagram.
- *Build a state diagram.* The state diagram shows all of the classes that have nontrivial dynamic behavior, along with the events that it receives and sends. Each event in the event trace is modeled by placing an arc between its affected classes. During the interval between any two of these events the system is considered to be in a particular state.
- *Match events between objects to verify consistency.* Once the state diagrams for each class are complete they are check for completeness and consistency. The set of state diagrams for object classes with important dynamic behavior constitute the dynamic model for an application.

2.4.1.4 Step 4 - Construction of Functional Models. The purpose of functional models are to "show how values are computed, without regard for sequencing, decisions, or object structure" and to "show which values depend on which other values and the functions that relate them".(20) In constructing a functional model the following steps are followed:

- *Identify input and output values.* To build the functional model a listing of all the input and output of the system is required. These input and output values are any type of event between the outside world and the system.
- *Build data flow diagrams showing functional dependencies.* A data flow diagram (DFD) is "a graphical technique that depicts information flow and the transforms that are applied as data move from input to output(18)." The DFD is constructed by showing how each output value is derived from input values.
- *Describe functions.* Once the DFD has been constructed a description of each of the functions provides a clear understanding of the relationships between input and output values. These descriptions can be in the form of natural languages, mathematical equations, pseudo code, decision tables, or some other appropriate form.
- *Identify constraints.* The next step is to identify any constraints between objects. Constraints are functional dependencies between objects that are not related by an input-output dependency. Constraints can be on two objects at the same time, between instances of the same object at different times (an invariant), or between instances of different objects at different times.
- *Specify optimization criteria.* During this step, values are maximized, minimized, or otherwise optimized. If conflicts arise between values to be optimized, it needs to be indicated how the conflict will be resolved.

2.4.1.5 Step 5 - Add Operations. Rumbaugh states that the OMT method to object-oriented analysis is much different than traditional programming-based object-oriented methodologies since it places much less emphasis on the defining of operations. This allows for a number of useful operations to be added. These operations can range from a query about an attribute or association of an object in the object model, to events in the dynamic model, to functions in the functional model.

2.4.1.6 Step 6 - Iteration of the Analysis Process. It is almost impossible to do a complete analysis the first time through these steps, therefore it is necessary to reiterate through the previous analysis steps before moving on the the design phase. The iteration process includes looking at the overall analysis model and identifying any inconsistencies and imbalances within and across the models. If possible, these are corrected to obtain a cleaner and more coherent design. Rumbaugh states that there is not a definite line between the analysis and design phases, and not to over iterate the analysis process.

2.4.2 Object-Oriented Design. The object-oriented design (OOD) phase is the second part of the object-oriented methodology. During the analysis phase, emphasis was placed on "what" needed to be accomplished. The design phase places its emphasis on "how" it is to be accomplished by making decisions about how the problem will be solved. The basic goal of the OOD phase is to translate the analysis models into an object-oriented software representation. Many authors describe different steps to construct the object-oriented software. Rumbaugh, *et al.* further divides this phase into a system design step and a object design step. Pressman(18), on the other hand, divides this phase into a preliminary design step and a detailed design step. Regardless of the author, their ultimate intentions are to obtain a sound object-oriented design. Pressman presents six guidelines for establishing this sound design:

1. A design should exhibit a hierarchical organization that makes intelligent use of control among components of software.
2. A design should be modular: that is, the software should be logically partitioned into components that perform specific functions and sub functions.
3. A design should contain distinct and separable representation of data and procedure.
4. A design should lead to modules (e.g. subroutines or procedures) that exhibit independent functional characteristics.
5. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.

6. A design hold be derived using a repeatable method that is driven by information obtained during software requirements analysis.

Pressman also goes on to say that these characteristics are not achieved by chance, but are a result of the application of fundamental design principles, systematic methodology, and thorough review.

2.4.3 Booch's Object-Oriented Design Process. As a comparison to Rumbaugh's object modeling technique, the five steps of Booch's object-oriented design process are described below. This process is widely used in the Ada community.

2.4.3.1 Identify the Objects and Their Attributes. "The first step, identify the objects and their attributes, involves the recognition of the major actors, agents, and servers in the problem space, plus their role in our model of reality(2)." Typically, the objects identified are derived from the nouns, pronouns, and noun phrases used to describe the problem space.

Many of the objects identified may be similar. In this case, a class of objects is established which represent instances of the object. After all the objects have been identified, several iterations of analysis must be accomplished. Freitas *et al.*(6) list four additional iteration steps:

- Once the original list of candidate objects and classes have been identified, the entries that refer to values, methods and other "non-objects" must be eliminated. This includes objects identified that describe the same thing but use different terms.
- The goal of the next iteration is to perform a detailed analysis of each object and object class in order to form a clear and precise picture of the objects that will compose the solution of the problem. Each entry is analyzed to detect if it represents a helpful object or class, and, if so, if it needs to be grouped with other objects or classes or divided into separate objects or classes.

- During this iteration, the results of the previous steps are used and each remaining object is grouped into its own class. Each object must belong to a class, even if that class is anonymous.
- During this last iteration, a semantic analysis of the requirements related to each object and class is performed. This is completed to verify their completeness and correctness.

Once the final list of objects has been identified, the attributes of each object must be determined. The attributes are what define the object or class characterizing its time and space behavior(1). The attributes "are given by the qualifiers of the objects and classes within the informal strategy and by the additional information found in the requirements analysis document(9)."

2.4.3.2 Identify the Operation Suffered By and Required of Each Object. During this step, the semantics of each object and class are established by determining operations that may be performed on or by the object or class(2). The operations usually can be extracted from the verbs, verb phrases, and predicates of the requirements document. These verbs, verb phrases, and predicates are then associated with their respective objects and classes. "It is also at this time that we establish the dynamic behavior of each object by identifying the constraints on time and space that must be observed."(2:2-9)

2.4.3.3 Establish the Visibility of Each Object in Relation to Other Objects. During this step, the visibility of each object in relation to other objects is established. These are the static dependencies among objects and classes, in other words, what objects "see" and are "seen" by a given object(2).

The OOD Handbook for Ada Software written by EVB Software Engineering, Inc.(9) breaks this step into four substeps, listed below:

- This first substep is to decide on how each of the operation will be implemented. In Ada, the program units will be a subprogram, a package, a task, or a generic.

- The second substep is to formally describe the interfaces among the objects, types, and operations. These descriptions can either be textual or graphic.
- The third substep is to create any additional objects, types, and operations which will help implement the strategy. These are items that were not identified as part of the informal strategy, but are required to implement the strategy.
- The fourth and last substep is to produce graphical annotations of the formal strategy. The graphical annotations are used to 1) define the interface among the program units, 2) indicate the order of dependent compilation for the program units, and 3) indicate which information (objects, types, and operations), if any, will be exported by each of the program units. The graphical annotations do not show the underlying implementations of the operations or indicate how an object or type is to be implemented.

2.4.3.4 Establish the Interface of Each Object. During this step, the interface of each object is established. In Ada, this is accomplished by constructing the specifications of each object. The interface forms the boundary between the outside view and the inside view of an object.(1)

2.4.3.5 Implement Each Object. The fifth and final step, implement each object, "involves choosing a suitable representation for each object or class of objects and implementing the interface from the previous step(1)." This could result in a further decomposition, composition, or both. Sometimes an object will be composed of several subordinate objects(1). In Ada, the object will be implemented using one of the program units as mentioned above; a subprogram, a package, a task, or a generic.

2.5 Advantages of Object-Oriented Techniques

There are many documented advantages to using object-oriented techniques. The following sections describe several of the advantages.

2.5.1 Modularity. "Modularity is the term used to qualify a system of objects where each object has a minimum of interaction with the other objects(11)." This allows a program to be "intellectually manageable". Also, as modification need to be made to an object of a modular system there are little, if at all, repercussions on the other objects of the system.

2.5.2 Resuability. Reusability is enhanced using object-oriented techniques because the concepts encapsulated in a class are provided in the method interfaces. The user needs to only understand only the behavior of the class objects as specified by the method interfaces, without concern about their implementation. From the user's point of of view, the method implementations are contained in a "black box" hidden from view. This allows for libraries of modules to be built. Software developers can then "check out" modules that provide the desired functionality without coding from scratch.

2.5.3 Maintainability. Maintainability is enhanced by using object-oriented techniques because changes in the implementation of a data structure or algorithm (e.g., code within the class implementation) can be localized to the region of code that implements the class or part of the class. There will be no ripple effects due to the changes because the class interface will be preserved. This interface is what forms the basis for "using" the class in terms of the actions that can be performed on the class objects from outside the scope of the class.

2.5.4 Reliability. The reliability of a software system is also enhanced because of the high-level integration that is built into the initial design. The major pieces that make up the system are configured from the beginning and fitted together. This allows for high-level integration testing to be performed before many of the low-level details have been implemented. This contributes to improved reliability.

Korson and McGregor describe other ways in which object-oriented techniques provide support for a good design. These include information hiding, weak coupling, strong cohesion, abstraction, and extensibility. Each of these adds to either the reusability, maintainability, or reliability of the overall object-oriented system.

2.6 *Object-Oriented Programming Languages*

"Object oriented programming languages are languages in which objects can be implemented, together with facilities for ensuring that the implementation of objects can be hidden from the programmer, and facilities that enable a high degree of reusability to take place."(10:48) Obviously, the most natural implementation target for an object-oriented design is an object-oriented language. According to (18:158), "in theory, the creation of objects and the construction of object-oriented software can be accomplished using any conventional programming language. But in practice, support for object-oriented approaches should be built directly into the programming language that will be used to implement an object-oriented design." This is true since object-oriented language constructs are similar to the object-oriented design constructs. Basically, "an object-oriented language supports objects (combining data and operations), polymorphism at run-time, and inheritance."(20:341)

2.7 *Object-Oriented Programming with Ada*

Can Ada be considered an object-oriented language? Dr. R. W. Shore(24) states that "while it is safe to say that Ada supports oriented-object design, it is perhaps more difficult to label Ada as an object-oriented language." Ada supports both data abstraction and discrete objects, however, Rumbaugh, *et al.* believe that since it does not support inheritance it cannot be considered truly object-oriented. Rumbaugh *et al.* go on to say that the main obstacle to the straightforward mapping between the object-oriented design and coding is a result of Ada's strong typing system and rigid procedure pointers.

Although Ada may not be considered a truly object-oriented programming language, there are many demonstrated examples of Ada that exhibit object-oriented programming features (24).

One of the major themes of object-oriented programming is the idea of encapsulation of data representation and method implementation. Strong enforcement of encapsulation is provided by Ada with the use of the package construct. The package construct is composed of a package specfication which provides the external view of the package, and a package body which encapsulates the implementation details of the package. A common style of Ada programming is to implement each object class as a package, however, it is not uncommon to implement several object classes within a single package. The attributes that define the object class as well as the operations that can be performed on the object class should only be accessible through the package specification.

Rumbaugh *et al.* point out two ways to implement inheritance in a language that does not directly support inheritance, such as Ada. The first is to avoid using inheritance at all. Many applications do not require inheritance features, therefore, the object classes can be implemented as simple records. The second way to implement inheritance is to flatten the class hierarachy. During design, inheritance is used as normal, however, during implementation, each concrete object class is expanded as an independent data structure. Each inherited operation must also be reimplemented for each of the concrete object classes. Flattening the hierarchy increases the amount of duplicated code, but the use of Ada's variant record and generic package constructs can reduce the duplication.

2.8 Object-Oriented Design and Simulation

There are many advantages to using object-oriented design and programming techniques when developing simulations. Roberts and Hiem (19) point out three immediate benefits of using the object-oriented approach.

1. The first major benefit of object-oriented systems is the design philosophy they bring to a problem. Rathar than relying on the processes or procedures that drive the system, they focus on the objects that compose the system. By encapsulating the characteristics and methods within the objects, the objects can be viewed as fundamental components of the system. This provides a natural decomposition of a system.

2. The second major benefit is that simulations become extensible. Existing models can form the basis for new ones and existing concepts can be enhanced to handle new systems. Inheritance permits new objects to be defined from existing ones by just

describing the differences. Old models now become reusable because their methods and objects continue to be useful.

3. The third benefit, in side-by-side comparisons of object-oriented programming with procedural programming, there has been a substantial reduction in the size of the resulting code. This reduction in code size means that a single person can manage more complexity. In the simulation of large and complex systems, this benefit can mean that larger and more realistic models are possible without an increase in manpower.

Roberts and Hiem also point out three long term benefits of using the object-oriented approach in developing simulations.

1. The first of these is that objects in most simulations tend to be physical and real. Generally they can by represented pictorially. Therefore, object-oriented simulation models often have a natural pictorial representation and are easily animated. The user can often directly translate his simulation model into an animated simulation without additional conceptual changes.
2. The second long term benefit is that because the objects contain their own functionality, intelligence can be built directly into this functionality using the machinery of artficial intelligence and expert systems.
3. And the third is that objects provide a natural basis for concurrency. The idea would be that each object could be assigned to its own processor and work away until it needed some form of coordination.

2.9 Summary

This chapter discussed object-oriented design. Several of the key concepts relating to object-oriented technology were defined. An object represents something in the real world and has a state, behavior and identity. Objects that are similar are then placed into classes. The concepts of inheritance, polymorphism and dynamic binding were also presented.

There are several object-oriented analysis and design processes that exist today, two of which were presented in this chapter. Booch's approach which consisted of five steps, and Rumbaugh's object modeling technique which consisted of a total of six steps. Booch's steps included the construction of graphical annotations with Rumbaugh including the construction of object, dynamic, and functional models. Although the terminology was different the underlying principles are the same. There were many similarities between the two processes.

The Ada programming language was also discussed. Ada does exhibit many of the object-oriented features required of an object-oriented language, however, since it does not support inheritance, it cannot be considered a true object-oriented language.

Finally, the short and long term benefits of using object-oriented design techniques in the development of simulations were discussed. Short term benefits included the reduction in code size and the ease with which the simulation could be extended. Long term benifits included the addition intelligence directly into each object and the natural basis in which concurrency can be exercised.

The next chapter describes the analysis and design phase that was accomplished as part of this thesis.

III. Object-Oriented Analysis and Design

3.1 Introduction

This chapter presents the results of the object-oriented analysis and design of the Saber Simulation. Booch's steps to object oriented analysis and Rumbaugh's Object Modeling Technique (OMT) for creating an object model were the foundations on which this analysis and design were based. The results presented in this chapter represent an iterative analysis and design process that was conducted throughout this thesis effort.

3.2 Requirements Analysis

The purpose of the analysis phase was to scope the requirements of the Saber simulation. Because of the magnitude of the simulation, a precise single problem statement was not developed nor does it exist. Instead, the requirements of Saber are located throughout a number of previous theses efforts. To accomplish this initial analysis step, a complete review of the following documents were conducted:

- Captain Ness' thesis,
- Captain Mann's thesis,
- Captain Horton's thesis,
- Captain Klabunde's thesis,
- Captain Sherry's thesis, and
- Captain Scagliolia's thesis.

As a result of this review, an understanding of the Saber simulation's assumptions, performance needs, and functionalities were gained. Emphasis was placed on identifying and understanding the major components of the simulation, each of which are briefly described in the following sections. Although the analysis phase should only be concerned with what needs to be done, there are a few details discussed. This is due to the fact that Saber has been an ongoing thesis effort and several of its data structures have been determined. For example, the playing field is composed of an interlocking structure of hexagons.

3.2.1 Saber's Playing Field. The playing field of the Saber simulation is modeled, similar to many other common war games, as an interlocking structure of hexagons (hexes) as shown in Figure 2. The hexagons are situated with the vertices (points) oriented in an east-west direction (modification from a previous north-south orientation (17)) and represents a distance of twenty-five kilometers (from flat to flat). The hexes are interlocked through a common hex side identification (neighbor id). For example, in Figure 2 the northeast hex side of hex 03 02 would have the same identification as the southwest hex side of hex 04 02. The minimum and maximum hex numbers are considered a "no man's zone" and is shown in Figure 2 with slashed lines. This "no man's zone" serves two purposes. The first is too alleviate some of the complexity of the movement algorithms. The second is to represent bases that are not able to be physically located in the current playing field, however are able to affect battle outcome with aircraft and supply support. Sherry used an example of a Korean scenario where a United States base located in the Phillipines would be able to support warfare in the Korean area, but is not able to be physically located in its proper location. This base is represented by locating it somewhere in the "no man's zone". An important aspect of the "no man's zone" is that bases located there are unable to be targeted.

Three dimensional play is modeled by placing six layers of air hexes on top of the ground hexes, resulting in a total of seven hex layers. Each air hex or "mega hex", a term often used by wargamers, encloses a single ground hex and its six surrounding ground hexes. Figure 3 depicts how these air hexes are oriented in relation to the ground hexes and can be seen outlined in the bolder faced lines. Each layer of air hexes represents a different altitude with the first layer representing treetop level and the sixth layer representing space. Exact level and altitude correspondances can be found in (8).

An identifier "HX", and a ZZ-XX-YY coordinate system is used to label each air and ground hex. The ZZ-coordinate representing the level, the XX-coordinate representing longitude, and the YY-coordinate representing latitude. Although most coordinate systems

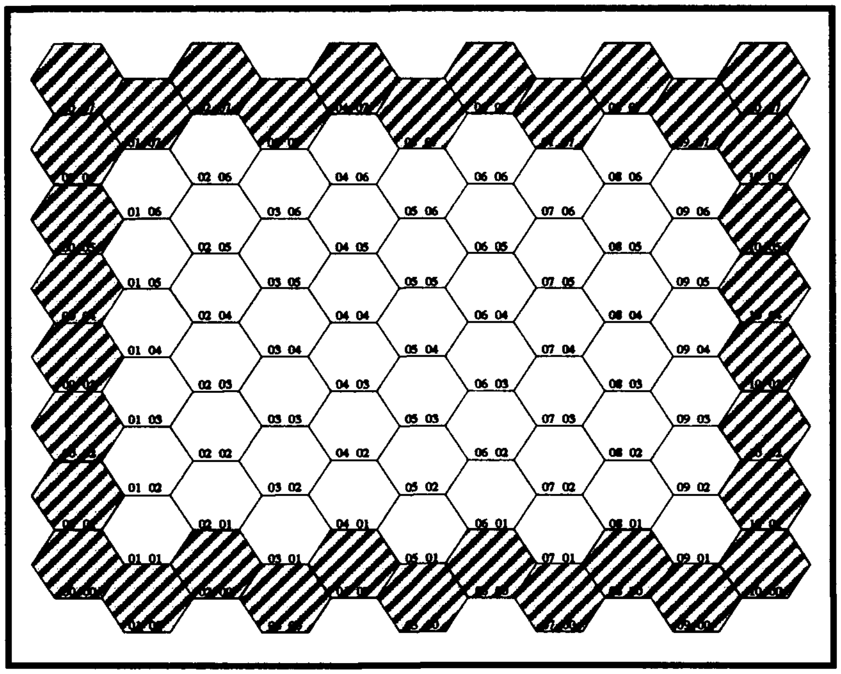

Figure 2. Saber Ground Grid Structure

use an X-Y-Z coordinate system, it was decided that it would be easier for the user if the altitude, or level was the first number (22). By constraining the number of hexes to two digits, the maximum grid size would be a 100 X 100 (numbering starts from 0) grid structure. With the hex sizes representing 25 kilometers across, the maximum playing field would be 250,000 km^2. For theater level warfare, this is sufficient (22).

Linkage between the ground and air hexes are realized through the ZZ-XX-YY numbering scheme. Figure 4 is an example of a single air hex. In this example, ground hex HX010201 is the center hex for its surrounding six ground hexes (each ground hex has a center hex attribute). The air hex HX040201, shown in bold lettering, is then linked through this ground center hex through the XX and YY coordinates. In this example, the air hexes HX020201 through HX070201 are also linked to ground hex HX010201. Visibility within the air and ground hexes is such that anything in a air hex can "see" any of the ground hexes that it encloses and vice versa. Any ground hex can "see" up into the air hexes that enclose it.

