

AD-A241 838

2



NAVAL POSTGRADUATE SCHOOL
Monterey, California

DTIC
30 JUN 91



THESIS

A TACTICAL DATABASE
FOR THE
LOW COST COMBAT DIRECTION SYSTEM

by

Everton Guilhao de Paula

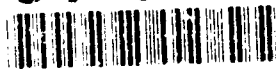
December 1990

Thesis Advisor:

Michael L. Nelson

Approved for public release; distribution is unlimited.

91-13930



91 10 24 098

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		4. PERFORMING ORGANIZATION REPORT NUMBER(S)	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NAVSEA	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N0002491WR01578	
8c. ADDRESS (City, State, and ZIP Code) Code 06D3131 Washington D.C. 20362-5101		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) A TACTICAL DATABASE FOR THE LOW COST COMBAT DIRECTION SYSTEM			
12. PERSONAL AUTHOR(S) De Paula, Everton G.			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM TO	14. DATE OF REPORT (Year, Month, Day) December, 1990	15. PAGE COUNT 233
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The purpose of this thesis is to design a Tactical Database for the Low Cost Combat Direction System. The resulting design, which is intended to serve as the basis for the implementation of the Tactical Database, is a set of classes (objects) organized in a hierarchy, according to the object-oriented programming paradigm. In Chapter I, basic concepts about Combat Systems and Object-Oriented Database Management Systems are introduced. Chapter II reviews the basic requirements for the Tactical Database. In Chapter III, an object-oriented design methodology is proposed. A step by step description of the design process applied to the Tactical Database is presented in Chapter IV. Clustering, concurrency control, crash recovery, garbage collection, and database security are discussed in Chapter V. Finally, in Chapter VI, additional considerations about the design approach and considerations for future improvements are presented. Readers already familiar with object-oriented programming and Combat Direction Systems may be able to skip Chapters I and III, without loss of continuity. For a quick reference, a summary of the design is provided in the Appendix.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL MAJ Nelson, Michael L.		22b. TELEPHONE (Include Area Code) (408) 646-2026	22c. OFFICE SYMBOL CS / NE

Approved for public release; distribution is unlimited.

A Tactical Database
for the
Low Cost Combat Direction System

by

Everton G. de Paula
Captain, Brazilian Air Force
B.S., Instituto Tecnológico de Aeronautica, Brazil, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the


NAVAL POSTGRADUATE SCHOOL
December 1990

Author:

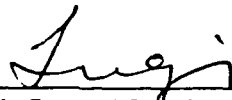


Everton G. de Paula

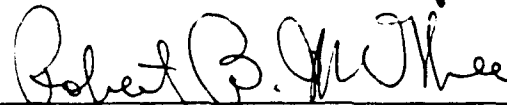
Approved by:



Michael L. Nelson, Thesis Advisor



Luqi, Second Reader



Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

The purpose of this thesis is to design a Tactical Database for the Low Cost Combat Direction System. The resulting design, which is intended to serve as the basis for the implementation of the Tactical Database, is a set of classes (objects) organized in a hierarchy, according to the object-oriented programming paradigm.

In Chapter I, basic concepts about Combat Systems and Object-Oriented Database Management Systems are introduced. Chapter II reviews the basic requirements for the Tactical Database. In Chapter III, an object-oriented design methodology is proposed. A step by step description of the design process applied to the Tactical Database is presented in Chapter IV. Clustering, concurrency control, crash recovery, garbage collection, and database security are discussed in Chapter V. Finally, in Chapter VI, additional considerations about the design approach and considerations for future improvements are presented.

Readers already familiar with object-oriented programming and Combat Direction Systems may be able to skip Chapters I and III without loss of continuity. For a quick reference, a summary of the design is provided in the Appendix.

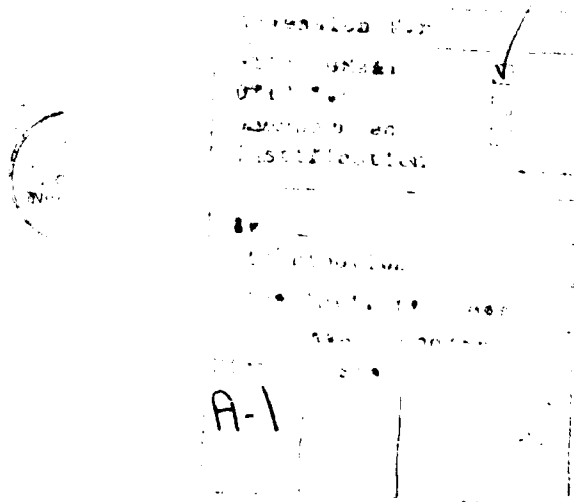


TABLE OF CONTENTS

I. INTRODUCTION	1
A. COMBAT SYSTEMS	1
1. Combat <i>Direction</i> Systems	2
2. Low Cost Combat <i>Direction</i> System	3
B. OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS	4
1. Object-Oriented Data Modeling	4
2. Why an Object-Oriented Database Management System	8
II. BASIC REQUIREMENTS	10
A. MINIMUM SET OF OBJECTS (CLASSES)	10
1. Terms and Definitions	10
2. The Objects	12
3. Class Hierarchies	18
B. ADDITIONAL REQUIREMENTS	19
1. Vocabulary	19
2. Object-Oriented Database Management System	19
3. Modularization and Concurrency	20
4. Performance	20
5. Display of Data	20