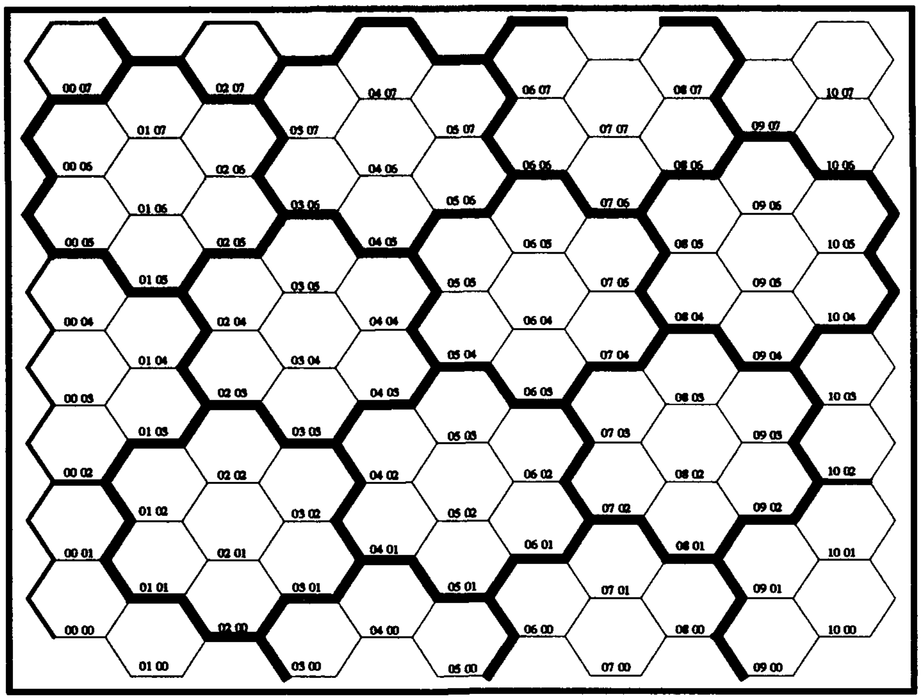
Figure 3. Saber Air Grid Structure

3.2.2 Ground Hex Terrain and Assets. The ground hex is the basic entity which portrays terrain features and in which a number of assets can reside. Each ground hex is divided into six pie pieces as shown in Figure 5 by the the dotted lines. Each pie piece has one of six pie trafficabilities associated with it, which range from excellent to very poor and can represent terrain features such as mountains, hills, deserts, oceans, water, or flat green land.

There are many entities that can be located within a particular ground hex. Land units can be located at the center or border of a ground hex. Obstacles, which include man made objects, impassable objects, mine fields, and bridges are attached to one of the six hex sides, such as the bridge in Figure 5. There is a difficulty value associated with each obstacle which is used to determine the amount of time and resources it will take for a land unit to cross the hex side into the adjacent hex. Obstacles are modified, created and destroyed by engineer, fire support, and air force units. For example, a bridge could be damaged by enemy aircraft and then repaired by a civil engineering unit. Rivers ar.d streams are also associated with a side of a ground hex.

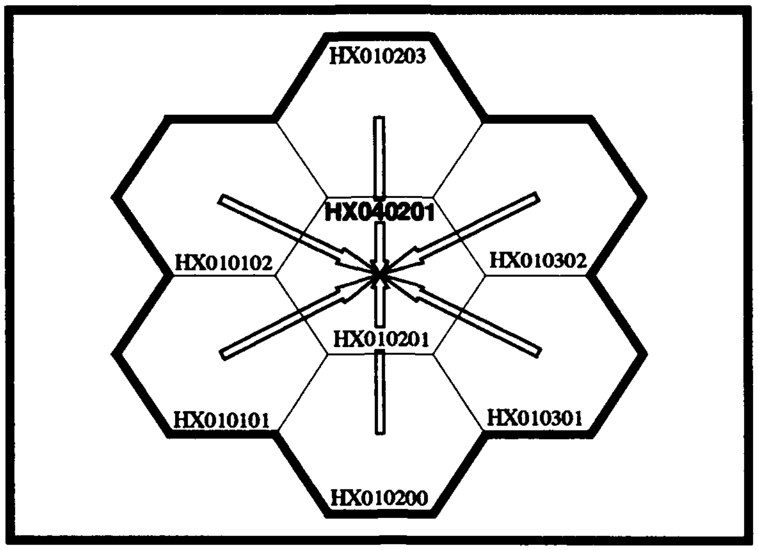

Figure 4. Center Hex

Road pieces, railroad pieces, and pipeline pieces are also modeled on a ground hex and run radially outward from the center of a hex to the center of a hex side. Each road, railroad, and pipeline piece represents a single segment of a larger road, railroad, or pipeline. Figure 5 shows how four road pieces are connected to form a larger road. The addition of road pieces increases the trafficability of the pie piece allowing for increased movement rates. The railroads and pipelines are logistic lines and allow for the flow of supplies and equipment form point to point. Road, railroad, and pipeline pieces are also targetable by enemy forces, and if destroyed, will cause movement rates to decrease, and equipment and supplies to stop flowing until an engineering unit is able to rebuild or repair the damages.

3.2.3 Land Units. The main moving and fighting entity of the land battle is the land unit, which at any given time are either located at a ground hex center or ground hex border. As a land unit receives orders (land unit missions) to move to a new location, the time to traverse the hex is calculated (grid time). After the unit has served its grid time, it

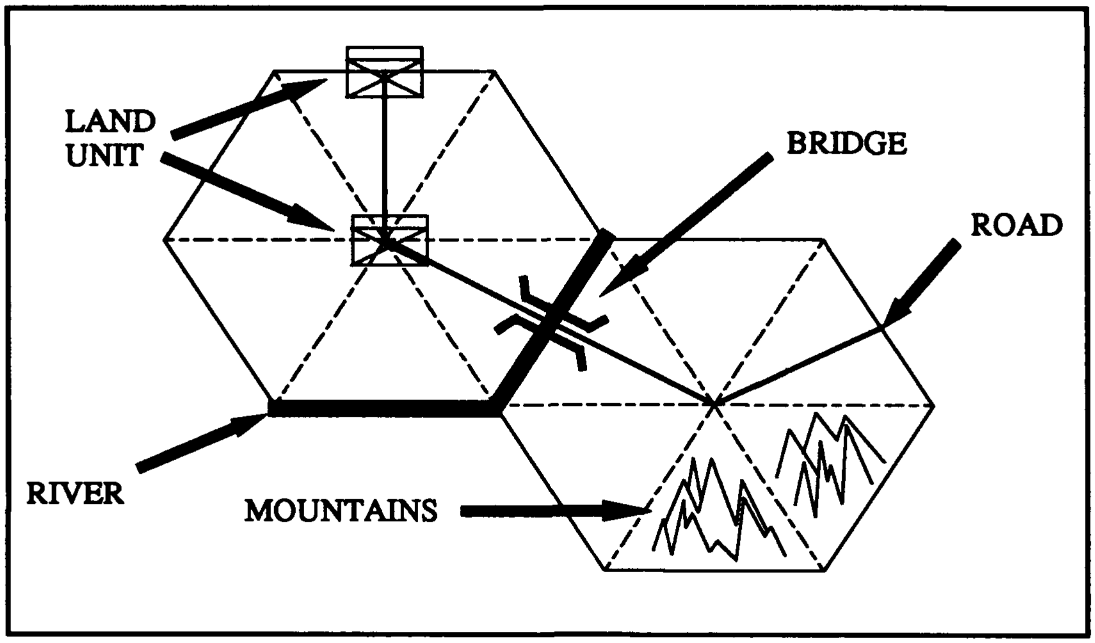

Figure 5. Example Modeled Assets of Hex

then is able to move to the border and must overcome any obstacles as described above. If there are any obstacles at the border, the land unit can either overcome the obstacle at its own rate, or it can receive engineering support to greatly reduce the transition time. After the obstacles have been overcome, the unit is then able to transition into the next hex and a new travel time is calculated. Land units are allowed to continue on their movement missions as long as they do not encounter any attacking enemy land units. If opposing land units do come in contact, and if one of them has a current attack mission, they must engage in combat until one or both units reach a breakpoint (point at which a land unit will withdraw from combat) and withdraws, a unit is completely destroyed, or the attacking unit receives an override mission to terminate the attack. The strengths (combat power) of the fighting forces determines what attrition rates each of the units will suffer.

3.2.4 Airbases and Depots. Airbases and depots are the main supporters of aircraft for the simulation. Airbases maintain aircraft which are used to support sortie missions (or aircraft packages discussed in the following section) against enemy targets. Airbases must also possess the necessary supplies such as fuel and munitions to fly the aircraft. As aircraft perform sorties the airbase resources are used. The depots main purpose is for bulk storage of resources and to provide these resources to the airbases. Supply movement from the depots to the airbases is discussed in section 3.2.6.

3.2.5 Aircraft Packages. The main moving and fighting entity of the air battle is the aircraft package. Aircraft packages are composed of primary, escort or combat air patrol, suppression of enemy air defense (SEAD), electronic combat, and refueling aircraft. To form the aircraft package the airbases are polled in order to determine whether there are enough aircraft to support the mission. Before each polled base can dedicate requested aircraft, several checks are accomplished. This includes making sure the base has enough munitions, fuel and a long enough runway for the aircraft to take off. If the type of munitions are not specified by the user, the aircraft will automatically be loaded with its preferred load. The preferred load is determined by several factors to include the current weather, hardness of the target, type of mission, and the type of warhead to be used. Other checks include making sure the base is able to operate in the current weather and if the target is within range of the aircraft. Only if all these checks are satisfied is the base allowed to dedicate the aircraft to the aircraft package. If a minimum user defined percentage of aircraft are available, the aircraft package is formed, otherwise the package is delayed. Of the formed aircraft packages, the area missions are executed first with the strike missions following. The aircraft package missions start at a designated rendezvous location and proceed to their target location. The aircraft packages are allowed to move towards its target location until it encounters an enemy air defense unit or enemy aircraft. If an aircraft package encounters enemy air defense units, the SEAD aircraft try to take out as many as the surface-to-air missile (SAM) sites as possible. If an aircraft package encounters other enemy aircraft the escort or combat air patrol aircraft will engage. In either case, stochastic attrition will determine loses, and as long as the aircraft package does not drop below its chicken factor (point at which the aircraft package will abort), it will continue towards its target. Once at its target location, SEAD aircraft will take out the enemy air defenses, the primary aircraft will deliver their ordinance and the aircraft package will return to its originating rendezvous hex. Aircraft are then returned to their orignating air bases where they must go through either a turn around time (time it takes for the aircraft to be refueled and loaded with new weapons) or a maintenance cycle (time to repair battle damages or conduct routine preventive maintenence) before they fly another sortie.

3.2.6 Supply Trains. Supply trains are the mechanisms by which supplies are moved during the simulation (the other major method of logistics movement is accomplished through the Saber user interface). Supply trains have the same features of a land unit plus additional features which allow it to move supplies from location to location. Supply trains originate from either a land unit depot or a base depot where they are loaded with the supplies needed to complete their missions. The supply trains then traverse the hexes moving towards their mission locations. Along the route they are succeptable to being targeted by either enemy aircraft or other enemy land units. Once the supply trains reach their mission location, they disburse their supplies and move on to their next mission location. If the supply train has finished its last mission it will do one of two things. If it is a regular supply train (ST), it will wait for further orders. If it is a predetermined supply train (PST), it will return to its originating depot for resupply and repeat the supply missions.

3.2.7 Intelligence. Bases, depots, ground hexes and land units all have an intelligence index associated with them. The intelligence index's purpose is to determine the amount of knowledge that an opponent knows. Over time, each entities intel index is reduced. This simulates less knowledge about the entity by the enemy. As land units come in contact, their intelligence indexes are raised, and as a result the enemy knows more about the land unit. Intelligence indexes are also increased by Army military intelligence units, reconnaissance missions and anti-satellite (ASAT), navigation (GPS) and photographic reconnaissance satellites.

3.2.8 Weather. The Saber playing field is divided into weather zones where each weather zone will have good, fair or poor weather. The actual weather of each zone is determined by a good weather percent, fair weather percent, and the outcome of a random number draw. The weather can play several factors in the simulation, which can

Table 1. Saber Simulation Object Classes

Air Force Mission	Air Hex	Air-to-Air Missile
Air-to-Ground Missile	Airbase	Airbase Component
Aircraft	Aircraft Package	Chemical Weapon
Day	Ground Component	Ground Hex
Hardness	Hex Side	Land Unit
Land Unit Mission	Nuclear Weapon	Obstacle
Period	Pipeline	Pipeline Pieces
Preferred Conventional Load	Preferred Biological Load	Preferred Nuclear Load
Radar	Railroad	Railroad Pieces
River	Road	Road Pieces
Runway	Satellite	Supply Train
Supply Train Mission	Surface-to-Surface Missile	Surface-to-Air Missile
Target	Weather	River

include land unit movement rates being decreased, bases becoming partially or completely inoperable and aircraft packages success rate being reduced.

3.2.9 Clock. The clock mechanism, composed of a day, period and weather period, is what gives the simulation its sense of time. There are twelve periods to a day making each period two hours. This two hour period is the minimum time step that the Saber simulation can perform and was selected because it was determined to be the appropriate amount of time needed for an aircraft package to complete its mission. There are six weather periods each day making each weather period four hours long.

3.3 Object-Oriented Design

The following sections describe the first three steps of Booch's object-oriented design technique as discussed in chapter II. This section also presents the object model of the Saber simulation, constructed using the OMT.

3.3.1 Identify the Objects and Their Attributes. The first step to Booch's object-oriented design and creating an object model was to identify and list all the object classes. This was accomplished by extracting all the relevant noun and noun phrases from the theses mentioned above. Table 1 is the resultant list of the identified object classes needed for the simulation.

Once the objects were identified, the attributes which define an object class needed to be determined. As part of Horton's thesis, he created a listing of objects and their attributes that are required for both the simulation and user interface. Since that time several modifications and enhancements have been made to the simulation. Appendix A contains an updated version of the objects required of the simulation along with the attributes that define each object class.

3.3.2 Identify the Operations Suffered By and Required of Each Object. Once the object classes and their attributes were determined, the associations between objects had to be identified. The operations were determined by extracting verbs and verb phrases from the requirements theses documents listed in Section 3.2. Appendix B is a complete listing of the object classes along with a brief description of their required operations. As a result of these first two steps, an object model was constructed. Figures 6 through 10 show the object model. Rumbaugh's OMT notation is used. Boxed items represent object classes, while the associations between the the object classes are represented by lines connecting the object classes. Because the entire object model could not fit on one page, object classes are redrawn in dashed boxes for ease of identifying associations between objects on separate pages. Associations having links to more than two object classes are modeled using a diamond with lines connecting the related classes. The solid and hollow balls at the ends of the connecting lines represent the multiplicity of the association. Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. A solid ball indicates a many (zero or more) relationship with its connected class while a hollow ball indicates a zero-or-one relationship. For example, in Figure 7 a land unit can have zero, one or many land unit missions, however, a particular mission can be assigned to one land unit. The relationship between a land unit and ground component is an example of a many-to-many relationship. A land unit can have zero or more types of ground components (armored vehicles, tanks, etc), while each ground component type can be part of zero or more land units. The absence of a ball at the end of an association indicates a mandatory relationship. An example of this type of relationship can be seen in Figure 9 between aircraft package and Air Force mission. An aircraft package must have one and only one Air Force mission, while several of the aircraft packages can have the same type of Air Force mission.

Generalization, also known as an "is-a" relationship, is also shown in the object model. Generalization is the relationship between a class and one or more refined versions of itself. The class that is being refined is called the superclass while the refined class or classes are called the subclasses. An example of generalization can be seen in Figure 8 between the superclass preferred weapons load and the three subclasses preferred conventional load, preferred biological load, and preferred nuclear load. Its notation is a triangle connecting the superclass and the subclasses. In this example, each of the subclasses inherit all of the attributes and operations of the superclass. The subclasses' own unique attributes and operations are what sets them apart from the other subclasses. The terms generalization and inheritance are often used interchangeably.

3.4 Establish the Visibility of Each Object in Relation to Other Objects.

During this step of the design process, the visibility of each object was determined. This was determined mainly from the constructed object diagram. For example, land units object can have different types of weapons. Therefore, the land units object "sees" the weapons object. The "with" statements in the specification and body of a package is Ada's way of showing what objects are visible to other objects. Appendix C is a complete listing of all the objects, indicating the visibility of each object with one another.

3.5 Establish the Interface of Each Object.

During this fourth step of Booch's five steps, the interfaces of each object were written. This correlates to developing the Ada package specifications of each object. Most of the interface procedures and functions matched with the operations identified for each of the objects, however, throughout the development process, there were additional functions and procedures required that were not originally anticipated. The code contains the specifications of each object class.

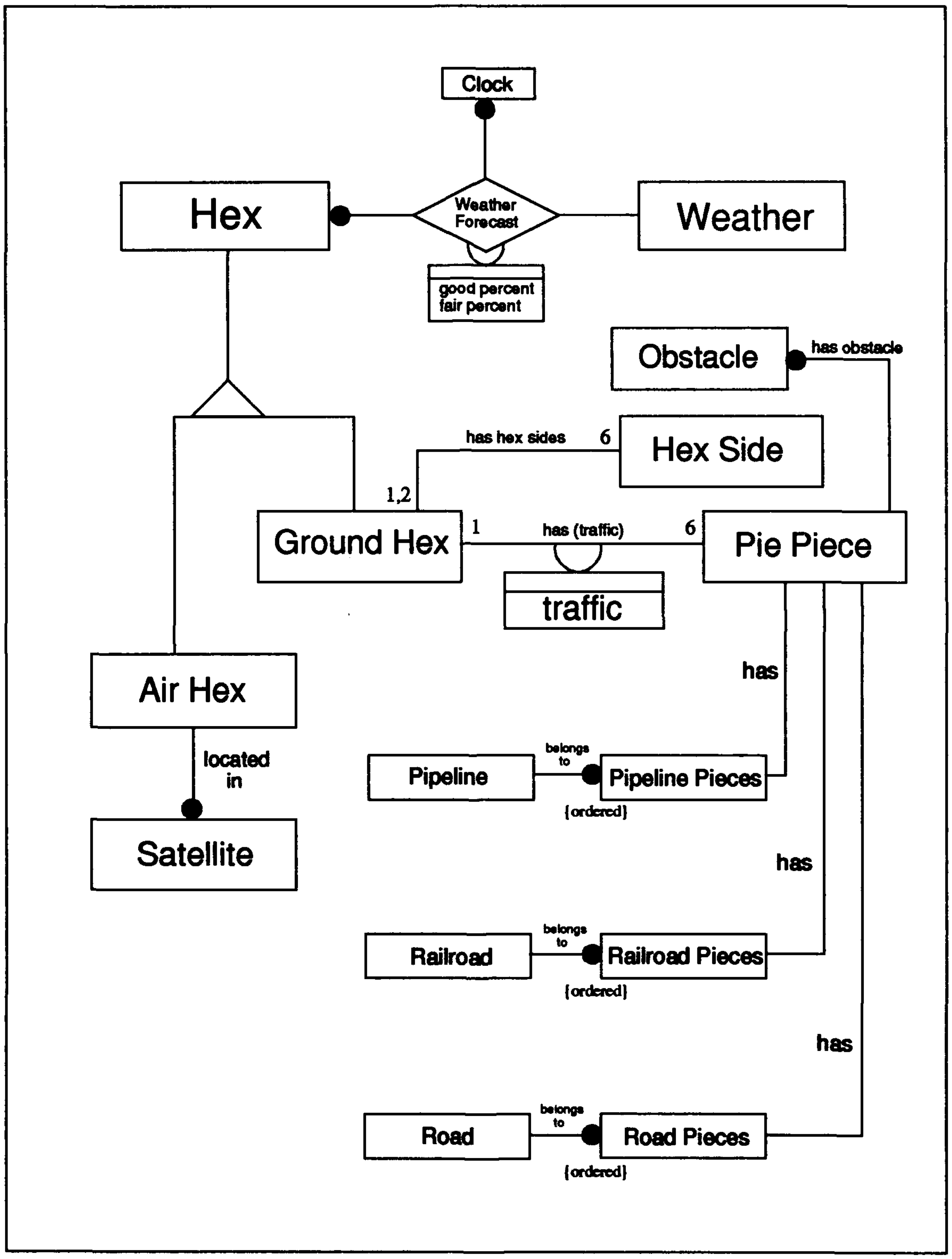

Figure 6. Object Model

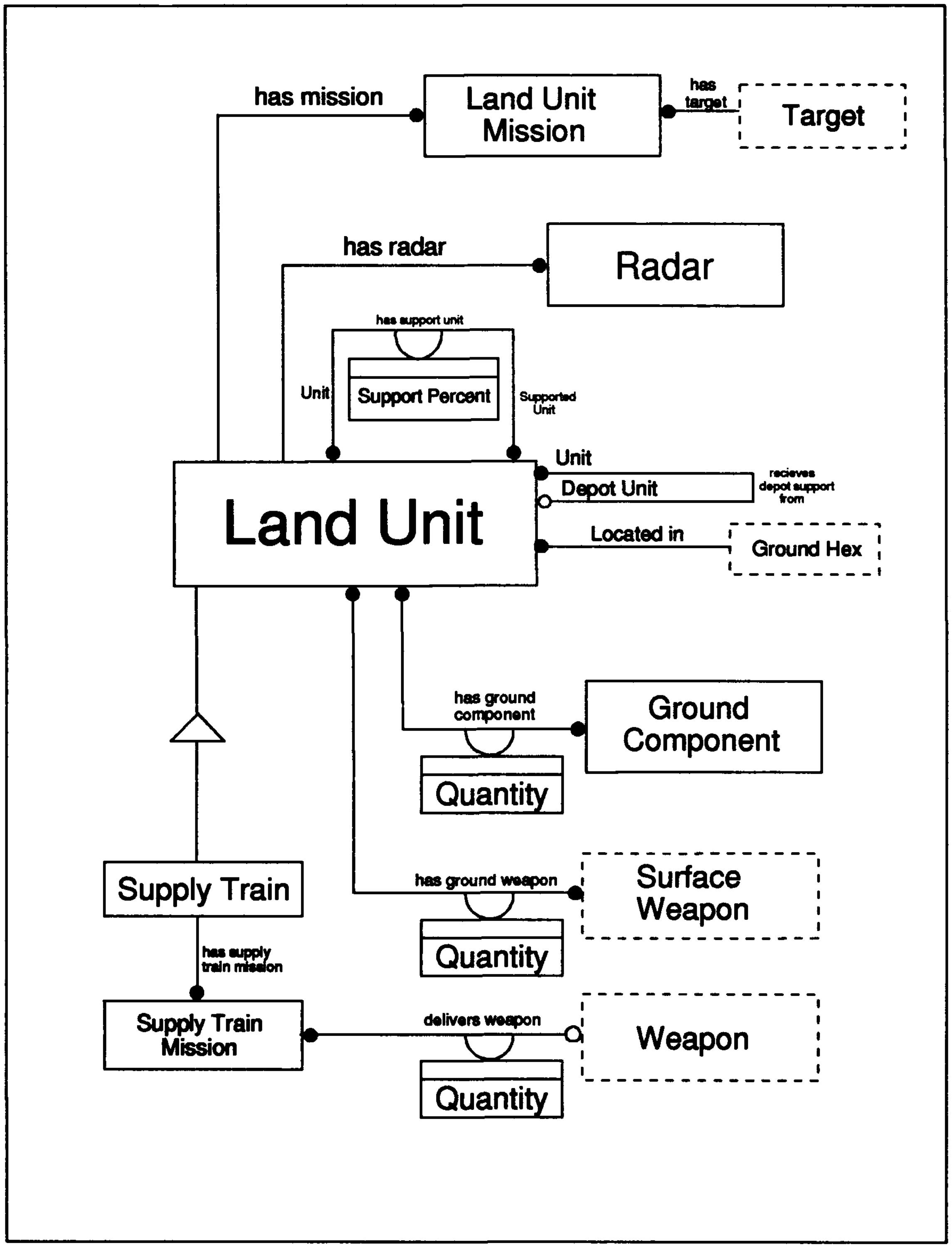

Figure 7. Object Model (cont.)