6. Range Scales	20
7. Constraints	20
8. Ownship	21
9. Closest Point of Approach	21
10. Waypoints	22
11. Tracks	22
12. Special Points	24
13. Tableaux	25
C. TACTICAL PLOT GRAPHIC TOOLS	27
1. Line	27
2. Circle	28
3. Range Circle	28
4. Expanding Circle	28
5. Moving Circle	28
6. Ellipse	28
7. Rectangle	29
8. Arc	29
III. DESIGN METHODOLOGY	30
A. OBJECT-ORIENTED TERMINOLOGY	30
B. OBJECT-ORIENTED DESIGN METHODOLOGIES	34
C. THE DESIGN PROCESS	36

1. Why Organize the Classes in a Hierarchy	36
D. THE METHODOLOGY	37
1. Identification of the Objects and Classes	38
2. Refinement of the Objects and Classes	42
3. Organization of the Classes in a Hierarchy	45
E. COMMENTS	51
IV. DESIGN OF THE TACTICAL DATABASE	53
A. IDENTIFICATION OF THE OBJECTS AND CLASSES	53
1. Initial Definition of the Objects and Classes	53
2. Analysis of the Object's Variables	63
3. Analysis of the Object's Methods	65
B. REFINEMENT OF THE OBJECTS AND CLASSES	66
1. Global Analysis of the Set of Objects and Classes	66
2. Individual Analysis of Objects and Classes	67
C. ORGANIZATION OF THE CLASSES IN A HIERARCHY	103
1. Analysis of the Implementation Language	103
2. Construction of the Hierarchies	104
3. Review of the Classes' Variables and Methods	111
D. DESCRIPTION OF THE METHODS DEFINED	135
1. Get Menu	136
2. Create Instance	136

3. Delete Instance	136
4. Update Instance	136
5. Display Instance	137
6. Get "Variable"	137
7. Set "Variable"	137
8. CPA Processing	137
9. Convert String to Numeric	138
10. Convert Numeric to String	138
11. Create New Class	138
12. Create New Variable	138
13. Create New Method	139
14. Dead Reckoning	139
15. Plot TDB Point	139
16. Draw TDB Line	139
17. Draw TDB Area	140
18. Calculate Geographical Position	140
19. Track Course and Speed Determination	140
20. New Track Establishment	140
21. Initial Category Assignment	141
22. Class Change Processing	141
23. Track Identification	141
24. Track History Processing	142

25. Monitor Ownship Position	142
26. Bearing and Range from Position to Position	142
27. Calculate Relative Position	142
28. Ownship Distance to Position	142
29. Estimated Time Enroute to Position	143
30. Estimated Time of Arrival at Position	143
31. Route Geometry	143
V. ADDITIONAL DESIGN CONSIDERATIONS	144
A. PHYSICAL STORAGE MANAGEMENT	144
1. Guidelines for Object Clustering	144
2. Clustering in the Tactical Database	148
B. CONCURRENCY CONTROL	152
1. Concurrency Management Mechanisms	154
2. Optimistic versus Pessimistic Concurrency Control	156
3. Concurrency Control in OODBMS	156
4. Concurrency Control in the Tactical Database	159
C. CRASH RECOVERY	160
1. Recovery in OODBMSs	161
2. Recovery in the Tactical Database	162
D. GARBAGE COLLECTION	162
1. Garbage Collection in OODBMSs	163

2. Garbage Collection in the Tactical Database	163
E. DATABASE SECURITY	163
1. Security in OODBMSs	165
2. Security in the Tactical Database	168
VI. COMMENTS AND CONCLUSIONS	169
A. DESIGN APPROACH	169
1. Data-Driven Design	169
2. Responsibility-Driven Design	170
3. Data-Driven versus Responsibility-Driven Design	171
4. Tactical Database Design Approach	173
B. TACTICAL DATABASE ISSUES	174
1. Object Space	174
2. Performance	174
3. Display Doctrine	176
4. Menus	176
C. FUTURE IMPROVEMENTS	177
1. Multiple Users	177
2. Automatic Multiple CPA Function	178
D. FINAL COMMENTS	178
1. Database Security and Data Integrity	179
2. Performance	179

E. CONCLUSION	180
APPENDIX	182
A. TACTICAL DATABASE CLASS HIERARCHY	182
B. CLASS REPRESENTATION FORMAT	183
C. CLASSES DEFINED FOR THE TACTICAL DATABASE	184
D. METHODS DEFINED FOR THE TACTICAL DATABASE	204
1. Get Menu	204
2. Create Instance	204
3. Delete Instance	204
4. Update Instance	205
5. Display Instance	205
6. Get "Variable"	205
7. Set "Variable"	205
8. CPA Processing	206
9. Convert String to Numeric	206
10. Convert Numeric to String	206
11. Create New Class	206
12. Create New Variable	207
13. Create New Method	207
14. Dead Reckoning	207
15. Plot TDB Point	207