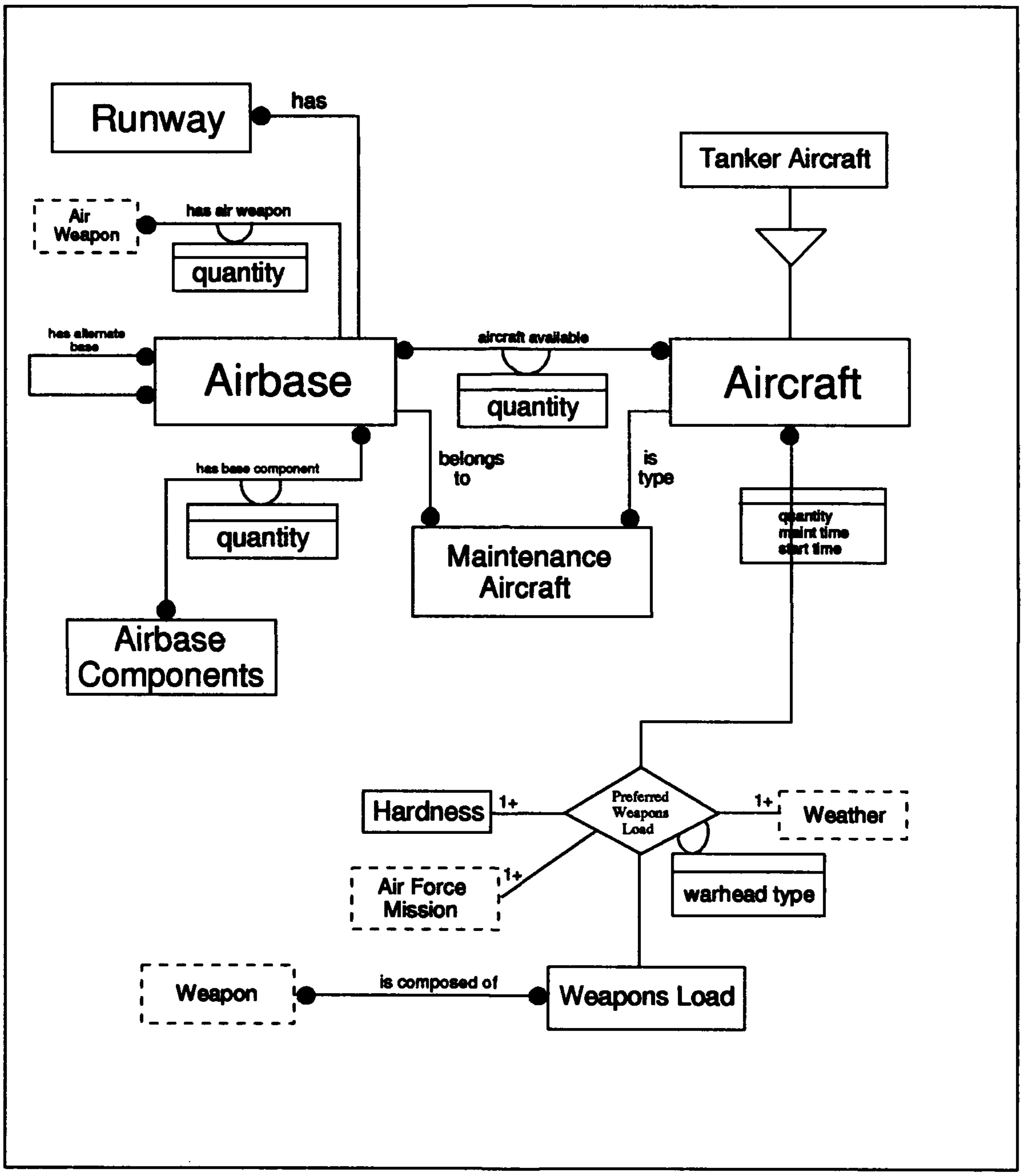

Figure 8. Object Model (cont.)

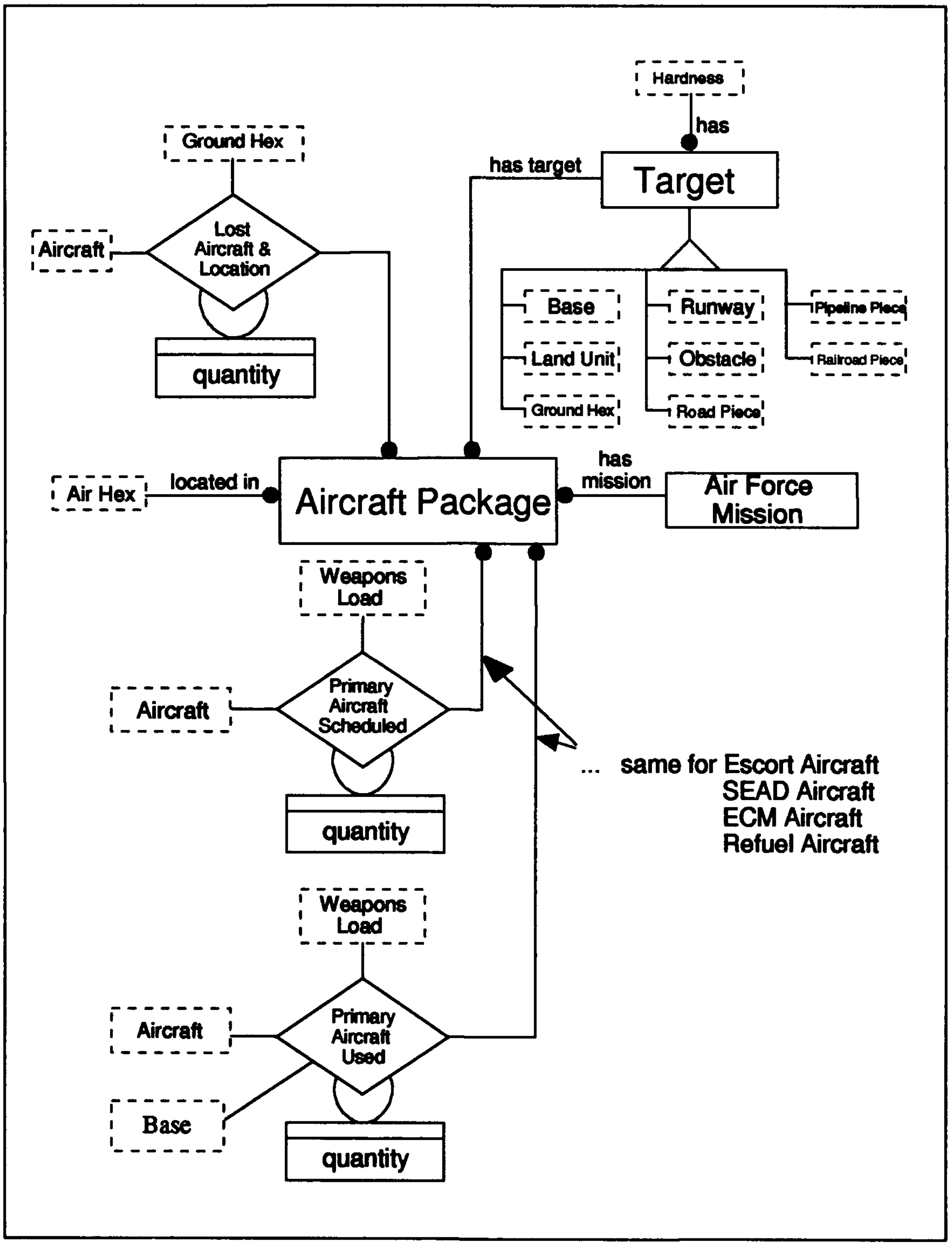

Figure 9. Object Model (cont.)

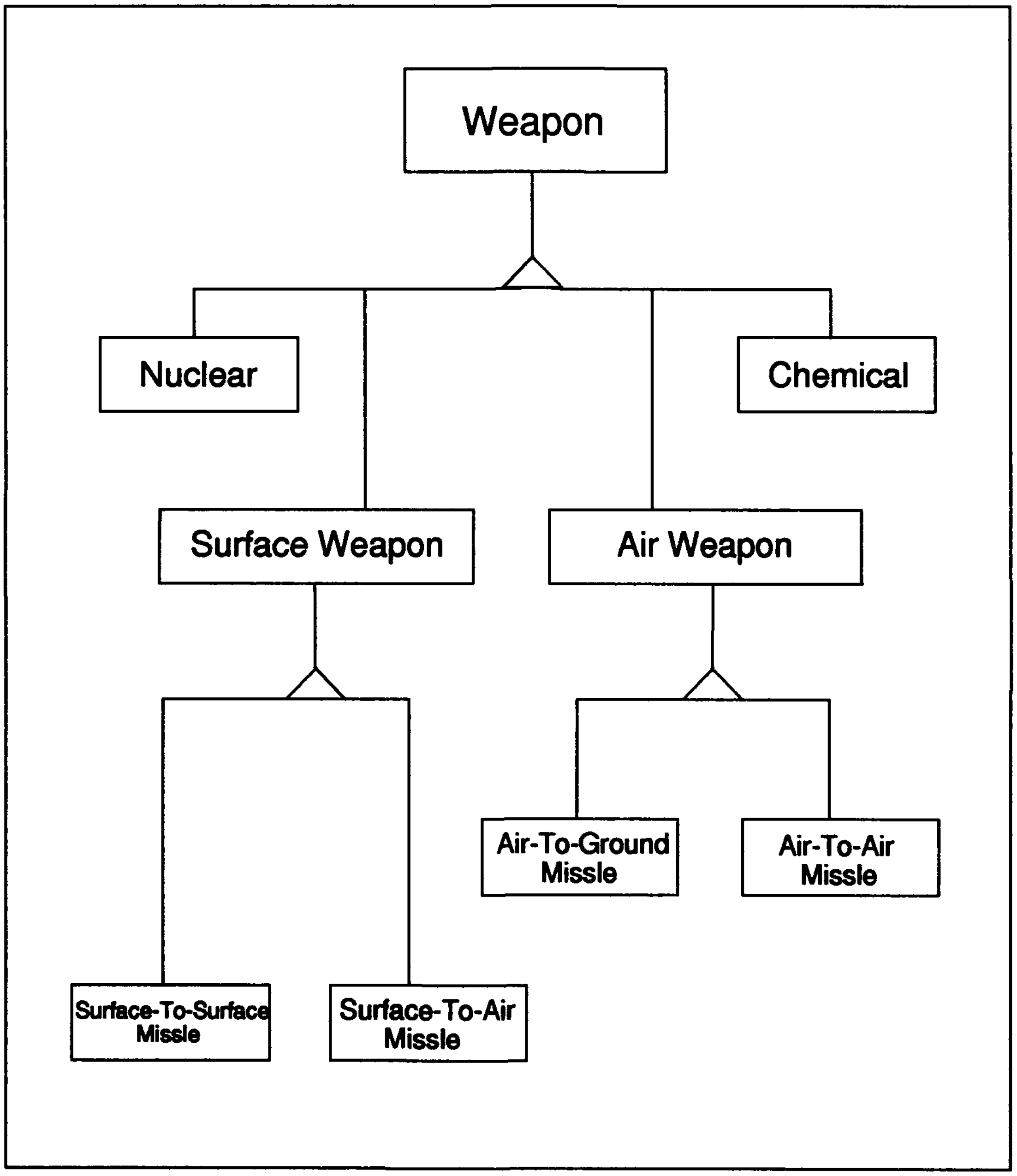

Figure 10. Object Model (cont.)

3.6 Summary

This chapter briefly described the first four steps of Booch's analysis and design process that was conducted on the Saber simulation. During this process, emphasis was placed on what was required of the simulation rather specific details of how things were going to be implemented. At times, this was difficult because several portions of the simulation were already coded, with only portions based on an object-oriented design.

IV. Modifications, Clarifications, and Enhancements

4.1 Introduction

As mentioned in Chapter I, two previous thesis efforts were involved with implementing portions of the simulation. Ness, who implemented the land battle, and Sherry, who layed an object-oriented framework for the air battle and made a few modifications to the land battle. The purpose of this chapter is to discuss details of modifications and enhancements made to existing code that were conducted as part of this thesis. Discussion first covers data structures followed by implementation details of the land and air battles.

4.2 Land Battle Modifications and Enhancements

Since Ness implemented the original land battle of the Saber simulation, there have been several changes. One of the major changes was the conversion of many of the static array information structures to dynamic linked list structures, as discussed in the following chapter. These changes resulted in modifying a majority of the functions and procedures of the land battle, however, the underlying implementation details remained intact. The following sections describe other modifications and enhancements added to the land battle.

4.2.1 Land Unit Movement Algorithm. There were several modifications made to the land unit movement algorithm implemented by Ness. Listed here are the major steps of the modified movement algorithm:

1. Before a land unit begins to move, the possible directions of movement (N, NE, SE, S, SW, and NW) are determined. The directions the land unit are allowed to move include the three directions which will take the land unit in the general direction of its mission location. Figure 11 is an example of a unit moving in a northeast direction. The arrows indicate the three directions the algorithm would allow the unit to proceed if there were no obstructions. Specifically, the solid arrows represent the routes the land unit will ultimately be able to travel. The outlined arrows represent routes the unit could have taken, but is unable to because one of the following reasons:

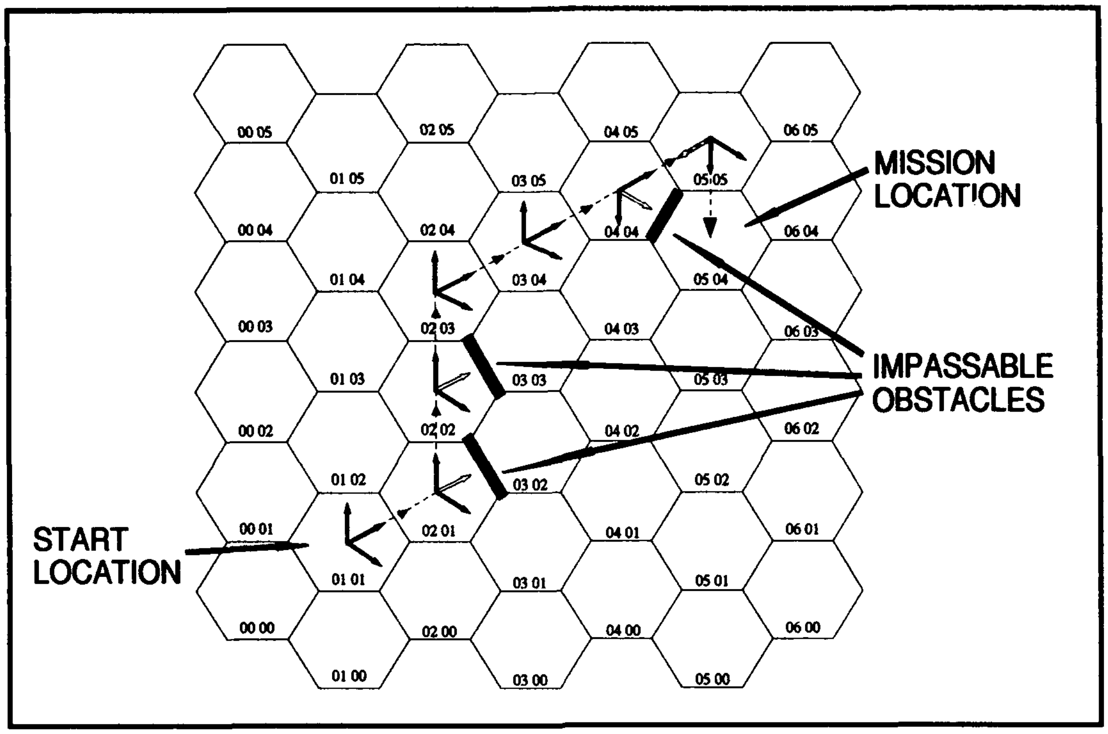

Figure 11. Example 1 : Land Unit Movement

- there is an impassable obstacle at the hexside,
- the hex it would enter would put the unit in the "no man's" zone, or
- the unit would be traveling in the direction it just came from. There is an exception to this last reason and is discussed later.

2. The optimum route based on the possible routes is then chosen. The attributes used to determine the optimum route are:

- the present pie piece trafficability,
- the obstacles that are located at the border being analyzed,
- the pie piece trafficability of the adjacent hex,
- the weather of the adjacent hex,

- and the distance from the adjacent hex to the unit's mission location. To account for the zigzagging effect of the ground hexes, 0.5 is subtracted from the y coordinate of the odd columns when calculating the distances.

3. Once the optimum path is chosen, the time (grid_time) for the land unit to travel to the border of the hex is calculated. The time is based on the calculated average movement rate for all units, the movement rate of the land unit, the traversed pie piece trafficability and the current weather.
4. Simulation of movement is accomplished by decreasing the grid_time of each land unit having a mission. The decrease of the grid_time is performed during each time slice of the simulation. As long as the land unit does not come in contact with an enemy force the unit is allowed to move. If the land unit does come in contact with an enemy force and at least one of the opposing land units has an attack mission, attrition occurs and none of the contacted units can move. The units can only move after one of the two units withdraws, the attacking unit receives an override mission, or a unit is destroyed. If the land unit does come in contact with an enemy force, but neither of the opposing land units have an attack mission, they gain intelligence on each other and can continue to move.
5. After the grid_time of the unit falls below a user specified value, the land unit must transition through the border. A new grid_time is then calculated which represents the amount of time it will take the unit to overcome the border obstacles. If the land unit has any engineering support units, it will overcome the obstacles much faster, otherwise, the reduction of the grid_time will occur at the land unit's own engineering rate.
6. When the land unit has overcome the border obstacles, it is removed from the old hex and added to the new hex. If the new hex is not the final destination of the land unit, the possible routes are determined, the optimum route is selected from the possible routes, and the grid_time for the land unit to move from the border to the new hexes' center and from the center to the border of the chosen route is calculated. This process continues until the land unit has reached its mission location.