16. Draw TDB Line	208
17. Draw TDB Area	208
18. Calculate Geographical Position	208
19. Track Course and Speed Determination	208
20. New Track Establishment	209
21. Initial Category Assignment	209
22. Class Change Processing	209
23. Track Identification	210
24. Track History Processing	210
25. Monitor Ownship Position	210
26. Bearing and Range from Position to Position	210
27. Calculate Relative Position	210
28. Ownship Distance to Position	211
29. Estimated Time Enroute to Position	211
30. Estimated Time of Arrival at Position	211
31. Route Geometry	211
 LIST OF REFERENCES	 212
 INITIAL DISTRIBUTION LIST	 215

LIST OF FIGURES

Figure 1: The Class Definition	32
Figure 2: The Extended Class Definition	33
Figure 3: TACTICAL DATABASE Initial Class Hierarchy	105
Figure 4: TRACK Class Hierarchy	107
Figure 5: SPECIAL POINT Class Hierarchy	108
Figure 6: TDB POINT Class Hierarchy	109
Figure 7: TACTICAL DATABASE Class Hierarchy	110
Figure 8: The Hierarchy of Classes Defined for the TACTICAL DATABASE .	182
Figure 9: The Class Definition Format	183

TABLE OF ABBREVIATIONS

CAD	Computer-Aided Design
CDS	Combat Direction System
CLF	Combat Logistic Force
CPA	Closest Point of Approach
DBMS	Database Management System
DLRP	Data Link Reference Point
DR	Dead Reckoning
ETA	Estimated Time of Arrival
ETE	Estimated Time Enroute
GMT	Greenwich Mean Time
ID	Identity
LCCDS	Low Cost Combat Direction System
LCCDSWS	Low Cost Combat Direction Software System
MMI	Man-Machine Interface
NAVSTA	Naval Sea Systems Command
NTDS	Navy Tactical Data System
OO	Object-Oriented
OODBMS	Object-Oriented Database Management System
OOP	Object-Oriented Programming

PIM	Position and Intended Movement	
SATNAV	Satellite Navigation	.
TacPlot	Tactical Plot	.
TDB	Tactical Database	-

ACKNOWLEDGEMENTS

I would like to express my thanks to Professor Luqi for her encouragement and trust and to MAJ Nelson for his valuable advice, ideas, and critical comments while I was working on this thesis.

I. INTRODUCTION

The Tactical Database is an important integral part of the Low Cost Combat Direction Software System (LCCDSWS). It will contain information necessary to establish an accurate picture of a ship's environment. For this purpose, the Naval Sea Systems Command (NAVSEA) requires the design/development or selection of an object-oriented database management system. [Ref. 1]

The objective of this thesis is to identify, define, and organize in a class structure the objects that will compose the Tactical Database, using an object-oriented approach. The minimum set of objects required may be found in [Ref. 1:pp. 78-87].

For a clear understanding of the purpose and characteristics of the Tactical Database, basic concepts about Combat Systems, including Combat Direction Systems and the Low Cost Combat Direction System, and Object-Oriented Database Management Systems, are presented in this chapter.

A. COMBAT SYSTEMS

A ship's combat system is typically considered to be the set of men and machines which give the ship its fighting capabilities. This includes both the weapons and the means by which weapons are controlled. However, subsystems such as navigation and communications, which are not directly involved in weapons control, also play an important part in today's combat systems. The combat system can be seen as the payload

carried by a Navy warship. It is a composition of shipboard elements and personnel that perform the functions of detection, tracking, identification, processing, evaluation and control of engagement of hostile threats, either actively or passively. In terms of equipment, the combat system includes subsystems containing missiles, guns, sensors, electronic warfare, navigation and communications. [Ref. 1]

1. Combat Direction Systems

A Combat Direction System (CDS) is defined as follows [Ref. 2:pp. 3-5]:

Those combinations of data and men handling systems, either manual or automated, employed to execute the combat direction functions. Levels of command, from the platform (ships, aircraft, submarines) up to, and including, task group/force are supported.

A CDS consists of a complex set of data inputs, user consoles, converters, adapters, and radio terminals interconnected with high speed general purpose computers and their stored programs. Combat data are collected, processed and composed into a picture of the overall tactical situation that enables the force commander to make rapid and accurate decisions.

The high-level goal for the CDS can be stated as follows:

Optimize the individual ship's fighting capability in concert with the other segments of the combat system [Ref. 2:pp. 3-7].

In general, the CDS role is composed of the following [Ref. 2:pp. 3-7]:

- The automated shipboard database manager of tactically significant tracks.
- The primary combat system element supporting the Combat Direction Center.
- The multiwarfare element of the combat system.

2. Low Cost Combat Direction System

The Low Cost Combat Direction System (LCCDS) project, sponsored by the Naval Sea Systems Command (NAVSEA), will implement the basic features of a Combat Direction System on a commercially available microprocessor based workstation. The system will be used on board non-combatant ships, where no automated combat information processing capability currently exists. [Ref. 1]

According to [Ref. 1], the LCCDS may be defined as:

The LCCDS shall be the integrating part between ownship and other units. It shall consist of hardware, software, and personnel to provide the ship's command personnel with a facility for monitoring the overall air, surface, and subsurface environment. Force and ownship sensor data shall be collected, correlated and evaluated by the LCCDS.

The increasing availability of small, powerful and relatively inexpensive microprocessor workstations, ruggedized for military applications, makes some form of Combat Direction System feasible for any Navy vessel. The U.S. Navy Combat Logistics Force (CLF) ships and other fleet support vessels currently handle all tactical navigation, contact management and intelligence tasks manually using only a maneuvering board and grease pencil. The LCCDS will be capable of filling this information processing void, at a fraction of the cost of a full tactical CDS. [Ref. 1]

The LCCDS, by means of its software system, will interact with the user, the Tactical Database, the Navigation System, the Radar System, the Link 11 System, and the Shoreline Database [Ref.1:p. 22].

The **Navigation System** is a device that provides the system with ownship information in terms of present geographical position, course, speed, and time. The **Radar System** is a device that provides the system with ownship contact information in terms of bearing and range. The **Link 11 System** is a device that provides the system with external information on tracks in terms of geographical position, course, speed, time, track number, and track identity. The **Shoreline Database** is a device that provides the system with external information on shorelines. The **user** is any crewmember designated to operate the LCCDS for the purpose of providing the necessary information for quick, accurate tactical decision-making. [Ref. 1]

B. OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

In recent years, a new trend has been emerging in data modeling and database processing. In this approach a database is considered as a collection of **objects**, where each object represents a physical entity, a concept, an idea, an event, or some aspect of interest to the application. In the classical **record-oriented** data models (relational, network, and hierarchical) data is viewed as a collection of record types (or relations), each having a collection of records (or tuples) stored in a file. In an **object-oriented** system, real-world objects are represented directly by database objects. [Ref. 3]

1. Object-Oriented Data Modeling

Consider, for example, the real-world object **EMPLOYEE**; using a record-oriented approach (in this case, a relational schema), the information about an employee could be scattered over the following relations [Ref. 3]:

- EMPLOYEE contains most of an employee's attributes.
- DEPARTMENT contains information on an employee's department.
- WORKS_ON contains information on the projects the employee works on.
- DEPENDENT contains information on an employee's dependents.

This schema is quite adequate to store the information about the object EMPLOYEE.

However, as more complex types of real-world objects are modeled in emerging database applications, the traditional data models would scatter information on a complex object into more and more relations or record types, leading to the loss of a direct correspondence between a real-world object and its database representation. One goal of object-oriented modeling is to maintain a direct correspondence between real-world objects and database objects so that the objects do not lose their integrity and identity and can easily be identified and operated upon. [Ref. 3]

However, the object-oriented approach is still being refined and, so far, a comprehensive model does not exist.

The term **object-oriented** (OO) is used in several disciplines, including programming languages, databases and knowledge bases, artificial intelligence, and computer systems in general. OO programming languages have their roots in the SIMULA language, which proposed the concept of the **class** or abstract data type that encapsulates structural properties of objects as well as specifying valid operations on types of objects. The main difference between OO programming languages and databases is that the latter allow the existence of **persistent objects** stored permanently in secondary

storage, whereas the objects in an OO programming language last only during program execution. [Ref. 3]

There has been a large number of proposals for OO data models and database systems. The following are the essential features of these proposals [Ref. 3]:

- **Data abstraction and encapsulation:** This refers to the ability to define a set of operations, called **methods** in OO terminology, that can be applied to the objects of a particular object class (or object type). All access to objects is constrained to be via one of these methods. An object has an **interface** and an **implementation**; the implementation is private and may be changed without affecting the interface. All access to an object is via its interface, which is public, i.e., can be seen by other objects and by the users of the system. Object classes may be organized class hierarchies.
- **Object identity:** The OO system maintains a unique identity for each object that is independent of its attribute values. This is typically represented by an **object identifier**, which is generated by the system. Hence any attribute of an object can be modified without affecting its identity.
- **Inheritance:** A subclass inherits both data attributes and methods from its superclass in the class hierarchy. Selective inheritance, where only some attributes and methods are inherited, is also possible.
- **Multiple Inheritance:** Some systems support multiple inheritance, in which a subclass is allowed to have several superclasses.
- **Complex objects:** The ability to define new composite objects from previously defined objects in a nested or hierarchical fashion.

In addition to the above features, the implementation of OO data models typically exhibits the following [Ref. 3]:

- **Message passing:** Objects communicate and perform all operations, including retrieval of values, computations, and updates, via messages. A message consists of an object (or several objects) followed by the name of a method to be applied to these objects.

- **Operator overloading:** This describes the convenient feature of being able to use the same operation name to denote different operations applied to different types of objects. The meaning of the operation is then "overloaded" and can be resolved only when the object it is to be applied to is provided. This overloading occurs in OO systems when methods for distinct object types are allowed to have the same name. It is related to the concept of late binding, where a message first determines the type of object to be applied to before being bound to the appropriate method. Operator overloading is also called polymorphism.

The following are advantages of the OO approach [Ref. 3]:

- **Extensibility:** Object types and their methods can be modified as needed. Such changes are localized to the relevant object type and hence are much easier than in record-based systems, where many record types may be affected. In addition, new object classes and their methods can be incorporated into the system.
- **Behavioral constraints:** Because of encapsulation, the behavior of each object type is predetermined by a fixed set of methods. Hence, database operations are constrained to be within these behavioral specifications.
- **Flexibility of type definition:** The user is not limited to the modeling concepts of a specific data model but can define many variations of data types, each with unique properties.
- **Modeling power:** The inheritance of both attributes and methods is a very powerful tool for data modeling. In general, the abstractions of generalization or specification, identification, and aggregation are well supported in current OO data models.