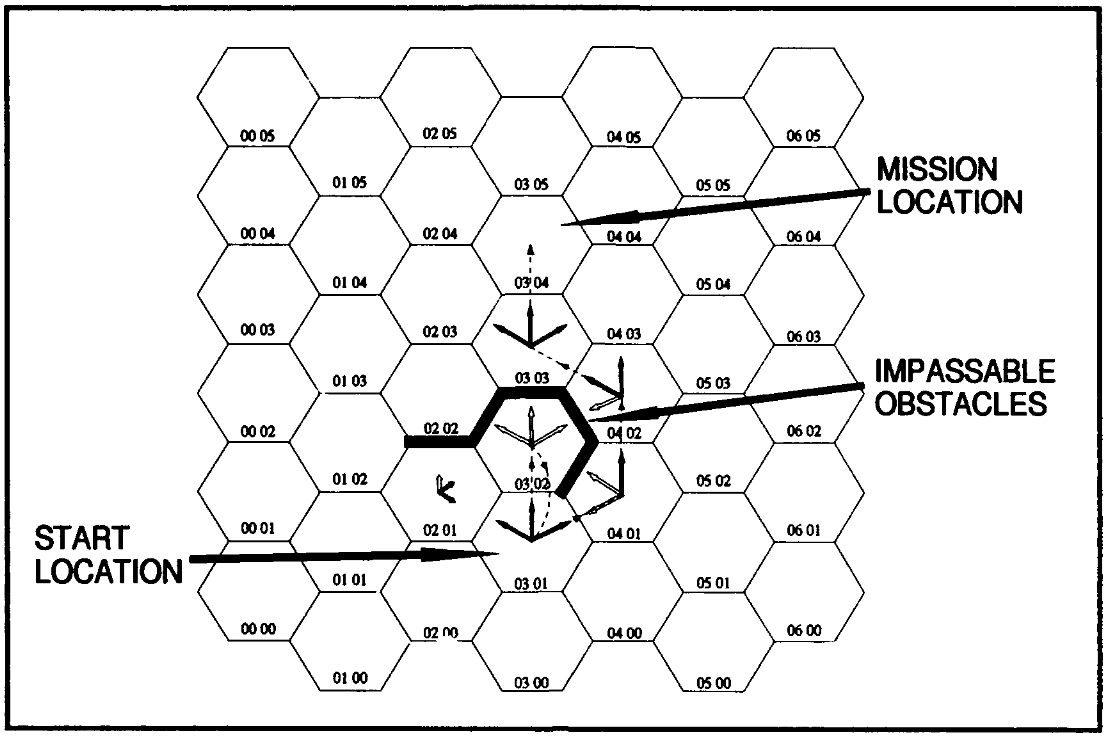

Figure 12. Example 2 : Land Unit Movement

Because this movement algorithm only looks one hex ahead in determining the best possible route, it is possible for the land unit to enter into a trap (all three of its allowed directions are blocked) as shown in Figure 12. The land unit starts out in hex 03 01 and chooses to move into hex 03 02. Once there, all three of its allowed directions are block. If this occurs, the land unit is allowed to traverse back in the direction it just came from (03 01). This is the exception "to not allowing a unit to move away from the general direction of its mission location." For the next move, the land unit would have to travel in either a northeast or a northwest direction and is not allowed to travel back into the trap since it is the direction it just came from. The land unit chooses to move in a northeast direction and eventually arrives at its mission location.

This movement algorithm, however, is not foolproof. From the example of Figure 12, if the land unit would have decided to travel in a north west direction after backing out of the trap, it would move into hex 02 01. From hex 02 01 it would be possible for the unit to move into the same trap (hex 03 02). It would then back out of the trap (hex 02 01) and have no other choice but to move into hex 03 01, once again, and start the whole cycle again never reaching its mission location. We considered developing some type of breadth or depth first search, however, a complete exhaustive search of the optimum route may be computationally intensive.

4.2.2 Land Unit Missions. In Ness' land battle, land units could only have one mission at a time and it was necessary for the land unit to complete that mission before it could start on another one. This meant that in order to move a land unit from one location to another and then to another, it was necessary for the simulation to stop once the land unit reached its first mission location, have the player input the new mission, and then have the simulation run again for the unit to reach its final desired location. This was not practical, plus with the addition of supply trains (discussed in a later section), it was paramount that land units could have several missions without having the simulation start and stop. To give the Saber simulation more flexibility, each land unit can now have a queue of several missions, each with a requested day and period the mission should be performed.

There are two types of land unit missions, regular missions and override missions. The regular missions are performed starting on the requested day and period as long as the land unit is not currently performing another mission. If the land unit is performing another mission, the mission is delayed until the current mission is completed. The override missions have a higher priority than regular missions. If a land unit is performing a regular mission and a override mission is to be performed that day and period, the regular mission will be aborted along with all the back logged regular missions. These aborted missions are deleted and will not be executed after the override mission has been completed. It should also be noted that an override mission does not have priority over another override mission.

The addition of allowing land units to have many missions gives the player the flexibility of entering as many missions as he/she chooses for the next simulation run. The override missions allow the user to specify that a mission is to override the current mission plus all back logged missions as it is possible for a land unit to be heavily delayed.

4.2.3 Combat of Land Units. During each simulation time slice, opposing land units which are in contact (at least one land unit in contact must have an attack mission), will engage in combat. During combat, the combat strength of the land units (their combat power attribute), will determine the outcome of the engagement. The combat powers of land units which are both at their final destination and defending, are calculated based on the following:

1. The trafficability of the hex.
2. The amount of time the land unit has been in a defensive posture in the hex.
3. The firepower of the unit. In Ness' original land battle, the firepower was an aggregated value representing the strength of the land unit. With the enhancements introduced by Mann, each land unit now possesses specific numbers of component types (tanks, helicopters, armored vehicles, etc.), each of which has a firepower score. The land unit's firepower is now calculated by summing the firepower scores of each of the component types multiplied by the quantity of that component type that the land unit possesses. A land units ammo usage rate and fuel usage rates are also determined by the amount and types of components it has.

If the land unit is not at its final location or is not defending, the combat power of the unit is merely equal to the inherent firepower. This accounts for the advantages a defending unit has over an attacking unit.

To simulate combat, the combat powers of all the land units within a hex are totaled to represent the combat power (CP_Out) of the hex they are located in and distributed proportionally against the hexes they are in combat with (CP_In). If a hex has opposing forces on multiple sides, the hexes combat power is divided proportionally into the opposing hexes. The attrition of each hex is then calculated from the hexes combat ratio (CP_In/CP_Out) and a user defined combat ratio adjustment. The hex's attrition rate is then applied to each of the land units located in the hex. Ness applied attrition by subtracting from the land units firepower, however, now attrition occurs by destroying land units components. For example, suppose a land unit possesses the component types and numbers shown in Table 2. If the land unit experiences an attrition rate of 34%, it will take the losses shown in Table 3.

Table 2. Example Land Unit's Components

Component Type	Quantity
M1A1	25
M107	14
TOW	3
FROG3	36

Table 3. Example of a Unit's Losses with 34% Attrition Rate

Component Type	Quantity Lost	Amount Remaining
M1A1	.34 X 25 = 9	16
M107	.34 X 14 = 5	9
TOW	.34 X 3 = 1	2
FROG3	.34 X 36 = 12	24

Because parts of components cannot be destroyed (for example, one half of a tank cannot be destroyed), the numbers of components lost are rounded to the nearest integer. The land units new firepower is now calculated on the components it has remaining.

4.2.4 Logistics. Transportation and distribution of supplies is the primary worry of the logistician.(8) Ness' land battle simulated logistics, however, the manner in which it was accomplished was unrealistic. During each simulation time slice, every land unit automatically received supplies from its depot land unit. Because of this, it was not possible for enemy forces (land units, air interdiction) to target supplies as they were being transported to front line troops.

The addition of supply trains, which provide supply transportation, adds to the realism of the Saber simulation. The supply trains are specialized land units and are able to carry ammunition, petroleum, hardware supplies, and aircraft spare parts from either land unit depots or airbase depots. Because supply trains are specialized land units, they perform all the same operations. This includes movement and the ability to be targeted by enemy forces. If the supply train is targeted and hit, the amount of supplies the supply train carries is reduced by its attrition rate.

There are two types of supply trains. A regular supply train (ST) and a predetermined supply train (PST). The ST is a "one way" supply train that transports logistics supplies from either a land or airbase depot to one or more land units and airbases. Once the ST supply train has finished its supply missions it then remains at its last location until it receives additional orders. The PST is different from the ST in that it continues to perform its missions over and over. After the PST performs its last mission, it returns to its originating depot, gets resupplied, and sets off to perform the same missions. If the supply trains are destroyed before they reach their destinations, it is the players responsibility to regenerate another ST or PST.

As part of the Saber group meetings, it was decided that the best way to model overall logistics movement among land units was to have the supply trains transport supplies from land depots (DEPOT) to front line depot support units (DPSUP). The front line fighting units would then receive supplies automatically from their depot support unit. This is accomplished each time slice by calculating the current capacity of each depot support unit. Each land unit then takes on the capacity of its depot support unit only if its own current capacity is lower. If a land unit does not have a depot support unit, the only way for it to be resupplied is with a ST.

Figure 13 is an example of supply train movement. It depicts the actions of a PST originating from its resupply land depot, providing supplies to depot support units, and returning back where it will resupply to continue the loop. The solid arrows indicate the automatic disbursion of supplies from the the depot supply units to its supporting units. Figure 13 also depicts a ST which moves from the land depot to its mission location and another PST which acquires its supplies from an airbase depot, supplies the airbase and returns. Once the ST reaches its destination, it waits for further orders, while the PST will continue to supply the airbase.

4.3 Air Battle Modifications and Enhancements

Sherry conducted an object-oriented design on the conceptual model developed by Mann. This section describes the specific design details of the implementation of the air battle.

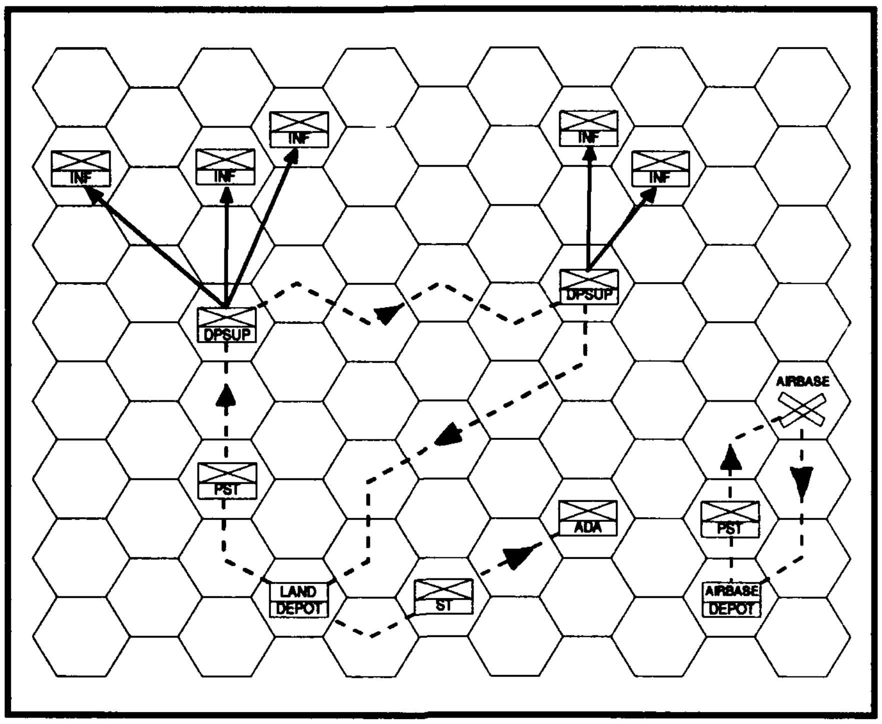

Figure 13. Supply Train Movement

4.3.1 Aircraft Package Movement Algorithm. As part of Sherry's thesis, she developed an initial aircraft package movement algorithm. For sake of consistency, it was decided to change the movement algorithm making it as similar to the land unit movement algorithm as possible. This new movement algorithm is much simpler and a more direct route is taken to the target and back.

The only aspect considered in the movement algorithm is what air hex will take the aircraft package closer to its destination. This is accomplished by calculating the distances from each of the six surrounding air hexes the aircraft package is currently located in and selecting the air hex with the shortest distance to the aircraft package's destination. Figure 14 is an example of an aircraft package moving from its start hex (030604), to its target location (032114), and back.

4.3.2 Formation of Aircraft Packages. Players input aircraft packages they wish to be executed. Each aircraft package has a mission, a region from which the aircraft package

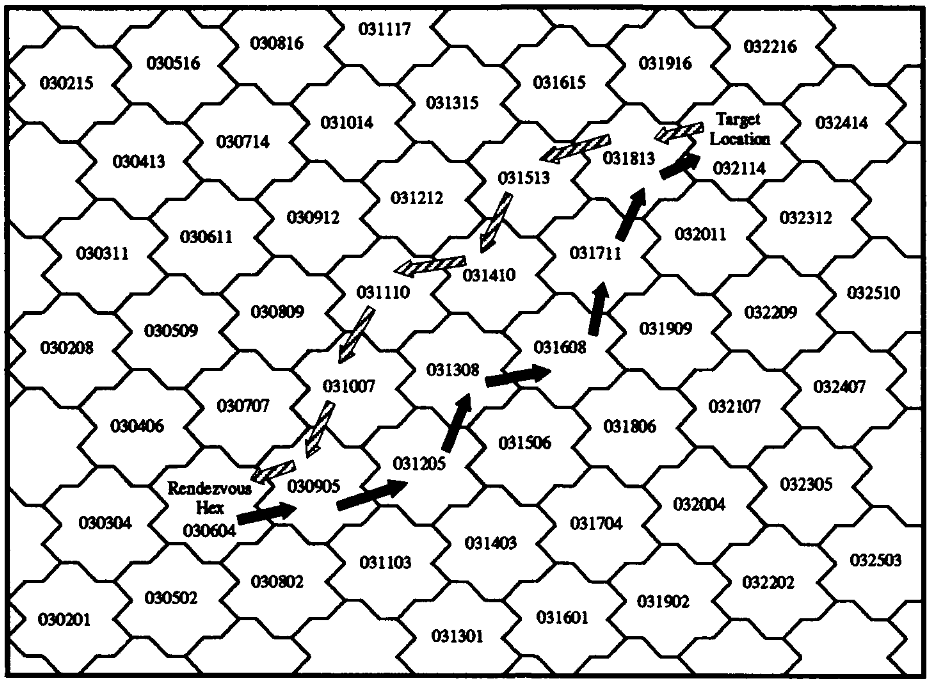

Figure 14. Aircraft Package Movement

will be formed, a rendezvous hex, a target, a priority, a day and period it is to be executed, and a list of primary aircraft. Along with the primary aircraft, aircraft packages can also have support aircraft which include escort, SEAD, ECM (Electronic Counter Measures), and refueling aircraft.

At the beginning of each time slice, requested aircraft packages are attempted to be formed. Aircraft packages that were delayed from the previous time slice are attempted first. The current aircraft packages are then attempted starting with the one that has the highest priority. In order for an aircraft package to be successfully formed, the bases in the region from which the aircraft package will fly are polled for aircraft availability. In order for a base to dedicate an aircraft the following are checked:

- the base's weather is sufficient for the base to operate,
- the base's weather is sufficient for the aircraft to fly,
- the aircraft is not in maintenance,

- the runway length at the base is long enough for the aircraft,
- the base has enough fuel for the aircraft,
- the base has the weapons available for the aircraft,
- and the target is within the range of the aircraft or refueling aircraft are available.

In forming an aircraft package, a check is first made to determine whether the minimum amount of aircraft are available. There are minimum percentages of primary, escort, SEAD, ECM and refueling aircraft that are needed before the aircraft package will be allowed to perform its mission, each of which are user specified. If the minimum aircraft are available, the aircraft package is formed. If aircraft are still available, the aircraft package is filled to its requested aircraft numbers. If the minimum aircraft are not available, the aircraft package is delayed until the next time slice where it will once again attempt to be formed. **If it fails a second time, the aircraft package is cancelled.**

4.3.3 Aircraft Package Missions. Once the aircraft packages are formed, they are ready to be executed. Each aircraft package starts at its designated rendezvous hex. The level at which the aircraft packages will fly is the highest allowed by the aircraft which comprise the aircraft package.

The simulation begins by performing all the area missions first. To simulate each of the area aircraft packages happening simultaneously, each package is flown in sequence one air hex at a time. This is then continued until all the area packages have reached their mission locations. Once all the area aircraft packages have reached their mission locations, the strike missions are performed. Again, each strike aircraft package moves one air hex at a time.

4.4 Summary

This chapter described the changes and enhancements to Saber that were conducted as part of this thesis. Some of the changes included modifications to the movement algorithms for both the land units and aircraft packages. Each of these were described in detail. Enhancements included the ability for land units to have more than one mission, the addition of logistics movement through the use of supply trains, and the formation of aircraft packageses. Also included was a description of how the results of combat between land units is resolved.

The following chapter will describe the implementation details of the simulation.

V. Implementation

5.1 Introduction

The last of Booch's object-oriented design steps is to actually implement each of the objects. This chapter includes a discussion of the data structures used to implement each object. It also describes the extensive testing that was accomplished during implementation phase.

5.2 Information Structures

The type of data structures, and the manner in which they are used in large programs, can play a very important role in determining the execution time and memory requirements necessary to run the program. Often, it is just these two aspects, speed versus memory, that need to be considered. The following sections describe the data structures used to implement each of the primary information structures.

5.2.1 Land Unit Structure. In the initial land battle implemented by Ness, the data structure used to represent land units was an array of records. Each record contained the general characteristics of the land unit plus an access type which "pointed" to another land unit for the purpose of stringing units together that were located in the same ground hex. In Ada, it is necessary to declare the size of an array prior to compilation time, therefore array sizes were "hard coded" into the simulation. This detracted from the desired flexibility of Saber in which the number of land units involved in the simulation could range from just a few to several hundred.

For purposes of using memory space efficiently, using a linked list of nodes is appropriate. In Ness' thesis, he discusses the idea of using a linked list to represent the land units, however, he reverted to the array structure "because of the inability of MS-DOS and the Janus Ada complier to handle access types that used over 64K of RAM"(17). The Saber simulation is now being compiled using VERDIX Ada and run on a Sun Sparc II workstation alleviating this problem.

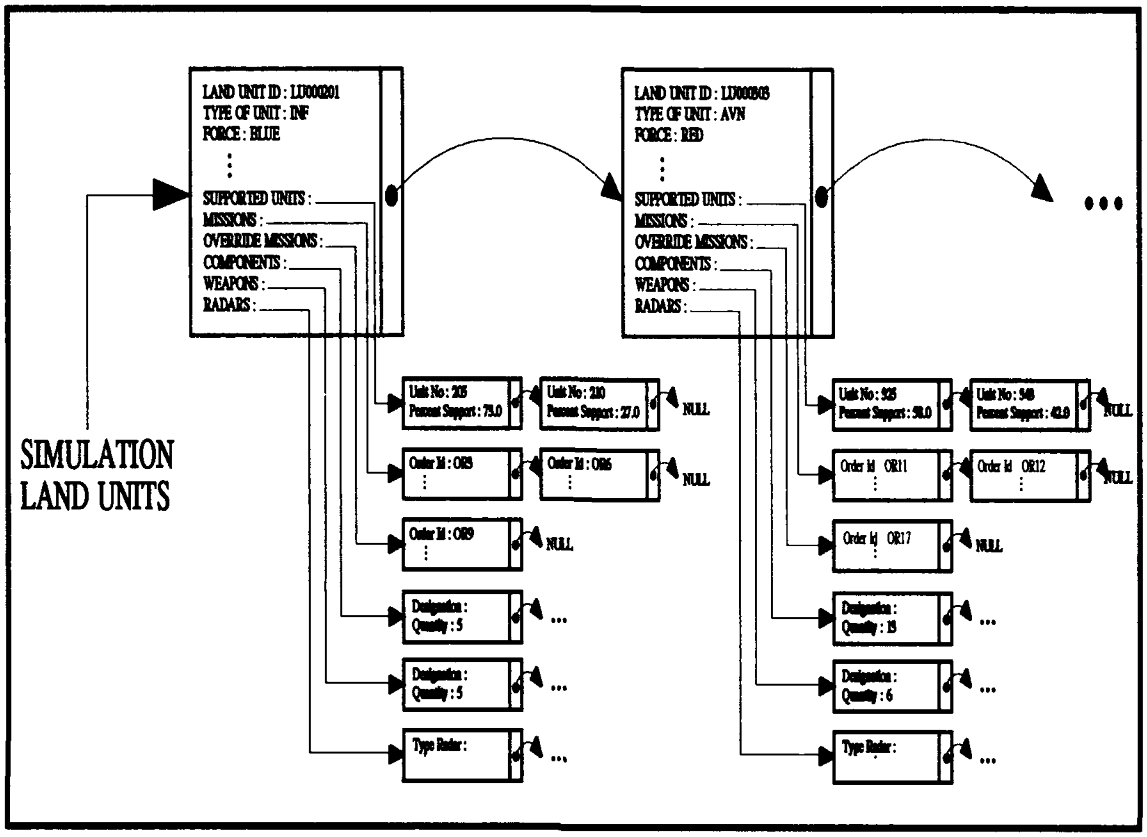

Figure 15. Example Structure of Simulation Land Units

Figure 15 shows the implemented information structure of the simulation land units. Access to the linked list always takes place through the head of the list, where each node is traversed until the desired land unit is found. This traversal of the linked list has an execution time slower than the direct access of an array structure, however, flexibility and memory requirements were the factors for using linked lists. The execution time requirement (four hours between input and analysis) does not seem to be a problem, however, at the current time, there have been no execution times run against different (large scale) scenario sizes.

Looking at Figure 15, each node of the land units list stores the characteristics of a land unit in a record type. Within the record type there are additional data structures. This includes additional access types which are used to hold the following land unit information:

1. which units it supports,
2. the missions it must perform,
3. the override missions it must perform,
4. the types and quantities of components (tanks, armored vehicles, etc.) it possesses,
5. the types and quantities of weapons it possesses, and
6. the types and quantities of radars it possesses.

Again, by using linked lists, the simulation will dynamically allocate memory only as needed.

Several other simulation objects were implemented in the same fashion as the land units. These include the :

1. supply train object,
2. base object.
3. depot object,
4. aircraft object, and
5. the aircraft package object.

5.2.2 Ground Grid Structure. The original ground grid system implemented by Ness was a two dimensional array in which the vertices pointed in a north and south direction. During Sherry's thesis effort, it was decided to modify the ground grid structure so that the vertices would point in a east and west direction. This change was made in order to stay consistent with the current grid structures used by other war games at the Wargaming Center, Maxwell AFB and allowed existing code provided by the Wargaming Center to be used in the Saber user interface(12). Figure 16 shows the old and current ground grid structures and their numbering schemes. As a result of this modification, several changes to the land battle were required. In the old hex system, the x and y coordinates followed straight lines. The x coordinate running in a north-east and south-west direction, and the y coordinate running directly east and west. As a result, the x

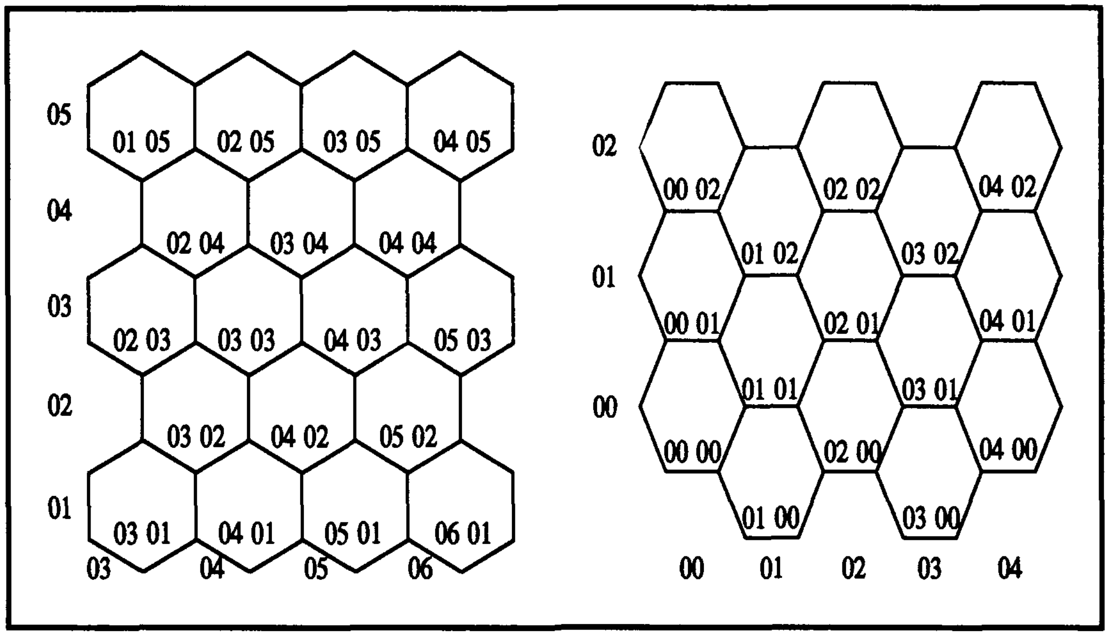

Figure 16. Old and Current Ground Hex Systems

and y differences moving from hex to hex in a particular direction were always the same. For example, moving from any hex in a northeasterly direction, the x coordinate did not change and the y coordinate increased by one. In the current hex system the x coordinate runs in a north and south direction, however, the y coordinate does not run in a straight line, but zigzags as it runs from east to west. As a result, the x and y differences moving from hex to hex in a particular direction are not always the same. For example, moving in a northeasterly direction from hex 01 01, the x coordinate increases by one and the y coordinate stays the same. Now, if you move in a northeasterly direction from hex 02 01, the x coordinate still increases by one, but the y coordinate also increases by one. To accommodate this zigzaging of the y coordinate, simulation code had to be duplicated. Once if the originating hex belonged to an even column, and the other if the hex belonged to an odd column.

Each ground hex (array element) is composed of a complex information structure. An example ground hex, 04 09, is shown in Figure 17. The top level data structure is a composite record type, and contains the hexes characteristics (weather, weather zone, force, mission, etc.). It also contains an array of the hexes six pie pieces (N, NE, SE, S, SW, NW). The example shows the structure of the north east side. Each hex pie piece is a record type which describes the characteristics of the of the pie piece as well as the side of the pie piece. For example, *Hexside_Id* is the unique identifier of the side, *Feba* indicates whether the side is part of the forward edge of the battle area, *River* indicates if there is a river or stream at the side, and *Traffic* indicates the trafficability of the entire pie piece. The top level record type also contains access types to lists of land units, bases, and depots that are located in the hex. Due to the tremendous amount of structural sharing taking place, these lists contain only the land units, bases, and depots identification numbers, and not pointers to the actual information structures. This increases the amount of overhead required, however, the amount of structural sharing and complexity are greatly reduced. The added overhead comes in the form of searching the linked lists to obtain the desired object information rather than having direct access. For example, if the total combat power of a hex needs to be determined, the list of land units in the hex must be traversed one by one, stopping to traverse the linked list of land units to obtain the combat power of each land unit.

5.2.3 Road, Railroad, and Pipeline Structure. Road, railroad and pipeline segments are located in a ground hex pie piece and run radially outward from the hex center to a hex side. During implementation, it was decided to place these structures individually within their own Ada packages rather physically adding them to the structure of the ground hex pie piece. Road, railroad and pipeline segments are targetable entities in the simulation. If the segments were physically placed within a ground hex, all the ground hexes and their pie pieces would need to be searched in order to find a particular road, railroad, or pipeline segment. Placing each of these segment types within their own package, allows for quick access to a particular segment and also allows for the ability to traverse the road, railroad, or pipeline segments that make a larger road, railroad, or pipeline.

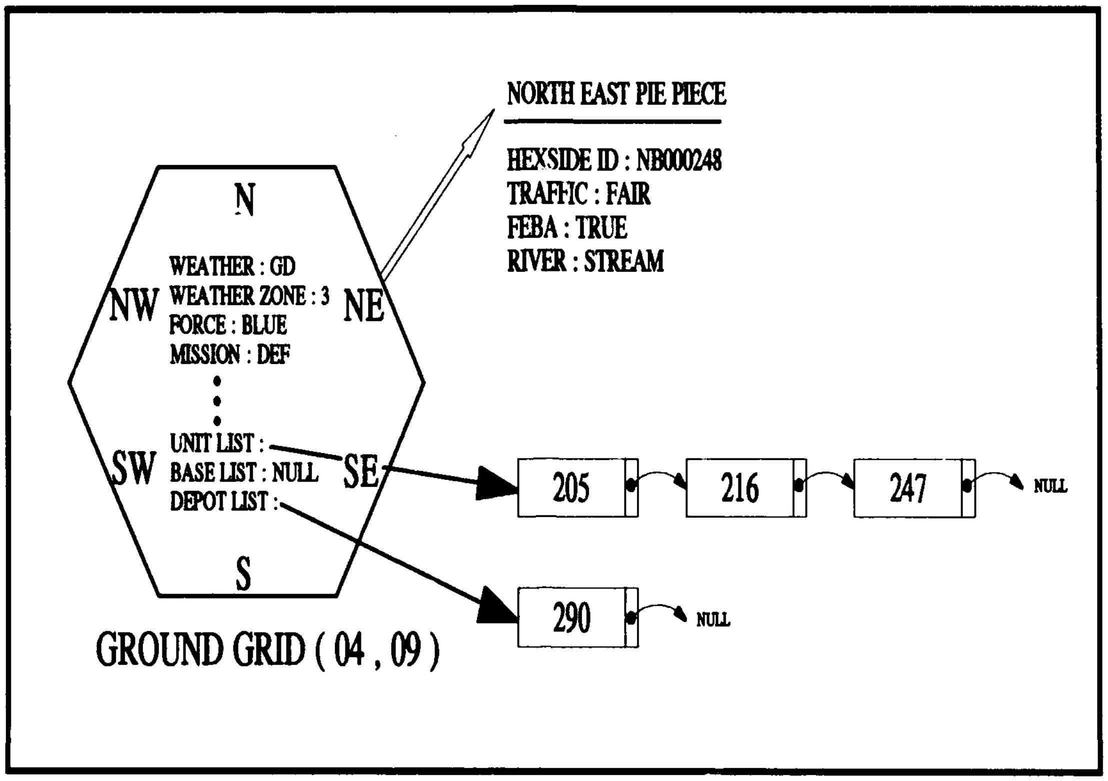

Figure 17. Structure of a Ground Hex

5.2.4 Air Grid Structure. The basic air grid structure was not changed from the sparse matrix implemented by Sherry. The sparse matrix provides an array of air hexes. The size of the array is hard coded in the simulation and should be close to the actual number of air hexes. This value can be easily calculated by taking the number of ground hexes, dividing by seven (each air hex encloses seven ground hexes), and multiplying by six (the number of air hex levels). Each array element is then composed of the air hex number, and a composite record type, which contains the air hex characteristics (weather, trafficability, etc.) and access types pointing to lists of aircraft packages and satellites located within the air hex. Just like the units list in a ground hex, the aircraft package lists do not contain all information on the aircraft packages nor does it give you direct access to the information. The lists contain only the aircraft packages identification. In order to obtain the information on a aircraft package the simulations linked list of aircraft packages must be searched.

5.3 Information Hiding of Saber's Objects.

In the last section, some of the object data structures were described. These data structures are important, however, they should be hidden from higher level program units through information hiding. "Information hiding is the practice of concealing implementation details from higher level program units, so that, at any given level of program development, the programmer has access only to the degree of detail that he needs (14)." Information hiding in the program promotes modifiability, reusability, and reliability. As an example, the following paragraphs describe how information hiding was accomplished on the land unit object.

As discussed earlier, the land unit structure is complex, having multiple linked lists within another linked list. This complexity is hidden from higher level units by placing the declarations of both the land units data type and data structure in the body of the land units package were it can only be "seen" and used by the body itself. The only way for a higher level program unit to access the information of a land unit is through the package specification. If a higher level program unit wanted to find the location of each land unit in the simulation, the code in Figure 18 would be used.

The procedure **Set_First_Land_Unit** sets an internal pointer to the first land unit in the linked list of land units. When the **Next_Land_Unit** procedure is invoked, if the internal pointer is not pointing to a null value, the land unit number (**Land_Unit_No**) is returned and the status value (**Another_Land_Unit**) is set to true indicating that there was another land unit. The internal pointer is then set to the next land unit in the linked list. If the internal pointer is pointing to a null value, the status value is set to false indicating that there are no more land units.

The land unit number is the key for gaining information of a particular land unit. As shown in the above code, the location of each land unit is obtained by passing the

```
with Land_Units;

procedure Traverse_Units_Locations;

  Land_Unit_No : Integer;
  Another_Land_Unit : Boolean;
  Location_Of_Land_Unit : Integer;

  begin
    Land_Units.Set_First_Land_Unit;
    Land_Units.Next_Land_Unit(Land_Unit_No, Another_Land_Unit);
    while Another_Land_Unit loop
      Location_Of_Land_Unit := Land_Units.Location_Of(Land_Unit_No);
      Land_Units.Next_Land_Unit(Land_Unit_No, Another_Land_Unit);
    end loop;
  end Traverse_Units_Locations;
```

Figure 18. Example Code - Traversing Simulation Land Units

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

with Land_Units;
with Ground_Components;

procedure Firepower_Of_Land_Unit (Land_Unit_No) is

  Ground_Component_Designation : String(1..5);
  Quantity : Integer;
  Another_Ground_Component : Boolean;
  Firepower : Integer := 0;

  begin
    Land_Units.Set_First_Ground_Component(Land_Unit_No);
    Land_Units.Next_Ground_Component(Ground_Component_Designation, Quantity,
                                     Another_Ground_Component);
    while Another_Ground_Component loop
      Land_Units.Next_Ground_Component(Ground_Component_Designation, Quantity,
                                       Another_Ground_Component);
      Firepower := Firepower +
         (Ground_Components.Firepower_Weight_Of(Ground_Component_Designation) *
                                                                      Quantity;
    end loop;
  end Firepower_Of_Land_Unit;

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

Figure 19. Example Code - Firepower of a Land Unit

land unit's number to the function **Location_Of**. All other attributes of a land unit are acquired through similar functions.

To obtain the ground components of a particular land unit, a similar technique is used as described above. Figure 19 is an example of code that would be used to calculate the firepower of a land unit.

The **Set_First_Ground_Component** procedure sets an internal pointer to the first ground component of the desired land unit. As long as the internal land units ground components pointer is not equal to null, the **Next_Ground_Component** function returns the ground component designation and quantity and updates the internal pointer to the next ground component. The status value is also set to true indicating that there is another ground component for land unit 101. For each ground component designation, the firepower weight is obtained by calling the **Firepower_Weight_Of** function in the ground components package. This value is then multiplied by the quantity and then added to the overall firepower of the unit. The loop continues until the there are no more ground components, indicated by a false value for the status value **Another_Ground_Component**.

With the procedures and functions described above, the users of the land units specification have no idea of the underlying data structures used to implement the land units object. This enhances maintainability of the code by reducing the coupling between objects.

5.4 Enhanced Performance of Implemented Objects

Let's consider a typical scenario where a supply train reaches a land unit it is to resupply with ammunitions and fuel. The resupply of the land unit is accomplished by envoking the procedures **Increase_Ammo_Of_Unit** and **Increase_Pol_Of_Unit** passing the land unit's unique identification and quantities to be resupplied. When **Increase_Ammo_Of_Unit** is executed, the linked list of all the simulation land units is traversed until the particular land unit's identification is found. The ammunitions of the land unit is then increased by the quantity to be resupplied. When **Increase_Pol_Of_Unit** is executed, the search for the land unit starts again at the beginning of the linked list.

To prevent this type of unnecessary searching, several options were considered. The first option was to implement the appropriate objects using hash tables, however, due to time constraints this was not pursued. Although this was not pursued, it should be the focus of further research. The second option was to create a small cache within each of the appropriate objects. The cache would contain pointers to each of the object instances that were most recently accessed, however, in order to accomplish this, pointers into the middle of the instantiated linked list would necessary. This would violate the linked list abstract data type. The third option, and the one used to speed up the simulation, involved moving object instances within the linked list. As an object instance is accessed, it is removed from its current position in the linked list and placed at the front. Now, if several attributes of the same object instance are accessed close together, the access time will be reduced since the object class will be located towards the front of the linked list. This is very similar to the second option in that the most recently accessed object instances will be located towards the front of the list. On the average, this will provide much faster access times to object instances and the larger the simulation grows, the greater this enhancement will be noticed. During simulation execution, it is sometimes necessary to traverse all the object instances. In these cases, the rearranging of the object instances is not completed, as it would invert the linked list and would not result in simulation speedup.

5.5 Testing

As part of Horton's thesis effort, he constructed the flat files necessary for a Korean scenario. For testing purposes, especially during the development phase of the simulation, this large Korean scenario proved to be cumbersome for practical, rapid testing.

A mini scenario was created to provide a smaller workable environment on which testing was conducted. The mini scenario has a 10×10 ground grid structure and did not conform to any real geographical location, but was concocted so that most possible battle scenarios could be represented. The mini scenario had a few instances of each object (about ten land units, four airbases, two depots, etc.), just enough to provide all possible test cases. As each of the objects were developed, test procedures were written and run against the mini scenario in order to verify the desired functionality.

5.6 Summary

This chapter described the significant issues of the Saber simulation implementation. A description was given of the data structures that are used by each of the major objects. Although some of these data structures are complex, once the interface of each object is developed, the users of the object are unaware of the underlying structures. This allows the data structures of an object to be modified without causing a ripple effect through other code. The end of the chapter briefly described the mini scenario that was created for testing purposes.

The next chapter summarizes this entire thesis effort and gives recommendations for future expansion and enhancements to the Saber simulation.

VI. Recommendations and Conclusion

6.1 Summary

This thesis effort involved conducting an object-oriented analysis, design and implementation of the air and land battle for the Saber theater-level war game. Several thesis efforts prior to this thesis were devoted to the development of the Saber simulation. Ness was responsible for the initial version of the stand alone land battle and Sherry took Mann's conceptual model and conducted an object-oriented analysis and completed a limited amount of implementation. This thesis integrated both the air and land portions into an air land battle simulation.

The methodology used to conduct the object-oriented design was Booch's five step process. These steps included:

1. Identifying the objects and their attributes. The objects and attributes were extracted by considering the nouns used in previous theses (mainly Ness and Mann's).
2. Identifying the operations needed for each object. The operations were determined by examining the verbs that were performed on each object.
3. Establishing the visibility required between objects. This was accomplished by determining what objects needed to interact with other objects.
4. Establishing the interface for each object. This consisted of creating the Ada package specifications and successfully compiling them.
5. Implementing each object. This was accomplished by writing an Ada package body for each of the identified object classes. The implementation of the simulation is not complete due to time constraints, however, a reduced functional model is up and running. The translation of existing code into the object-oriented design proved to be cumbersome at times. Much of the previous code did not conform to object-oriented methods. Many of the air and land battle algorithms were also modified to accommodate enhancements. The code that was generated with this thesis effort provides a solid object-oriented foundation on which the simulation can continue to mature, be modified and enhanced with relative ease.

As part of the analysis and design phase, an object model using Rumbaugh's notation was constructed. The object model captured the static structure of the system by showing the objects in the system and the relationships between the object classes.

During implementation of the object classes, object-oriented techniques were followed. This entailed developing the object class Ada specifications such that a user would be unaware of the underlying implementation details. For example, the user would be unaware whether an object class was implemented using an array or linked list structure. Also, during the implementation phase, some of the object classes were packaged individually while others were combined within a single Ada package. For example, the land unit missions object was packaged within the land units object because a particular land unit mission belongs to only one instance of a land unit. Although a road segment belongs to a particular ground hex pie piece, the road class was individually packaged because it is targetable by enemy forces. By individually packaging the road object, it is much easier and quicker to identify the attributes (for example, its location) of a road segment or to traverse the road segments that are associated with a road name or number.

6.2 Recommendations for Follow-On Efforts

This thesis provided a strong object-oriented design of the Saber simulation. As mentioned earlier, there are still portions of the simulation that must be completed before it can be considered fully functional. The following are specific areas that require further study. Some of these items must be accomplished to complete the simulation, while others indicate recommendations of further research.

- There are still portions of the air battle that must be completed. The formation of aircraft packages and their movement to their target location and back are completed, however, the portions of the air battle that must be completed are:

    1. The detection of aircraft packages by enemy SAM sites and patrolling aircraft as they traverse their mission route.

    2. The resolution of conflict between enemy engagements.

3. The amount of damage the aircraft package inflicts on its target once it has released it weapons load.

- System integration must be conducted. This includes integration of the simulation, user interface, and history file which is produced by the simulation to provide the graphical postprocessor an account of events and status that occurred during simulation runs. Verification should also be conducted on the database in order to verify that the flat files it generates match with requirements of the simulation and user interface/postprocessor.

- As mentioned earlier, the current movement algorithm of land units is sufficient but not foolproof. With the current movement algorithm, it is possible for a land unit to become trapped and be unable to complete its mission. This trapping occurs due to the fact that the movement algorithm only looks one hex ahead in its selection for the land unit's route. Further research should be accomplished to enhance the land unit movement algorithm. Suggestions include some sort of depth or breadth first search to determine the best possible route, however, it would not be practical to complete an exhaustive search as this would become to computationally intensive.

- The structures of roads, railroads and pipelines have been designed and implemented, however, utilization of these assets are not fully exploited. Improvements should be added to the simulation so that roads play a larger role in the movement of units and railroads and pipelines play a larger role in the movement of logistic supplies.

- Much like the land unit movement algorithm, the aircraft package movement algorithm is sufficient for a first iteration, however, it could also use some improvements. Currently, the aircraft package movement algorithm will select the highest level at which all the aircraft that compose the aircraft package can fly. The aircraft package will then fly in a straight line to its target, the primary aircraft will release their weapons at a selected optimal level, and then the aircraft package will return in a straight line at the highest level possible. Further research should be conducted to determine the validity of the current movement algorithm. Perhaps the aircraft package movement algorithm should be modified so that it will select the path of least resistance, changing altitudes as its traverses to its target and home.