The following are disadvantages of the OO approach [Ref. 3]:

- **Lack of associations:** The abstraction of association (represented by relationship types in the Extended Entity-Relationship model) is not directly supported and is achieved indirectly by allowing interobject references. This is an inherent weakness of the OO approach and is due to the fact that this approach treats each object as a self-contained unit of information.
- **Behavioral rigidity:** The notion of predetermining and prespecifying all operations by a fixed set of methods is a rigid constraint that is counter to the typically evolving nature of database technology.

- **No high-level query language:** There are no high-level query languages for current OO data models. So far, the power and elegance of relational algebra or calculus languages do not have counterparts in OO systems.

2. Why an Object-Oriented Database Management System

Relational databases have been used extensively for commercial applications, in which information has traditionally been viewed as flat collections of records, but pure relational databases do not appear adequate for representing highly structured data models such as those found in engineering applications, either conceptually or as efficient direct implementations. [Ref. 4]

Relational databases offer a uniform, elegant model which is well understood mathematically, but they have only one level of structure and are therefore not well suited to represent complex objects. The simplicity of the relational database model is a drawback in dealing with highly structured problems, because the semantics of a problem cannot be totally represented in flat relations among attributes; the semantics must reside, therefore, in the code. [Ref. 4]

Relational tables are not organized into any superstructure, unlike classes in an object-oriented model. A relational table contains only attribute values from a predefined set of primitive types, such as integer, real, character string (often fixed length), and date; abstract data types are not generally supported. [Ref. 4]

There is no way to create a unique object, except by specifying unique values for certain fields, since there is no concept of object identity. In practice, the database implementor must often generate arbitrary IDs for objects and include ID fields in relational tables to ensure uniqueness. If IDs are not generated, then relations among

objects must be represented by relational tables in which each object is represented by a set of unique attributes. Taken with the lack of abstract data types as field values, this implies that the program must manipulate composite keys as if they were single entities.

[Ref. 4]

Object-oriented systems, on the other hand, can represent complex data structures and some of their semantics directly in the data model. [Ref. 4]

A large class of applications exists for which relational database systems and other more conventionally-structured database systems are too limited. As previously mentioned, these applications can be characterized as complex, large-scale, data-intensive programs. One example is a computer-aided design (CAD) system in which the designer creates a hierarchy of inter-related design objects. This class of applications needs a database model that is more expressive and flexible than the relational model. Furthermore, this class of applications needs database technology that is designed to facilitate programming-in-the-large, i.e., designing, constructing, and maintaining large, complex programs. Object-oriented databases are being developed to meet the data handling needs of such applications. [Ref. 5]

In the Low Cost Combat Direction System, the data that is to be stored in the Tactical Database is complex, interrelated, and has to be manipulated by a set of diverse operations. A conventional (relational, hierarchical, or network) database is inadequate to efficiently store and manipulate such kind of data. For this reason, an object-oriented database management system is a necessity.

II. BASIC REQUIREMENTS

The Tactical Database (TDB) will contain information necessary to establish an accurate picture of a ship's environment. The unit of information within the TDB will be the **object**. Information necessary to generate an object will be provided by the user, an external system (Radar, Link 11), or an internal module. [Ref. 1]

A minimum set of objects (classes) was identified in [Ref. 1:pp. 78-87]. This set will be reviewed in this chapter, along with the addition of a brief description of the role played by each object in the context of the TDB.

The requirements for the LCCDS are stated in [Ref. 1]. Among these are the basic requirements for the TDB. However, they are scattered throughout the whole document. Those requirements for the TDB that are relevant to the design of the objects and their class structure are also reviewed in this chapter.

A. MINIMUM SET OF OBJECTS (CLASSES)

1. Terms and Definitions

Before describing the minimum set of objects, it is necessary to understand the terms that appear in the definition of the objects:

- **Geographical Position** is a position relative to the earth's surface expressed in terms of latitude and longitude.

- **Relative Position** is a position expressed in terms of range and bearing from an arbitrary point.
- **Course** is the angle between true north and the direction in which a ship, submarine, or aircraft proceeds.
- **True Bearing** is the angle between true north and an imaginary line between two objects.
- **Relative Bearing** is the angle between the bow of Ownship and an imaginary line between Ownship and another object.
- **Track** is a representation of some environmental phenomena converted into accurate estimates of geographical position with respect to time, course, speed, and depth and height when applicable.
- **Vehicular Track** is any object with attributes course and speed.
- **Track Number** is a positive integer uniquely related to each track.
- **Track Identity** classifies each track as one of the following: Friend, Hostile, Unknown, and Special.
- **Track Category** classifies each track as one of the following: Tentative, Air, Surface, Subsurface, and Special.
- **Origin** describes the source of information of a track/special point object. The origin can be Local Manual, Local Auto, or Remote.
- **Sensor** is any device which somehow scans the external environment surrounding the LCCDS and detects a certain class of tactically significant phenomena.
- **Local Manual** denotes an object who's source is user input.
- **Local Auto** denotes an object who's source is an ownship sensor.
- **Remote** denotes an object who's source is external, in particular Link 11.
- **Special Point** is an object, other than a vehicular track, which supports the user in establishing a complete picture of the tactical environment. A special point may be fixed to a geographic point, or move with any valid course and speed.
- **Data Link Reference Point** is a fixed geographic position representing the origin of a cartesian coordinate system in which track positions are reported.