- Additional work must still be accomplished on the use of satellites. Procedures and algorithms need to be designed and implemented for the movement of satellites and the intelligence they will provide to the battle.
- The testing up to this point has been conducted on a mini scenario containing only a handful of land units, bases, depots, aircraft packages, obstacles, etc. Further verification and validation should be accomplished on larger sized scenarios for which the simulation is intended. Although it is expected that speed will not be a problem, further research could be conducted in the use of the Ada task construct or the use of a parallel processor machine.
- Further work needs to be accomplished is the area of nuclear and chemical warfare. Many of the basic building blocks are already available to conduct non-conventional warfare, however, specific procedures and algorithms need to be developed and implemented.
- Currently an airbase can have a mission of DEPLOY, however, the mechanism for a base to deploy do not currently exist. Further research needs to be conducted to determine whether entire bases or parts of bases deploy. For example, certain USAF Red Horse units deploy in time of war as well as aircraft and supporting personnel.
- Currently there are only two forces, RED and BLUE, that are involved in the simulation. Further research could be conducted in the area of multiple-sided warfare, which would allow additional forces, such as YELLOW or GREEN, to participate.
- Further research and development could be conducted to speedup the execution of the simulation. For example, current links between ground hexes and the land units located in a ground hex are maintained as a linked list of the land units unique identifications (natural number). A tremendous amount of speedup could be obtained by maintaining a linked list of access types rather than the unique identifier. This would allow direct access to each of the land units located in the hex rather than

traversing the entire linked list of land units to find each of the land units. However, careful consideration of the impact on the object-oriented design will be required.

6.3 Conclusion

This thesis documented the object-oriented analysis, design, and implementation of the Saber simulation. It showed how the Ada programming language was used to successfully develop the Saber simulation. The object-oriented design provides a platform on which the simulation is flexible, easy to understand, and easy to maintain. The simulation is only one component of the Saber war game, and once completed and integrated will provide a tool that will help teach air and ground employment doctrine to the future leaders of the United States Air Force, all other services and its allies.

Appendix A. *Object Classes and Their Attributes*

This appendix lists all of the object classes (alphabetic order) and the attributes that define the object classes. There are several object attributes that are not used by the simulation, however, must be read in order to regenerate the flat files. Brief definitions are given for the attributes used by the simulation. Attributes read, but not used are by the simulation, are not defined and marked as such. For complete definitions of all attributes and example values, see the database data dictionary.

+ indicates attribute is read by simulation but not used except for regeneration of flat files.

* indicates attribute not read in but needed by simulation.

# indicates attribute not currently used by simulation.

- AIR HEX

  1. Air_Hex_Weather - (actual_wx) The actual weather that is occuring in the air hex.
  2. Weather_Zone - (wz) The weather zone of the air hex
  3. Attrition - Attrition of the air hex.
  4. EC - Electronic countermeasure of the air hex.
  5. Trafficability - Trafficability of the air hex.
  6. Persistence - (persistence_time) The time remaining the air hex will feel the effects of a nuclear or chemical weapon.
  7. (*) Blue_Acpkg_List - Linked list of blue aircraft package identification numbers located in the air hex.
  8. (*) Red_Acpkg_List - Linked list of red aircraft package identification numbers located in the air hex.
  9. (*) Satellite_List - Linked list of satellite identification numbers located in the air hex.

- AIR-TO-AIR MISSILE

  1. Designation - The alphanumeric designation of the air-to-air missle.
  2. Force - The color designation of the weapon.
  3. Miss_Range - (range) The range of the air-to-air missle.
  4. SSPK - Single shot probability of a kill.

- AIR-TO-GROUND MISSILE

  1. Designation - The alphanumeric designation of the air-to-ground missle.

2. Leathality_Radius - (lethal_radius) The lethal blast radius of the surface-to-surface missle.
3. CEP - Circular error of probability.
4. PK_Hard_Point_Type - (pk_hard) The probability of destroying a hardened target with a direct hit.
5. PK_Med_Point_Type - (pk_med) The probability of destroying a medium-hardened target with a direct hit.
6. PK_Soft_Point_Type - (pk_soft) The probability of destroying an unhardened target with a direct hit.

- AIRBASE (DEPOT)

1. Base_Id - (airbase_id) The unique identifier of the base.
2. (+) Full_Designator - The full alphanumeric designation of the airbase.
3. (+) Abbrev_Designator - The abbreviation of the full designator field.
4. (+) Command - The air force command that operates the base.
5. Country - The country the base is located in.
6. Force - The color designation of the base.
7. (+) HQ - The command headquarters of the base.
8. (*#) Move_Allowed - Boolean value indicating whether the base can deploy of not.
9. Mission - (base_mission) The mission of the base.
10. Pres_Loc - (location) Present location of the base.
11. Fut_Loc - (future_location) Future location of the base.
12. Width - Width of the base.
13. Length - Length of the base.
14. Region - The region the base is located. (Note : Region also defines the area of a hex a land unit is located (center, border)).
15. Weather_Min - (weather_minimum) The minimum amount of weather necessary at the airbase in order to fly any missions.
16. (*) Is_Base_Overrun - Boolean value indicating whether the base is overrun.
17. (*) Is_Base_Within_Enemy_Art - Boolean value indicating whether the base is within the range of enemy artillery.
18. (*) Is_Base_Under_Nuc_Chem_Atk - Boolean value indicating whether the base is under chemical or nuclear attack.
19. Active_Enemy_Mines - (enemy_mines) The number of enemy mines that have been dropped on the base.
20. Mopp_Posture - Mopp posture of the base.

21. Is_Base_Under_Air_Attack - Boolean value indicating whether the base is under air attack.
22. Pol_Soft_Store - The amount of petroleum, oil, and lubricants the base has in soft storage.
23. Pol_Hard_Store - The amount of petroleum, oil, and lubricants the base has in hard stroage.
24. Max_Pol_Soft - The maximum amount of petroleum, oil, and lubricants the base can have in soft stroage.
25. Max_Pol_Hard - The maximum amount of petroleum, oil, and lubricants the base can have in hard stroage.
26. Maint_Pers_On_Hand - (maint_personnel) The number of maintenence personnel at the base to repair aircraft.
27. Maint_Hrs_Accum - The number of maintenance hours accummlated at the base.
28. Maint_Equip_On_Hand - (maint_equip) Maintenance equipment located on the base.
29. Spare_Parts - The amount of spare airplane parts at the base.
30. Max_Ramp_Space - The maximum ramp space at the base for parking aircraft.
31. Ramp_Avail - The amount of ramp space currently available at the base.
32. Shelters - The number of shelters at the base.
33. EOD_Crews - The number of explosive ordinance crews at the base.
34. RRR_Crews - The number of rapid runway repair crews available at the base.
35. (+) Vis_To_Enemy - Visibility of base to the enemy.
36. Status - The status of the base (active, overrun, etc.).
37. No_Times_Atck - Total number of times the base has been attacked.
38. Intel_Index - Intelligence index of the base.
39. Alt_Bases - Linked list of alphanumeric designators of base's alternate airbases.
40. Base_Weapons - Linked list of weapons at the base.
    - Designation - The alphanumeric designation of the weapon type.
    - Quantity - The quantity of the weapon the base has.
41. Base_Ac_Avail - Linked list of aircraft available at the base.
    - Designation - The alphanumeric designation of the aircraft type.
    - Quantity - The quantity of the available aircraft the base has.
42. Base [illegible]_Maint - Linked list of aircraft in maintenance at the base.
    - Designation - The alphanumeric designation of the aircraft type.
    - Quantity - The quantity of the maintenance aircraft the base has.
    - Maint_Time_Remaining - The amount of time the aircraft have remaining before maintenance is completed and can be returned to available status.

- AIRBASE COMPONENT

  1. Designation - The alphanumeric designation of the airbase component.
  2. Target_Weight - (target_wgt) The weight of the component.
  3. Length - Length of the component.
  4. Width - Width of the component.

- AIRCRAFT

  1. Designation - The alphanumeric designation of the aircraft type.
  2. Force - The color designation of the aircraft type.
  3. (+) Common_Name - The Nato common name of the aircraft type.
  4. Night_Cap - (night_capability) The percentage of the aircraft's full capability that can be used at night.
  5. Weather_Cap - (wx_capability) The minimum weather conditions that the aircraft type can fly a mission in.
  6. Size - (ac_size) The size of the aircraft type.
  7. Avg_Sorties_Per_Week - (sorties_week) The maximum number of sorties per week that the aircraft type can fly.
  8. Search - The diameter of the aircraft type's sensor detection area in kilomenters.
  9. EC - Electronic countermeasure of the aircraft type.
  10. Max_Speed - The maximum speed the aricraft type can fly (km/hr).
  11. Combat_Radius - (raduis) The maximum radius the aircraft type can fly without being refueled.
  12. Loiter_Time - The length of time the aircraft type can circle over a battle area.
  13. Cargo - The cargo capacity of the aircraft type.
  14. Recon_Ability - The reconaissance ability of the aircraft type.
  15. Refuel - (refuelable) Boolean value indicating whether the aircraft type can be refueled in flight.
  16. Maint_Dist - (maintain_dist) Maintenance distribution type for the aircraft type.
  17. Maint_Mean - (maintain_mean) The mean (average) of the maintenance distribution.
  18. Maint_Stand_Dev - (maint_standev) The standard deviation of the aircraft maintenance distribution.
  19. Amt_Spares - (spare_parts) The number of spare parts required by the aircraft type.
  20. Pol - (pol_usage_rate) The amount of petroleum, oil, and lubricants the aircraft type uses per day.

21. Ramp - (ramp_space) The amount of ramp space taken up by the aircraft type.
22. Min_Runway_Needed - (min_runway) The minimum runway length needed by the aircraft type to take off.
23. Air_Air_Rating - (a2a_rating) The air to air combat (dogfight) rating of the aircraft type.
24. Air_Ground_Rating - (a2g_rating) The air to ground attack rating of the aircraft type. The ability of the aircraft type to accurately atack ground targets.
25. Max_Hex_Level - The maximum air hex level that the aircraft type can achieve.
26. Weapons_Release_Level - The level in which the aircraft type will release its weapons and achieve its highest probability of hitting its target.
27. (*) PCL - The aircraft type's preferred conventional loads.
    - Mission - (mission_type) - The mission the aircraft is to fly.
    - Load - Two dimensional array (current weather , hardness of target) that gives the identification of a weapons load. See weapons load object for its attributes.
28. (*) PBL - The aircraft type's preferred biological loads. Same attributes as PCL.
29. (*) PNL - The aircrft type's preferred nuclear loads. Same atributes as PCL.

- AIRCRAFT PACKAGE

1. Mission_Id - The unique aircraft package identifier.
2. Force - The color designation of the aircraft package.
3. (+) HQ - The command headquarters of the aircraft package.
4. Primary_Mission - (mission_type) The primary aircraft package type.
5. (*) Present_Location - The present location of the aircraft package.
6. Target_Id - The unique identifier of the aircraft package's target.
7. (*) Mission_Location - The location of the aircraft package's target.
8. Rqst_Prd_On_Target - The requested period that the aircraft package should hit its target.
9. Rqst_Day_On_Target - The requested day that the aircraft package should hit its target.
10. Actual_Start_Prd - The actual period that the aircraft package hit its target.
11. Actual_Start_Day - The actual day that the aircraft package hit its target.
12. Loiter_Time - The time the aircraft package should circle over its target.
13. Rqst_Return_Prd - The requested period that the aircraft package should return to its bases.
14. Rqst_Return_Day - The requested dat that the aircraft package should return to its bases.

15. Actual_Return_Prd - The actual period that the aircraft package returned to its bases.
16. Actual_Return_Day - The actual day that the aircraft package returned to its bases.
17. Priority - The aircraft package's priority.
18. Activated - Boolean value indicating whether the aircraft package was activated.
19. Rendezvous_Hex - The air hex that the aircraft package start from.
20. Region - The region in which the aircraft package is formed.
21. (+) Distance - The total distance that the aircraft package can fly. This is based on the aircraft in the package which flies the shortest distance.
22. (+) Altitude - The highest level at which the aircraft package can fly (based on the aircraft in the aircraft package with the least amount of altitude capabilities).
23. (+) Speed - The speed at which the aircraft package will fly (the speed of the slowest aircraft in the aircraft package).
24. Ineffective_Reason - The reason the aircraft package was aborted of ineffective.
25. Orbit_Location - The air hex location that the aircraft should center its orbit in.
26. (+) Detected - Boolean value indicating whether the aircraft package has been detected by the enemy's early warning systems.
27. (+) Positive_Id - Boolean value indicating that not only was the aircraft package detected, but was also positively identified by the enemy.
28. (+) Delayed - Boolean value indicating that the aircraft package was delayed.
29. (+) Was_Cancelled - Boolean value indicating that the aircraft packages was cancelled.
30. (+) Warhead - The type of warhead to be used by the aircraft package.
31. (+) Prim_Msn_Level - The level at which the primary aircraft of the aircraft package will delivery their weapons.
32. Hexes_Overflown - Linked list of air hex numbers the aircraft package flew through.
33. Attrit_Per_Air_Hex - Linked list of the aircraft lost per air hex.
    - Air_Hex - The air hex number.
    - Quantity - The number of aircraft lost in the air hex.
34. Prim_Ac_Sched - Linked list of the primary aircraft scheduled for the aircraft package.
    - Designation - The alphanumeric designation of the aircraft type.
    - Quantity - The number of that type of aircraft scheduled.
35. Prim_Ac_Used - Linked list of the primary aircraft actually used to perform the aircraft mission.

- Designation - The alphanumeric designation of the aircraft type.
- Quantity - The number of that type of aircraft used.
- The_Airbase_No - The base identification the aircraft originated from.

36. Escort_Or_Cap_Ac_Sched - Linked list of the escort and close air patrol (CAP) aircraft scheduled for the aircraft package. Same attributes as Prim_Ac_Sched.
37. Escort_Or_Cap_Ac_Used - Linked list of the escort and CAP aircraft actually used to perform the aircraft mission. Same attributes as Prim_Ac_Used.
38. Sead_Ac_Sched - Linked list of the suppression of enemy air defense (SEAD) aircraft scheduled for the aircraft package. Same attributes as Prim_Ac_Sched.
39. Sead_Ac_Used - Linked list of the SEAD aircraft actually used to perform the aircraft mission. Same attributes as Prim_Ac_Used.
40. Ecm_Ac_Sched - Linked list of the electronic countermeasure (EC) aircraft scheduled for the aircraft package. Same attributes as Prim_Ac_Sched.
41. Ecm_Ac_Used - Linked list of the EC aircraft actually used to perform the aircraft mission. Same attributes as Prim_Ac_Used.
42. Refuel_Ac_Sched - Linked list of the refueling aircraft scheduled for the aircraft package. Same attributes as Prim_Ac_Sched.
43. Refuel_Ac_Used - Linked list of the refueling aircraft actually used to perform the aircraft mission. Same attributes as Prim_Ac_Used.

- CHEMICAL WEAPON

  1. Designation - The alphanumeric designator of the chemical weapon.
  2. Force - The color designation of the chemical weapon.
  3. Persistence - (persistence_time) The time the chemical weapon is effective after detination.
  4. Lethality - The lethality of the chemical weapon.
  5. CEP - The circular error of probability of the chemical weapon.

- CLOCK

  1. Period - The session period.
  2. Cycle - Determines whether the session period is day or night.

- FORCES

  1. Country - The country participating in the simulation.
  2. Force - The force (red, blue, neutral) the country fights on.

- GROUND COMPONENT

  1. Designation - The alphanumeric designator of the ground component type.

2. Ammo_Usage_Rate - The amount of ammunition the ground componet type uses on a daily basis.
3. Pol_Usage_Rate - The amount of petroleum, oil, and lubricants that the ground component type uses per day.
4. Hardware_Usage_Rate - (hw_usage_rate) The amount of hardware used by the ground component type on a daily basis.
5. Target_Weight - (target_wgt) The weight of the ground component type.
6. Firepower_Weight - (firepower) The firepower the ground component type.
7. Length - The length of the ground component type.
8. Width - The width of the ground component type.

- GROUND HEX
  1. Hex_Weather - (actual_wz) The actual weather in the ground hex.
  2. Weather_Zone - (wz) The weather zone the ground hex is located in.
  3. Force - The color designation of the ground hex.
  4. Mission - (army_mission_type) The mission of the ground hex.
  5. (*) In_Contact - Boolean value indicating whether the ground hex is in contact with an opposing ground hex (adjacent hexes have opposing forces located in them).
  6. (*) In_Attrition - Boolean value indicating whether the ground hex's land units are in attrition.
  7. CP_Out - (cpo) The combat power that the ground hex projects out towards its surrounding ground hexes.
  8. CP_In - (cpi) The combat power that the ground hex has projected into it from its surrounding ground hexes.
  9. Attrition - Attrition rate of the ground hex.
  10. SAI - The aggregated surface-to-air index for the units located in the ground hex.
  11. Intel_Index - Intelligence index of the ground hex.
  12. Persistence - (persistence_time) The remaining time the ground hex will feel effects of a nuclear or a chemical weapons attack.
  13. EC - Electronic countermeasure of the ground hex.
  14. Center_Hex - The air hex located directly over the ground hex.
  15. Sides - Array of the six sides of the ground hex (N, NE, SE, S, SW, NW). Each side has the following attributes.

      - Hexside_No - (neighbor_id) The unique alphanumeric label that identifies the common border between two ground hexes.

- Feba - Boolean value indicating if side is part of the feba.
- River - The width of the river at the hex side.

16. Pie_Pieces - (pie_trafficability) Array of the six hex pie pieces that indicate the trafficability of the pie piece. The trafficability is the difficulty of travel between the center of the ground hex to the side.
17. Unit_List - Linked list of land unit identifications located in the ground hex.
18. Base_List - Linked list of airbase identifications located in the ground hex.
19. Depot_List - Linked list of depot identifications located in the ground hex.

- HARDNESS

1. Target - (target_type) The alphanumeric identification of the target type.
2. PK_Value - (hardness) The hardness of the target type.

- LAND UNIT

1. Land_Unit_Id - (unit_id) The unique alphanumeric identifier for the land unit.
2. (+) Corps_Id - The identifier of the corps the land unit belongs to.
3. (+) Parent_Unit - The identifier of the land unit's parent unit.
4. (+) Full_Designator - The full alphanumeric designator of the land unit.
5. (+) Abbrev_Designator - The abbreviated designator of the land unit.
6. (+) Country - The country the land units belongs to.
7. Type_Of_Unit - (unit_type) The type of land unit.
8. Force - The color designation of the land unit.
9. Present_Location - (location) The ground hex in which the land unit is located.
10. (*) Current_Mission - The current mission the unit is executing.
    - Mission_Location - The location of the unit's current mission.
    - Mission - The type of mission being performed. Values : (primary, override, supply, none).
11. Msn_Eff_Day - The day in which the land unit becomes active.
12. Region - Region of the hex the land unit is located in.
13. (*) Hex_Dir - The direction the land unit is headed.
14. (*) Move_Allowed - Array of the directions the land unit is allowed to move.
15. (*) In_Attrition - Boolean value indicating the land unit is in attrition with enemy units.
16. Firepower - The firepower of the land unit.
17. Combatpower - (combat_power) The combat power of the land unit.
18. Attrition - The attrition rate of the land unit.

19. Total_Pol - The total amount of petroleum, oil, and lubricants the land unit has.
20. Pol_Max_Capacity - The maximum amount petroleum, oil and lubricants the land unit can hold at one time.
21. (*) Pol_Usage_Rate - The amount of petroleum, oil, and lubricants the land unit uses in a day.
22. Total_Ammo - The total amount of munitions the land unit has.
23. Ammo_Max_Capacity - The maximum amount of munition the land unit can hold at one time.
24. (*) Ammo_Usage_Rate - The amount of munition the land unit uses in a day.
25. Total_Hardware - The total amount of hardware the unit has.
26. Hardware_Max_Capacity - The maximum amount of hardware the unit can hold at one time.
27. (*) Hardware_Usage_Rate - The amount of hardware the unit uses in a day.
28. (*) In_Contact - Boolean value indicating whether the unit is in contact with enemy units.
29. Intel_Index - Intelligence index of the land unit.
30. Intel_Filter - Intelligence filter of the land unit.
31. (*) Was_Intelled - Boolean value indicating whether or not the land unit was intelled.
32. Breakpt - (breakpoint) The maximum combatpower the level that a land unit must sustain in order to engage in battle. When the combat power of the land unit falls below this value, it automatically withdrawals from combat.
33. Grid_Time - If the land unit is moving the grid time is either the amount of time a unit needs to cross a ground hex or the amount of time needed to overcome an obstacle at a border. If the unit is stationary, it is the amount of time the unit has acquired in the ground hex.
34. Day_Last_Intelled - The last day intelligence was performed on the land unit.
35. Prd_Last_Intelled - The last period intelligence was performed on the land unit.
36. Loc_Last_Intelled - The ground hex on which the land unit last had any reconaissance performed on it.
37. Depot_Spt - The identification of the unit that provides the land unit depot support.
38. (*) Under_Chem_Nuc_Atk - Boolean value indicating whether the land unit is under a chemical or nuclear attack.
39. Mopp_Posture - Mopp posture of the land unit.
40. Troop_Quality - The experience, leadership, and fighting skill of the land unit.
41. Groundspeed - The top speed that the land unit can move across a ground hex.

42. Fuel_Trucks - The number of fuel trucks the land unit possesses.
43. Ammo_Trucks - The number of ammunition trucks the land unit possesses.
44. Water - The amount of water the land unit possesses.
45. Water_Percent - The percentage of water used by the land unit on a daily basis.
46. Water_Trucks - The number of water truck the land unit possesses.
47. Engineers - The number of engineers the land units possesses.
48. Eng_Vehicles - The number of engineering vehicles the land unit possesses.
49. Status - The status of the land unit.
50. Unit_Size - The size of the land unit.
51. (+) Vis_To_Enemy - Boolean field which describes whether the land unit will appear on the computer of an opposing player.
52. Supported_Units - Linked list of the units the land unit supports.
    - Unit_No - The identification of the land unit to be supported.
    - Percent_Support - The amount of support that land unit recieves.
53. Missions - Linked list of the land units primary missions. See LAND UNIT MISSION object class for its attributes.
54. Override_Missions - Linked list of the land units override missions. See LAND UNIT MISSION object class for its attributes.
55. Components - Linked list of the land units ground components.
    - Designation - The alphanumeric designation of the ground component type.
    - Quantity - The quantity the land unit has of that ground component type.
56. Weapons - Linked list of the land units weapons. These include both surface-to-surface and surface-to-air weapon types.
    - Designation - The alphanumeric designation of the weapon type.
    - Weapons_Quantity - The quantity the land unit has of that weapon type.
    - Launchers_Quantity - The quantity of launchers the unit has to support the weapon type.
57. Radars - Linked list of the land units radars. See RADAR object class for its attributes.

- LAND UNIT MISSION

  1. Order_Id - The land unit mission identifier.
  2. Target_Id - The unique identifier of the target.
  3. Mission - (army_mission_type) The mission type.
  4. (*) Mission_Location - The location of the target.
  5. Day - The day on which the mission is to be executed.

6. Period - The period on which the mission is to be executed.

- NUCLEAR WEAPON

  1. Designation - The alphanumeric designation of the nuclear weapon.
  2. Yield - The yield of the nuclear weapon.
  3. Force - The color designation of the nuclear weapon.
  4. CEP - The circular probability of the nuclear weapon.
  5. Persistence - (persistence_time) The amount of time the effects of the nuclear weapon will be felt after detenation.

- OBSTACLE

  1. Obstacle_Id - The unique identifier of the obstacle.
  2. Hexside_No - The unique identifier of the side the obstacle is located.
  3. Obstacle - The type of obstacle.
  4. Obs_Diff - (difficulty) The difficulty of the obstacle to overcome.
  5. Vis_To_Enemy - The knowledge or visibility the enemy has of the obstacle.

- PIPELINES

  1. Pipeline_Id - The unique identifier of the pipeline.
  2. Hex_No - The ground hex the pipeline piece is located.
  3. Pie_Piece - Indicates which of the six pie pieces the pipeline piece is located.
  4. Product - The product carried by the pipeline piece.
  5. Name - The name of the pipeline.
  6. Flow - Boolean value indicating whether the pipeline piece is flowing or not.

- WEAPONS LOAD

  1. Weapons_Load_Id - (load_id) The weapons load identifier.
  2. Designation - The alphanumeric designation of the weapon type.
  3. Quanitity - The quantity of that weapon type that comprise the weapons load.

- RADAR

  1. Type_Radar - (radar_type) The type of radar used with surface-to-air missile systems.
  2. Quality - The quality of the radar system.
  3. Quantity - The quantity the land unit has of that radar type.

- RAILROAD

1. Railroad_Id - The unique identifier of the railroad.
2. Hex_No - The ground hex the railroad piece is located.
3. Pie_Piece - Indicates which of the six pie pieces the railroad piece is located.
4. Name - The name of the railroad.
5. Flow - Boolean value indicating whether the railroad piece is flowing or not.

- ROAD

  1. Road_Id - The unique identifier of the road.
  2. Hex_No - The ground hex the road piece is located.
  3. Pie_Piece - Indicates which of the six pie pieces the road piece is located.
  4. Name - The name of the road.
  5. Size - (road_size) The width of the road piece.
  6. Flow - Boolean value indicating whether the road piece is flowing or not.

- RUNWAY

  1. Base_Id - Base identification of location of runway.
  2. Runway - The unique identifier of the runway.
  3. Condition - (difficulty) The condition of the runway.
  4. Current_Length - The current length of the runway.
  5. Max_Length - The maximum length of the runway.

- SATELLITE

  1. Satellite_Id - The unique identifier of teh satellite.
  2. Designation - The alphanumeric designation of the satellite.
  3. Force - The color designation of the satellite.
  4. Location - The air hex identification the satellite is located in.
  5. Sat_Type - The satellite type.
  6. Status - The status of the satellite.
  7. Speed - The speed of the satellite.
  8. Direction - The current direction in which the satellite is moving.
  9. Orbit - The satellite orbit type.
  10. Sat_Delay - (delay) The time delay required by the satellite from the time of its launching to the time it is operational in orbit.

- SUPPLY TRAIN

  1. All attributes plus the following attributes.

2. Supply_Train_Id - The identifier which links it with the land unit it belongs to.
3. Resupply_Unit_Id - The land unit idendifier which resupplies the supply train.
4. ST_Missions - Linked list of the supply trains missions. See SUPPLY TRAIN MISSION object class for its attributes.
5. Tot_Cap - Total amount of supplies the supply train can carry at one time.
6. In_Use - Boolean value indicating whether the supply train is currently being used or not.
7. Type_Of_ST - (supply_type) The type of supply train.
8. Trans_Mode - The supply train' type of transportation.
9. Total_Pol - The total amount of petroleum, oil, and lubricants the supply train currently carries.
10. Total_Ammo - The total amount of ammunitions the supply train currently carries.
11. Total_Hardware - The total amount of hardware the supply train currently carries.
12. Total_Spares - The total amount of spares the supply train currently carries.

- SUPPLY TRAIN MISSION

1. Order_Id - The unique identifier of the order.
2. Type_Of_Supply - (designation) The type of supplies to deliver.
3. Delivery_Quantity - The amount of that supply type to deliver.

- SURFACE-TO-SURFACE MISSILE

1. Designation - The alphanumeric designation of the surface-to-surface missile.
2. Force - The color designation of the surface-to-surface missile.
3. Warhead - (class) The type of warhead the surface-to-surface missile has.
4. Leathality_Radius - (lethal_area) The lethal blast radius of the surface-to-surface missile.
5. CEP - The circular error of probability of the surface-to-surface missile.
6. PK_Hard_Point_Type - (pk_hard) The probability of destroying a hardened target with a direct hit.
7. PK_Med_Point_Type - (pk_med) The probability of destroying a medium-hardened target with a direct hit.
8. PK_Soft_Point_Type - (pk_soft) The probability of destroying an unhardened target with a direct hit.
9. Min_Range - The mimimum effective range that the surface-to-surface missile can be used to hit a target.

10. Max_Range - The maximum range a surface-to-surface missile can fly.
11. Launcher_Rounds - (rnds_per_launcher) The number of missile rounds that are normally loaded on a single launching platform.
12. Reload_Time - The minimum time necessary to reload a missle launcher after a missile has been fired.

- SURFACE-TO-AIR MISSILES

    1. Designation - The alphanumeric designation of the surface-to-air missile.
    2. Force - The color designation of the surface-to-air missile.
    3. Warhead - (class) The type of warhead the surface-to-air missile has.
    4. Slow_High - Air Defense Artillery value for a slow moving attack aircraft with high probability.
    5. Slow_Low - Air Defense Artillery value for a slow moving attack aircraft with low probability.
    6. Fast_High - Air Defense Artillery value for a fast moving attack aircraft with high probability.
    7. Fast_Low - Air Defense Artillery value for a fast moving attack aircraft with low probability.
    8. SSPK - Single shot probability of kill.
    9. Miss_Radar_Range - Array of the radar's range and the missile's range for each hex level.
        - Missile_Range - The range of the surface-to-air missile.
        - Radar_Range - The range of the radar used by the surface-to-air missile.
    10. Launcher_Rounds - (rnds_per_launcher) The number of missile rounds that are normally loaded on a single launching platform.
    11. Reload_Time - The minimum time necessary to reload a missle launcher after a missile has been fired.
    12. Weather_Min - (weather_minimum) The minimum weather in which the surface-to-air missile will be effective.

- WEATHER

    1. Good_Percent - (forecast_good) The probability of having good weather, expressed as a percentage.
    2. Fair_Percent - (forecast_fair) The probability of having fair weather, expressed as a percentage.
    3. Weather - (actual_wx) The actual calculated weather.

Appendix B. *Object Classes and Their Operations*

This appendix lists all of the operations that are required of each object classes (alphabetic order). The operations shown indicate what is seen in the specfications of each package and do not show operations (functions and subprocedures) that compose the specfication operations.

- AIR HEX
  1. GET_AIR_GRID - Reads air hex information form the flat file.
  2. LOAD_AIR_HEX_WEATHER - Loads the weather into each of the air hexes.
  3. ADD_ACPKG_TO_HEX - Adds an aircraft package identification number to the hex.
  4. DELETE_ACKG_FROM_HEX - Deletes an aircraft package identification number from a hex.
  5. ADD SATELLITE_TO_HEX - Adds a satellite identification number to a air hex.
  6. DELETE_SATELLITE_FROM_HEX - Deletes a satellite identification number from a air hex.
  7. Functions that retreive air hex attributes. For example, WEATHER_OF, which will return the weather of the specified air hex number.
  8. Update procedures. For example, UPDATE_ATTRITION, which will update the attrition of the specified given a air hex number to the new specified attrition value.
  9. SET_FIRST_BLUE_ACPKG_IN_HEX - Sets an internal pointer to the first blue aircraft package in a air hex.
  10. SET_FIRST_RED_ACPKG_IN_HEX - Sets an internal pointer to the first red aircraft package in a air hex.
  11. NEXT_ACPKG_IN_HEX - Returns the next blue or red aircraft package identification number (depending on what set procedure was last invoked) located in the air hex. The boolean variable Another_Acpkg indicates whether there is another aircraft package or not.
  12. SET_FIRST_SATELLITE_IN_HEX - Sets an internal pointer to the first satellite in a air hex.
  13. NEXT_SATELLITE_IN_HEX - Returns the next satellite identification number located in the air hex.
  14. WRITE_AIR_GRID - Regenerates flat file with modified data.
- AIR SIMULATION
  1. INITIALIZE_AIR - Initializes all the objects related to the air battle.

2. SET_UP - Provides the necessary set-up procedures before each simulation time slice.
3. PERFORM_MISSIONS - Determines current missions, formulates possible aircraft packages and executes the mission.
4. WRITE_DATA - Regenerates all the flat files related to the air battle.

- AIRCRAFT

  1. GET_AIRCRAFT - Retrieves all the aircraft types from flat file.
  2. Functions for each aircraft attribute that will return the desired attribute of the specified aircraft type.
  3. PCL_OF - Returns the preferred weapons load number if the aircraft is flying a conventional mission.
  4. PBL_OF - Returns the preferred weapons load number if the aircraft is flying a biological mission.
  5. PNL_OF - Returns the preferred weapons load number if the aircraft is flying a nunclear mission.

- AIRCRAFT PACKAGE

  1. GET_AIRCRAFT_PACKAGES - Reads the aircraft packages from the flat file.
  2. SET_FIRST_AIRCRAFT_PACKAGE - Sets an internal pointer to the first aircraft package.
  3. NEXT_AIRCRAFT_PACKAGE - Returns the next aircraft package identification number. The boolean variable Another_Aircraft_Package indicates whether there is another aircraft package or not.
  4. FORM_A_ACPKG - Performs the necessary operations to form a aircraft package.
  5. DELETE_A_ACPKG - Deletes a aircraft package from the list.
  6. SET_FIRST_SCHED_AC - This operation will set a pointer to the first scheduled aircraft of the designated aircraft role (primary, escort, SEAD, ECM, refuel) of a aircraft package.
  7. NEXT_SCHED_AC - Returns the next acheduled aircraft designation and quantity. Will return a boolean value stating whether there are more scheduled aircraft.
  8. SET_FIRST_MISSION_AC - Sets a pointer to the first mission aircraft of a designated aircraft package. The aircraft role indicates whether the pointer will be set to the primary, escort, sead, ec, or refueling aircraft.
  9. NEXT_MISSION_AC - Returns the next aircraft type, quantity, and base from which the aircraft originated.

10. ADD_MISSION_AC - Adds an aircraft type, quantity, and originating base to either the primary, escort, sead, ec, or refueling aircraft of a designated aircraft package.
11. CLEAR_ALL_MISSION_AC - Removes all the mission aircraft of a designated aircraft package.
12. SET_FIRST_HEX_OVERFLOWN - Sets a pointer to the first hex the designated aircraft package flew throung during its mission.
13. NEXT_HEX_OVERFLOWN - Returns the next hex the aircraft packge flew through. A boolean value indicates the last of the hexes the aircraft package traversed.
14. ADD_OVERFLOWN_HEX - Adds a hex the the list of hexes a aircraft package traversed.
15. SET_FIRST_ATTRITED_AIRHEX - Sets a pointer to the first hex the aircraft package lost aircraft in combat.
16. NEXT_ATTRITED_AIRHEX - Returns the type of aircraft and quantity lost along with the location (hex number) where the loses occurred.
17. ADD_ATTRITED_AIRHEX - Adds a aircraft type, quantity lost and the location of aircraft lost by an aircraft package.
18. Functions to retrieve each aircraft package attribute and procedures to update each aircraft package attribute.
19. WRITE_AIRCRAFT_PACKAGES - Regenerates the aircraft package flat files.

- BASE (DEPOT)

  1. GET_BASES - Reads base information from flat files.
  2. GET_DEPOTS - Reads depot information from flat files.
  3. SET_FIRST_BASE - Sets a pointer to the first base.
  4. NEXT_BASE - Returns the base identification number as long as there is another base, otherwise a boolean variable is set to false to indicate that the last base was visited.
  5. SET_FIRST_DEPOT - Sets a pointer to the first depot.
  6. NEXT_DEPOT - Returns the depot identification number as long as there is another depot, otherwise a boolean vairable is set to false to indicate that the last depot was visited.
  7. INCREASE_POL_SOFT_OF_BASE - Increases the fuel in soft storage of a designated base or depot.
  8. DECREASE_POL_SOFT_OF_BASE - Decreases the fuel in soft storage of a designated base or depot.
  9. INCREASE_POL_HARD_OF_BASE - Increases the fuel in hard storage of a designated base or depot.

10. DECREASE_POL_HARD_OF_BASE - Decreases the fuel in hard storage of a designated base or depot.
11. INCREASE_SPARES_OF_BASE - Increases the amount of spares of a designated base or depot.
12. DECREASE_SPARES_OF_BASE - Decreases the amount of spares of a designated base or depot.
13. INCREASE_BASE_AVAIL_AC - Increases the amount of available aircraft at a base or depot by the designated aircraft type and quantity.
14. DECREASE_BASE_AVAIL_AC - Decreases the amount of available aircraft at a base or depot by the designated aircraft type and quantity.
15. INCREASE_BASE_MAINT_AC - Increases the amount of maintenance aircraft at a base or depot by the designated aircraft type and quantity.
16. INCREASE_BASE_WEAPONS - Increases the amount of weapons of a base a depot by the designated weapon type and quantity.
17. DECREASE_BASE_WEAPONS - Decreases the amount of weapons of a base a depot by the designated weapon type and quantity.
18. SET_FIRST_AVAIL_AC - Sets a pointer to the first aircraft available of a designated base or depot.
19. NEXT_AVAIL_AC - Returns the next available aircraft type and quantity at the base or depot. A boolean value indicates when the last available aircraft was visited.
20. QUANT_OF_AVAIL_AC_TYPE - Returns the quantity available of a particular aircraft type at a designated base or depot.
21. SET_FIRST_MAINT_AC - Sets a pointer to the first aircraft in maintenance at a designated base or depot
22. NEXT_MAINT_AC - Returns the next maintenance aircraft type, quantity and time remaining in maintenance. A boolean value indicates when the last maintenance aircraft was visited.
23. QUANT_OF_MAINT_AC_TYPE - Returns the quantity of maintenance aircraft of a particular aircraft type at a designated base or depot.
24. SET_FIRST_WEAPON - Sets a pointer to the first weapon of a designated base or depot.
25. NEXT_WEAPON - Returns the next weapon and quantity located at the base or depot. A boolean value indicates that all the weapons have been visited.
26. QUANT_OF_WEAPON_TYPE - Returns the quantity of a particular weapon type a base or depot possesses.
27. SET_FIRST_ALT_BASE - Sets a pointer to the first alternate base of a designated base or depot.
28. NEXT_ALT_BASE - Returns the unique identification of the next alternate base. A boolean value indicates there are not any more alternate bases.

29. DETERMINE_REGION_AIRCRAFT - Creates a list of all the available aircraft located in a particular region.
30. SET_FIRST_REGION_AIRCRAFT - Sets a pointer to the first available aircraft in the region.
31. NEXT_REGION_AIRCRAFT - Returns the next type of aircraft, quantity, and base identification the aircraft is located. A boolean value indicates that all the region aircraft have been visited.
32. Functions to retrieve base attributes.
33. Procedures to update base attributes.
34. WRITE_BASES_DEPOTS - Regenerates the base and depot flat files.

- CLOCK
    1. GET_CLOCK - Reads the current day, current period and the number of time slices the simulation will execute.
    2. GET_CYCLES - Reads the day and night cycles.
    3. CURRENT_CYCLE - Returns whether the given period is day or night.
    4. WRITE_CLOCK - Regenerates the clock flat file.
- FORCES
    1. GET_COUNTRIES - Reads the list countries and the force (red, blue) they fight with.
    2. GET_FORCE_OF_COUNTRY - Returns the force of a given country.
- GROUND COMPONENT
    1. GET_GROUND_COMPONENTS - Reads all the ground components available from the flat files.
    2. Functions for each attribute that will return the attribute for a specified ground component type.
- GROUND HEX
    1. GET_GROUND_GRID - Reads the ground grid information from the flat files.
    2. APPLY_FS - Applies field artillery, aviation, and air defense fire support with the combat power of the units being provided the support.
    3. APPLY_CP - Sets up the numbers required for calculating attrition. It also takes all the firepower in a hex and applies it equally to all adjacent hexes which contain opposing forces.
    4. ATTRIT - Performs attrition on every unit in combat.
    5. LOAD_GROUND_HEX_WEATHER - Loads the weather of the specified weather period and day into each of the hexes.

6. ADD_UNIT_TO_HEX - Adds a land units unique identification to the designated ground hex.
7. DELETE_UNIT_FROM_HEX - Deletes a land units identification from the designated ground hex.
8. ADD_BASE_TO_HEX - Adds a bases unique identification to the designated ground hex.
9. ADD_DEPOT_TO_HEX - Adds a depots unique identification to the designated ground hex.
10. SET_FIRST_UNIT_IN_HEX - Sets a pointer to the first land unit located in the designated ground hex.
11. NEXT_UNIT_IN_HEX - Returns the next land unit indentification located in the ground hex. A boolean value indicates that there are not any more land units in the ground hex.
12. SET_FIRST_BASE_IN_HEX - Sets a pointer to the first base located in the designated ground hex.
13. NEXT_BASE_IN_HEX - Returns the next base identification located in the ground hex. A boolean value indicates that there are not any more bases in the ground hex.
14. SET_FIRST_DEPOT_IN_HEX - Sets a pointer to the first depot located in the designated ground hex.
15. NEXT_DEPOT_IN_HEX - Returns the next depot identification located in the ground hex. A boolean balue indicates that there are not any more depots in the groung hex.
16. Functions to retrieve attributes of a specified ground hex.
17. Procedures to update attributes of a specified ground hex.
18. WRITE_GROUND_GRID - Regenerates the ground grid flat files.

- HARDNESS

  1. GET_HARDNESS - Reads the hardnesses of each of the entites that can be targeted.
  2. HARDNESS_OF - Returns the hardness of the designated target type.

- LAND SIMULATION

  1. INITIALIZE_LAND - Initializes all the objects related to the land battle.
  2. SET_UP - Provides the necessary set up procedures before each simulation time slice.
  3. PROVIDE_LOGISTICS_SUPPORT - Provides the logistics support of supply trains and mini depots.
  4. MOVEMENT - Controls the overall movement of units.

5. ATTRITION - Determines units that are in combat and applies attrition rates to the units.
6. INTELLIGENCE - Controls the land unit intelligence index reduction and itelligence operations.
7. WRITE_DATA - Regenerates all the flat files related to the land battle.