- **Closest Point of Approach (CPA)** is the position of a vehicle in range, true or relative bearing and time to another vehicle, when the two vehicles will be closest to each other.
- **Dead Reckoning** is the estimation of a ship's position based on its course and speed and not from observation.
- **Tactical Plot (TacPlot)** is a graphical representation of the spatial relationship between ownship and tactically significant tracks. A TacPlot is a window for graphical input and output.
- **Hook** is a function that identifies a position on the TacPlot selected for some program action.
- **Mark** is a function that fixes a previously drawn graphic object to its current position on the TacPlot.
- **Save** is a function that operates on user generated graphic objects which exist on the TacPlot allowing them to be saved in the Tactical Database as user defined instantiations of the class USERDEFINED.
- **Ball Tab** is a moving cursor represented by a one-quarter inch diameter circle with a point in the center. The Ball Tab provides direct feedback to the operator as he manipulates the trackball (or mouse, if employed). Movement of the Ball Tab within the TacPlot matches the relative speed and direction of any trackball movement.
- **Julian Date** is simply the last digit of the current year followed by the day represented as a three digit integer which corresponds to a sequential count beginning on 1 January.
- **GMT** stands for Greenwich Mean Time.

2. The Objects

Class: OWNSHIP

Attributes: Geographical Position (latitude:longitude)
 Time of Position (hh:mm:ss)
 Course (degrees)
 Speed (knots)
 Origin (Local Manual)

Tracknumber {positive integer 0..9999}
Type {string of ten characters}
Track History {boolean}

Operations: Monitor Ownship Position
Navigational Computation
CPA Processing

Description: An object of class OWNSHIP represents the ship that is operating the Low Cost Combat Direction System. It therefore follows that there should be only a single instance of OWNSHIP for each Tactical Database.

Class: TENTATIVE TRACK

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}
Time of Position {hh:mm:ss}
Course {degrees}
Speed {knots}
Category {Tentative}
Identity {Unknown}
Origin {Local Manual, Local Auto}
Tracknumber {positive integer 0..9999}
Type {string of ten characters}

Operations: New Track Establishment
Track Position Data
Track Course And Speed Determination
Dead Reckoning
Initial Category Assignment
Initial Identity Assignment
Track Termination

Description: An object of class TENTATIVE TRACK represents a new track generated by local sensors (radar). Once a valid course and speed is established for it, the Tentative Track will become a Firm Track. This can happen as a result of one of the following conditions: after three manual position updates by the user, when the category is entered manually before three position updates, or when manually declared firm by the user.

Class: AIR TRACK

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}
Time of Position {hh:mm:ss}
Course {degrees}
Speed {knots}
Height {feet}
Category {Air}
Identity {Unknown, Friendly, Hostile}
Origin {Local Manual, Local Auto, Remote}
Tracknumber {positive integer 0..9999}
Type {string of ten characters}

Operations: CPA Processing
Track Position Update
Track Course And Speed Determination
Dead Reckoning
Track Position Prediction
Track History Processing
Manual Termination
Identification Function

Description: An object of class AIR TRACK represents a real-world object which is in the air and, consequently, has Height as one of its attributes to indicate its current altitude.

Class: SURFACE TRACK

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}
Time of Position {hh:mm:ss}
Course {degrees}
Speed {knots}
Category {Surface}
Identity {Unknown, Friendly, Hostile}
Origin {Local Manual, Local Auto, Remote}
Tracknumber {positive integer 0..9999}
Type {string of ten characters}

Operations: The same operations that apply to the class AIR TRACK.

Description: An object of class SURFACE TRACK represents a real-world object which is on the surface.

Class: SUBSURFACE TRACK

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}
Time of Position {hh:mm:ss}
Course {degrees}
Speed {knots}
Depth {feet}
Category {Subsurface}
Identity {Unknown, Friendly, Hostile}
Origin {Local Manual, Local Auto, Remote}
Tracknumber {positive integer 0..9999}
Type {string of ten characters}

Operations: The same operations that apply to the class AIR TRACK.

Description: An object of the class SUBSURFACE TRACK represents a real-world object which is underwater and, consequently, has Depth as one of its attributes.

Class: REFERENCE POINT

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}
Time of Position {hh:mm:ss}
Category {Special}
Identity {Refpoint}
Origin {Local Manual}

Operations: Enter/Update Reference Point
CPA Processing

Description: An object of class REFERENCE POINT represents a fixed point on the surface of the Earth which is used as a reference.

Class: NAVIGATION HAZARD

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}

Time of Position {hh:mm:ss}
Category {Special}
Identity {Navhaz}
Origin {Local Manual}

Operations: Enter/Update Navigation Hazard
CPA Processing

Description: An object of class NAVIGATION HAZARD represents a point on the surface which might be hazardous to navigation (icebergs, shallow waters, reefs, mines, etc.) and, therefore must be avoided.