- LAND UNIT

1. GET_LAND_UNITS - Reads all the land unit information form the flat file.
2. SET_FIRST_LAND_UNIT - Sets a pointer to the first land unit.
3. NEXT_LAND_UNIT - Returns the next land units unique identification number. A boolean value indicates when all the land units have been visited.
4. DELETE_A_LAND_UNIT - Deletes a specified land unit from the simulation.
5. MOVE_IN_GRID - The procedure will move all land units as necessary. As long as a unit is performing a movement or attack mission, it is not in contact with enemy forces, and it has enough fuel, the uint will move at its prescribed rate.
6. DETERMINE_FIREPOWERS - Calculates the firepowers of all land units.
7. DETERMINE_AMMO_HW_POL_USAGE_RATES - Caluculates the ammunition, harwdare and fuel usage rates for all land units.
8. RESUPPLY_UNITS - Provides the automatic resupply of ammunitions, hardware, and fuel from mini depots to front line fighting units.
9. REDUCE_INTEL - Reduces the intel index of all land units by the prescribed amount.
10. INCREASE_LAND_UNITS_COMPONENTS - Increases the amount of components a land unit has by the designated component type and quantity.
11. DECREASE_LAND_UNITS_COMPONENTS - Decreases the amount of components a land unit has by the designated component type and quantity.
12. INCREASE_LAND_UNITS_WEAPONS - Increases the amount of weapons a land unit has by the designated weapon type and quantity.
13. DECREASE_LAND_UNITS_WEAPONS - Decreases the amount of weapons a land unit has by the designated weapon type and quantity.
14. INCREASE_LAND_UNITS_LAUNCHERS - Increases the amount of launchers a land unit has by the designated launcher type and quantity.
15. DECREASE_LAND_UNITS_LAUNCHERS - Decreases the amount of launchers a land unit has by the designated launcher type and quantity.
16. INCREASE_AMMO_OF_UNIT - Increases the amount of ammunitions a land unit has by the designated quantity.
17. DECREASE_AMMO_OF_UNIT - Decreases the amount of ammunitions a land unit has by the designated quantity.

18. INCREASE_HARDWARE_OF_UNIT - Increases the amount of hardware a land unit has by the designated quantity.
19. DECREASE_HARDWARE_OF_UNIT - Decreases the amount of hardware a land unit has by the designated quantity.
20. INCREASE_POL_OF_UNIT - Increases the amount of fuel a land unit has by the designated quantity.
21. DECREASE_POL_OF_UNIT - Decreases the amount of fuel a land unit has by the designated quantity.
22. SET_FIRST_SUPPORTED_UNIT - Sets a pointer to the first supported unit of the designated land unit.
23. NEXT_SUPPORTED_UNIT - Returns the next land unit identification number that is supported and the amount of support that unit will receive. A boolean value indicates there are no more supported units.
24. SET_FIRST_COMPONENT - Sets a pointer to the first component of the designated land unit.
25. NEXT_COMPONENT - Returns the next component type and quantity. A boolean value indicates the last component was visited.
26. SET_FIRST_WEAPON - Sets a pointer to the first weapon of the designated land unit.
27. NEXT_WEAPON - Returns the next weapon type, the quantity of weapons, and the quanitity of launchers the unit has to support that weapon type. A boolean value indicates that the last weapon was visited.
28. SET_FIRST_RADAR - Sets a pointer to the first radar of the designated land unit.
29. NEXT_RADAR - Returns the next radar type, quality, and quantity. A boolean value indicates that the last radar was visited.
30. SET_FIRST_MISSION - Sets a pointer to the first mission of the designated land unit.
31. NEXT_MISSION - Returns the next mission number. A boolean value indicates that the last mission was visited.
32. SET_FIRST_OVERRIDE_MISSION - Sets a pointer to the first override mission of the designated land unit.
33. NEXT_OVERRIDE_MISSION - Returns the next override mission number. A boolean value indicates that the last override mission was visited.
34. Functions to retrieve attributes of a specified land unit.
35. Procedures to update attributes of a specified land unit.
36. WRITE_LAND_UNITS - Regenerates the flat files containing the land units information.

- OBSTACLE

1. GET_OBSTACLES - Reads all the obstacle information from the flat files.
2. SET_FIRST_OBSTACLE - Sets a pointer to the first obstacle of the designated hexside number.
3. NEXT_OBSTACLE - Returns the next obstacle number of the hexside. A boolean value indicates that there are not any more obstacles at the hexside.
4. MAX_OBSTACLE_DIFF_OF - Returns the maximum difficulties of all the obstacles of the disignated hexside number.
5. VALUE_OF_OBS_DIFF - Returns the specific value number of an obstacle type.
6. Functions to retrieve attributes of a specified obstacle.
7. Procedures to update attributes of a specified obstacle.
8. WRITE_OBSTACLES - Regenerates the flat files containing obstacle information.

- PIPELINE

1. GET_PIPELINES - Reads all the pipeline information from the flat files.
2. SET_FIRST_PIPELINE_SEGMENT_OF_PIE_PIECE - Sets a pointer to the first pipeline segment located in the designated hex number and pie piece.
3. SET_FIRST_SEGMENT_OF_PIPELINE - Returns the next pipeline segment in a hex pie piece or the next pipeline segment of a larger pipeline.
4. NEXT_PIPELINE_SEGMENT - Returns the next pipeline segment number in a hex pie piece or the next pipeline segment of a larger pipeline. A boolean value is set when there are not any more pipeline segments.
5. Functions to retrieve attributes of a specified pipeline segment.
6. Procedures to update attributes of a specified pipeline segment.
7. WRITE_PIPELINES - Regenerates the flat files containing all pipeline information.

- RAILROAD

1. GET_RAILROADS - Reads all the railroad information from the flat files.
2. SET_FIRST_RAILROAD_SEGEMNT_OF_PIE_PIECE - Sets a pointer to the first railroad segment located in the designated hex number and pie piece.
3. SET_FIRST_SEGMENT_OF_RAILROAD - Sets a pointer to the first railroad segment of a larger railroad.
4. NEXT_RAILROAD_SEGMENT - Returns the next railroad segment number in a hex pie piece or the next railroad segment of a larger railroad. A boolean value is set when all the railroad segments have been visited.
5. Functions to retrieve attributes of a specified railroad segment.
6. Procedures to update attributes of a specified railroad segment.

7. WRITE_RAILROADS - Regenerates the flat files containing all railroad information.

- ROAD

  1. GET_ROADS - Reads all the road information from the flat files.
  2. SET_FIRST_ROAD_SEGMENT_OF_PIE_PIECE - Sets a pointer to the first road segment located in the designated hex number and pie piece.
  3. SET_FIRST_SEGMENT_OF_ROAD - Sets a pointer to the first road segment of a larger road.
  4. NEXT_ROAD_SEGMENT - Returns the next road segment number in a hex pie piece or the next railroad segment of a larger railroad. A boolean value is set when all the road segments have been visited.
  5. Functions to retrieve attributes of a specified road segment.
  6. Procedures to update attributes of a specified road segment.
  7. WRITE_ROADS - Regenerate the flat files containing all road information.

- RUNWAY

  1. GET_RUNWAYS - Reads the simulation runways from the flat file.
  2. SET_FIRST_BASE_RUNWAY - Sets a pointer to the first runway of the specified base.
  3. NEXT_BASE_RUNWAY - Returns the next runway located at the base. A boolean value indicates whether there is another runway at the base or not.
  4. BASE_MAX_RUNWAY_LENGTH_OF - Returns the longest runway length of the runways at a particular base.
  5. Functions and procedures to obtain and update each attribute of a particular runway number.
  6. WRITE_RUNWAYS - Writes runway information out to flat file.

- SATELLITE

  1. GET_SATELLITES - Reads all the satellite information from the flat files.
  2. SET_FIRST_SATELLITE - Sets a pointer to the first satellite in the simulation.
  3. NEXT_SATELLITE - Returns the next satellite number. A boolean value is set when all the satellites have been visited.
  4. Functions and procedures to obtain and update each attribute of a given satellite number.
  5. WRTIE_SATELLITES - Regenerates the flat files containing all the satellite information.

- SUPPLY TRAIN

1. GET_SUPPLY_TRAINS - Reads all the supply train information from the flat files.
2. SET_FIRST_SUPPLY_TRAIN - Sets a pointer to the first supply train in the simulation.
3. NEXT_SUPPLY_TRAIN - Returns the next supply train number. A boolean value is set when all the supply trains have been visited.
4. DELETE_SUPPLY_TRAIN - Deletes a supply train from the simulation.
5. DETERMINE_FULL_LOAD_OF_ST - Returns the total amount of ammunitions, fuel, hardware and spares the supply train needs to fulfill its missions.
6. INCREASE_AMMO - Increases the amount of ammunitions of the supply train by the designated quantity.
7. DECREASE_AMMO - Decreases the amount of ammunitions of the supply train by the designated quantity.
8. INCREASE_POL - Increases the amount of fuel of the supply train by the designated quantity.
9. DECREASE_POL - Decreases the amount of fuel of the supply train by the designated quantity.
10. INCREASE_HW - Increases the amount of hardware of the supply train by the designated quantity.
11. DECREASE_HW - Decreases the amount of hardware of the supply train by the designated quantity.
12. INCREASE_SPARES - Increases the amount of spares of the supply train by the designated quantity.
13. DECREASE_SPARES - Decreases the amount of spares of the supply train by the designated quantity.
14. Functions and procedures to obtain and update each attribute of a given suppy train number.
15. WRITE_SUPPLY_TRAINS - Regenerates the flat files containing all the supply trains information.

- WEAPONS

  1. GET_SAM - Reads all the surface-to-air missile information from the flat files.
  2. GET_SSM - Reads all the surface-to-surface missile information from the flat files.
  3. GET_AAW - Reads all the air-to-air weapon information from the flat files.
  4. GET_AGW - Reads all the air-to-ground weapon information from the flat files.
  5. GET_CHEM - Reads all the chemical weapon information from the flat files.
  6. GET_NUC - Reads all the nuclear weapon information form the flat files.

7. Functions for each attribute and and weapon type that will return the attribute value of the requested weapon designation.

- WEAPONS LOAD

  1. GET_WEAPONS_LOAD - Reads the weapons loads from the flat file.
  2. SET_FIRST_WEAPON_OF_LOAD - Will set a pointer to the first weapon of the given load identification number.
  3. NEXT_WEAPON_OF_LOAD - Returns the next weapon type and quantity that compose a weapons load. A boolean value is used to determine if there are any more weapons for the load.

- WEATHER

  1. GET_WEATHER - Reads in all the weather information from the flat files.
  2. CALC_WEATHER_VAL - Returns the value associated with a particular weather type (good, fair, poor).

Appendix C. *Visibility of Each Object Class*

This appendix lists all of the object classes. Under each object class are two lists. One indicates what the object class's package specification "sees" and the other is what the object class's package body "sees". This listing corresponds to the "with" clauses of the object class's specification and body. The TEXT_IO package is used by every object class and is not shown. If the specification of a package has visibility to another object, so does the package body.

- AIR HEX

  Specification

  1. FORCES
  2. WEATHER
  3. HEX

  Body

  1. SIMULATION_CONSTANTS
  2. DOUBLY_LINKED_LIST

- AIR SIMULATION

  Body

  1. AIR_HEX
  2. AIRCRAFT
  3. AIRCRAFT_PACKAGES
  4. ALGORITHMS
  5. BASE_COMPONENTS
  6. BASES
  7. CLOCK
  8. DOUBLY_LINKED_LIST
  9. GROUND_HEX
  10. HARDNESS
  11. HEX
  12. HISTORY
  13. RUNWAYS
  14. SATELLITES
  15. WEAPONS
  16. WEAPONS_LOAD

- AIRCRAFT

  Specification

  1. FORCES
  2. HARDNESS
  3. WEATHER

  Body

  1. DOUBLY_LINKED_LIST
  2. COMPOSITE_LINKED_LIST

- AIRCRAFT PACKAGES

  Specification

  1. AIRCRAFT
  2. FORCES
  3. HEX
  4. WEAPONS

  Body

  1. BASES
  2. COMPOSITE_LINKED_LIST
  3. DOUBLY_LINKED_LIST
  4. GROUND_HEX
  5. HARDNESS
  6. RUNWAYS
  7. SIMULATION_CONSTANTS
  8. WEAPONS_LOAD

9. WEATHER
10. WEAPONS

- ALGORITHMS

  NONE

- BASE_COMPONENTS

  Body

  1. DOUBLY_LINKED_LIST

- BASES (DEPOT)

  Specification

  1. FORCES
  2. WEATHER

  Body

  1. DOUBLY_LINKED_LIST
  2. COMPOSITE_LINKED_LIST

- CLOCK

  Body

  1. SIMULATION_CONSTANTS

- COMPOSITE_LINKED_LIST

  NONE

- DOUBLY_LINKED_LIST

  NONE

- FORCES

  Body

  1. DOUBLY_LINKED_LIST

- GROUND COMPONENTS

  Body

  1. DOUBLY_LINKED_LIST

- GROUND HEX

  Specification

  1. FORCES
  2. HEX
  3. LAND_UNITS
  4. WEATHER

  Body

  1. DOUBLY_LINKED_LIST
  2. CLOCK
  3. HISTORY
  4. SIMULATION_CONSTANTS

- HARDNESS

  Body

  1. DOUBLY_LINKED_LIST

- HEX

  Body

  1. SIMULATION_CONSTANTS

- HISTORY

  Specification

  1. AIRCRAFT_PACKAGES

  Body

  1. AIRCRAFT
  2. CLOCK
  3. FORCES
  4. LAND_UNITS
  5. SUPPLY_TRAINS

- LAND SIMULATION

  Body

  1. ALOGRITHMS
  2. BASES
  3. CLOCK
  4. FORCES

5. GROUND_COMPONENTS
6. GROUND_HEX
7. HEX
8. HISTORY
9. LAND_UNITS
10. OBSTACLES
11. PIPELINES
12. RAILROADS
13. ROADS
14. SIMULATION_CONSTANTS
15. SUPPLY_TRAINS

- LAND UNIT

  Specification

  1. FORCES
  2. HEX

  Body

  1. COMPOSITE_LINKED_LIST
  2. DOUBLY_LINKED_LIST
  3. GROUND_COMPONENTS
  4. SIMULATION_CONSTANTS

- OBSTACLES

  Body

  1. DOUBLY_LINKED_LIST
  2. SIMULATION_CONSTANTS

- PIPELINES

  Specification

  1. HEX

  Body

  1. DOUBLY_LINKED_LIST

- RAILROADS

  Specification

  1. HEX

  Body

  1. DOUBLY_LINKED_LIST

- ROADS

  Specification

  1. HEX

  Body

  1. DOUBLY_LINKED_LIST

- RUNWAYS

  Body

  1. DOUBLY_LINKED_LIST

- SABER

  Specification

  1. TBD

  Body

  1. TBD

- SATELLITES

  Specification

  1. FORCES
  2. HEX

  Body

  1. DOUBLY_LINKED_LIST

- SIMULATION_CONSTANTS

  NONE

- SUPPLY TRAINS

  Body

  1. DOUBLY_LINKED_LIST

2. COMPOSITE_LINKED_LIST

- UNIFORM_PACKAGE

  NONE

- WEAPONS

  Specification

  1. FORCES
  2. HEX
  3. WEATHER

  Body

  1. DOUBLY_LINKED_LIST

- WEAPONS LOAD

  Body

  1. DOUBLY_LINKED_LIST

- WEATHER

  Body

  1. SIMULATION_CONSTANTS
  2. UNIFORM_PACKAGE

Bibliography

1. Booch, Grady. *Software Components with Ada.* Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1987.
2. Booch, Grady. *Software Engineering with Ada* (second Edition). Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1987.
3. Booch, Grady. *Object-Oriented Design With Applications.* Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.
4. Coad, P. and E. Yourdon. *Object-Oriented Analysis.* New York, NY: Yourdon Press, 1990.
5. Coleman, D. and F. Hayes. "Lessons From Hewlett-Packard's Experience of Using Object-Oriented Technology," *Proceedings of TOOLS 4*, 327–333 (March 1991).
6. Freitas, Maria Manuel, et al. "Object-Oriented Requirements Analysis in an Ada Project," *Ada Letters*, *6*:97–109 (July, August 1990).
7. Horton, Andrew. *Design and Implementation of a Graphical User Interface and Database Management System for the Saber Wargame.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH USA, December 1991.
8. III, William F. Mann. *Saber : A Theater Level Wargame.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH USA, March 1991.
9. Inc., EVB Software Engineering. "An Object Oriented Design Handbook for Ada Software,". EVB Software Engineering, Inc, 1985.
10. Ince, Darrel. *Object-Oriented Software Engineering with C++.* Maidenhead, England: McGraw-Hill, Inc., 1991.
11. Jean, Cathrine and Alfred Strohmeier. "An Experience in Teaching OOD for ADA Software," *Software Engineering Notes*, *15*(9):44–49 (October 1990).
12. Klubunde, Gary Wayne. *An Animated Graphical Postprocessor for the Saber Wargame.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH USA, December 1991.
13. Korson, Tim and John D. McGregor. "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, *33*(9):40–60 (September 1990).
14. Maruyama, Robert and Meredith Stoehr. *Computer Science II with Ada* (Third edition Edition). University of Dayton, 1991.
15. Mathias, Karl, "Saber Simulation Environment Special Study - Findings and Recommendations," September 1992. Written for Special Studies Course at AFIT.
16. Moore, Donald R. *An Enhanced User Interface for the Saber Wargame.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH USA, December 1992.
17. Ness, Marlin Allen. *A New Land Battle for the Theater War Exercise.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH USA, June 1990.

18. Pressman, Roger S. *Software Engineering : A Practitioner's Approach.* New York, NY: McGraw-Hill, Inc., 1992.

19. Roberts, Stephen D. and Joe Heim. "A Perspective on Object-Oriented Simulation." *Proceedings of the 1988 Winter Simulation Conference.* 277–281. 1988.

20. Rumbaugh, James, et al. *Object-Oriented Modeling And Design.* Schenectady, NY: Prentice-Hall, Inc., 1991.

21. Scagliola, David L. *Saber : Airland Combat Training Model ; Credibility Assessment and Methodology.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH USA, March 1992.

22. Sherry, Christine M. *Object-Oriented Analysis and Design of the Saber Wargame.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH USA, December 1991.

23. Shlaer, S. and S. J. Mellor. *Object-Oriented Systems Analysis : Modeling the World in Data.* Prentice-Hall, 1988.

24. Shore, Dr. R. W. "Object-Oriented Techniques with Ada : An Example," *Ada Letters*, *9*:54–67 (September, October 1989).

25. Sommerville, Ian. *Software Engineering* (3 Edition). Wokingham, England: Addison-Wesley Publishing Company, Inc., 1989.

26. Winbald, Ann L., et al. *Object-Oriented Software.* Reading, MA: Addison-Wesley Publishing Company, Inc., 1990.

Vita

First Lieutenant David Scott Douglass was born on 15 May 1966 in Tokyo, Japan to Mr. and Mrs. David V. Douglass. He graduated from Mannheim American High School in Mannhiem, Germany in 1984. In December of 1988, he graduated with a Bachelor of Science in Electrical Engineering from Clemson University, South Carolina where he received Distinguished Graduate honors from the Clemson ROTC detachment. His first assignment was to the Air Force Logistics Center at Robins AFB, Georgia as an airborne avionics project engineer. He entered the School of Engineering at the Air Force Institute of Technology in May 1991.

Permanent address: 1411 Cimarrin Circle
Fairborn OH 45324

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE

Object-Oriented Analysis, Design, and Implementation of the Saber Wargame

6. AUTHOR(S)

David Scott Douglass, First Lieutenant, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GCE/ENG/92D-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AUCADRE/WG
Air Force Wargaming Center
Maxwell AFB, AL 36112-5532

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. ABSTRACT (Maximum 200 words)

Saber is a two-sided, air and land war game that simulates decisions made of commanders at the theater-level. It is being developed by the Air Force Institute of Technology for the Air Force Wargaming center at Maxwell AFB, Alabama. Saber models conventional, chemical, and nuclear warfare between aggregated air and land forces. It also portrays the effects of logistics, satellites, weather, terrain, and intelligence which add to the realism of the Saber war game.
The Saber war game has three main components, the preprocessor, which is responsible for scenario development and pre-game activities, the simulation, the guts of the war game that provides execution of missions and conflict resolution, and the postprocessor, which provides detailed reports and an animated graphical output of troop movement.
This thesis documents the object-oriented analysis, design and implementation of Saber simulation. During the analysis and design phase, a five step process was used. These steps included identifying the objects along with their attributes and operations and encapsulating them within Ada Packages. During implementation, sound object-oriented principles were used to ensure a system that could be easily understood, modified, and enhanced.

14. SUBJECT TERMS

Object-Oriented Design, Object-Oriented Programming, Simulation, Wargame

15. NUMBER OF PAGES

114

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

NSN 7540-01-280-5500