Class: MAN IN WATER

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}
Time of Position {hh:mm:ss}
Category {Special}
Identity {Man In Water}
Origin {Local Manual}

Operations: Enter/Update Man in Water
CPA Processing

Description: An object of class MAN IN WATER represents a point on the surface of the water on which a man (or a group of men) is supposed to be.

Class: WAYPOINT

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}
Time of Position {hh:mm:ss}
Steaming Route {integer 1..6}
Category {Special}
Identity {Waypoint}
Origin {Local Manual}

Operations: Waypoint Geometry

Description: An object of class WAYPOINT represents a destination point on the surface. Each waypoint can be viewed as a "node" in a determined route.

Class: DATA LINK REFERENCE POINT

Attributes: Geographical Position {latitude:longitude}
Time of Position {hh:mm:ss}
Category {Special}
Identity {Dlrp}
Origin {Local Manual}

Operations: Enter/Update Data Link Reference Point
CPA Processing

Description: An object of class DATA LINK REFERENCE POINT represents a fixed geographic reference position common to all Link 11 participating units.

Class: FORMATION CENTER

Attributes: Geographical Position {latitude:longitude}
Time of Position {bearing:range}
Course {degrees}
Speed {knots}
Category {Special}
Identity {Fc}
Origin {Local Manual}

Operations: Enter/Update Formation Center
Dead Reckoning
CPA Processing

Description: An object of class FORMATION CENTER represents a moving geographic position representing the center of a group of ships steaming in formation.

Class: POSITION AND INTENDED MOVEMENT

Attributes: Geographical Position {latitude:longitude}
Relative Position {bearing:range}
Time of Position {hh:mm:ss}
Course {degrees}
Speed {knots}
Category {Special}
Identity {Pim}
Origin {Local Manual}

Operations: Enter/Update Position And Intended Movement
Dead Reckoning
CPA Processing

Description: An object of class POSITION AND INTENDED MOVEMENT represents the Ownship or formation planned position based on a pre-computed base course and speed to arrive at a destination at the required time.

Class: USERDEFINED

Attributes: A minimal set is:
Geographical Position (latitude:longitude)
Time of Position (hh:mm:ss)
C {character/string}
N {numeric}
F {floating value}
D {date}
T {time}
L {logical}
DD {condition}

Operations: Not specified.

Description: The Tactical Database shall allow the user to define new classes of objects at run-time. The definition of a new class shall allow the user to instantiate objects of that class. A new class shall be defined either by selection from a given set of attributes and operations or by definition of new attributes or operations. [Ref. 1]

3. Class Hierarchies

Due to several similarities in the attribute sets of the classes Air Track, Surface Track, and Subsurface Track it would probably be appropriate to make these classes subclasses of a class TRACK. [Ref. 1]

For the same reason, the classes Reference Point, Navigation Hazard, Man In Water, Waypoint, Data Link Reference Point, Formation Center, and Position And Intended Movement could be made subclasses of a class SPECIAL POINT. [Ref. 1]

B. ADDITIONAL REQUIREMENTS

This section contains additional requirements, also extracted from [Ref. 1], that are relevant to the design of the objects and their class structure:

1. Vocabulary

- The LCCDSWS shall be defined using the vocabulary of existing CDSs and their environment.
- All LCCDS terms, acronyms, and data naming conventions shall be limited to those commonly used and meaningful within the Navy Tactical Data Systems (NTDS) environment.

2. Object-Oriented Database Management System

- The Tactical Database shall be an object-oriented database management system (OODBMS). It shall be an integral part of the LCCDSWS.
- The OODBMS shall provide a set of object types.
- The OODBMS shall allow the user to define new object types at run-time.
- The OODBMS shall provide features for data input and output.
- The OODBMS shall provide a user friendly query language for building logical and arithmetic relationships between database objects.
- Use of commercial window and database management software shall be explored in the development of the LCCDS prototype.
- The Tactical Database shall be object-oriented, allowing all associated track data (text, graphics, functions) to be stored as a single entity. The database shall be designed to utilize dynamically allocated storage. The maximum number of stored tracks shall be limited only by physical memory availability.
- The OODBMS shall provide system alerts when an invalid operation on an object is issued, when an invalid condition for an object is specified, and when a relationship between objects cannot be established.

3. Modularization and Concurrency

- The system shall be modularized to allow an implementation in incremental steps.
- The system shall be highly concurrent and amenable to future extensions.

4. Performance

- System response shall be sufficiently fast, and predictable under varying system loading to support critical decision-making reliably in a highly stressful, rapidly changing tactical environment.
- Response times for menu selections, track information requests, tactical display aids (e.g., graphical tools) shall be less than one second.

5. Display of Data

- The system shall provide, via a display doctrine, the capability for user defined conditional statements, in IF-THEN form, which control data filtering and specify presentation parameters for displayed data.

6. Range Scales

- The system shall provide for selection of range scales.
- The Range Scale shall be an integer and a member of the following set of values: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

7. Constraints

- All range information shall be expressed in nautical miles and/or yards.
- All azimuth (bearing/heading) information shall be expressed in degrees (000.00-359.99) relative to either true or magnetic north as indicated by an uppercase T or M immediately following the second decimal place of the number.

- Altitude and depth information shall be expressed in feet.
- Time should be expressed in 24-hour clock digital format followed by the appropriate upper case letter time zone identifier. The default time zone shall be Greenwich Mean Time (GMT). The user shall have the option of displaying current time in any time zone.
- Geographical position shall be expressed in terms of latitude (north-south) and longitude (east-west) degrees, minutes, and seconds.
- Speed shall be expressed in knots.

8. Ownship

- The system shall provide the operator with the capability to enter the following navigation inputs: Julian Date, GMT, Ownship heading, Ownship speed, Ownship latitude and longitude, true or relative bearing and range to a Special Point.
- The system shall update ownship position and velocity at a minimum of once every four seconds based on the positional data received from either manual or automatic input.
- The system shall provide the capability to reposition ownship relative to a geographically fixed Special Point resulting in adjustments to ownship latitude and longitude.
- The system shall provide automatic dead reckoning calculations for latitude and longitude position based on the last position, course, and speed.

9. Closest Point of Approach

- The system has to provide Closest Point of Approach (CPA) geometry from ownship to any track and between any two tracks upon user request.
- When the CPA of a specified track meets specified time and range criteria, the user shall be notified of a Close CPA or Collision CPA as appropriate. The system shall allow the user to manually adjust the criteria for Close CPA and Collision CPA calculations and alert as follows:

- Close CPA Range: 200 - 9,999 yards.
 - Close CPA Time: 5 - 99 minutes.
 - Collision CPA Range: 1 - 199 yards.
 - Collision CPA Time: 5 - 99 minutes.
- Preset parameters for CPA calculations shall be requested from the user upon system initialization.

10. Waypoints

- The system has to allow for the specification of up to six steaming routes with up to 50 waypoints (destinations) per route.
- The system shall determine the course, at a specified speed, that a vehicular track or the ownship must take in order to intercept a designated waypoint.
- The system shall update the maneuvering geometries every four seconds.
- Waypoint maneuvering data shall be presented with course, speed, bearing, and distance to the waypoint and shall include the GMT of the estimated arrival time at the waypoint, the waypoint designation and Time-To-Go to the waypoint.

11. Tracks

- The user should be allowed to assign additional information to any type of track.
- All tracks shall be updated at least every four seconds.
- The system shall support up to 1000 active tracks/special points displayed on the TacPlot. The number of tracks/special points held in the Tactical Database shall be constrained only by physical memory limits of the system.
- The user shall be allowed to display all Link 11 remote tracks.
- The system shall automatically assign track numbers to all locally generated tracks based upon a user designated set of integers. Valid track numbers must be four

digit integers in the range of 0200-7776 to ensure compatibility with Link 11 remote tracks.

- The system shall provide the capability to enter new tracks manually.
- A newly entered vehicular track shall be initiated as a tentative track.
- The system shall determine when local track entries are sufficiently reliable to warrant a firm track status. Firm track status shall be based on the number of updates. Firm/tentative status shall only apply to vehicular tracks.
- The tentative tracks shall become firm tracks as a result of one of the following conditions: after three manual position updates, when the category is manually entered before three position updates, or when manually declared firm.
- The system shall automatically compute the course and speed for a newly entered track or any track on which a position update is taken.
- Course and speed shall be accepted for all vehicular tracks with bearing values in degrees and speed values up to 9999 knots.
- The system shall automatically perform Dead Reckoning computations for all vehicular tracks via an extrapolation of the last known position, course, and speed. If no Track Position Update is received for a vehicular track within four seconds, the track shall be automatically repositioned. This shall not cause automatic course and speed computation.
- The system shall compute predicted track positions on user selected vehicular tracks via an extrapolation of the last known position, course, and speed. The prediction rate shall be four seconds.
- The system shall maintain history positional data on all air, surface, and subsurface vehicular tracks. The points shall be displayed upon request for an individual track.
- The system shall provide for the manual and automatic termination of all local tracks by responding to Drop Track notifications by deleting the specified track from the track stores, erasing its symbol from the screen, and notifying the user.
- The system shall provide for automatic termination of a tentative track if no update is received within six minutes for an air track or 12 minutes for a surface track. Track termination shall be performed on user confirmation only.

- The system shall provide the capability to manually terminate a specified track. Tracks that have other tracks slaved to them shall not be dropped until all slaved tracks have been dropped.
- The system shall perform vehicular track category determination. The categories to be assigned are Air, Surface, and Subsurface.
- The system shall assign a category to all new vehicular tracks on initial entry. The initial category assignment shall be based on category speed as follows:
 - speed less than category speed: Surface.
 - speed greater than category speed: Air.
- The category speed shall be modifiable; the preset value shall be 50 knots.
- The system shall perform a vehicular track identity determination. Track identities are Unknown, Friendly, and Hostile.
- The system shall assign a vehicular track the identity Unknown upon new track establishment.
- The user shall be capable of changing the identity of all vehicular tracks.

12. Special Points

- Special points that shall not be dropped once entered are Ownship and Data Link Reference Point (DLRP).
- The system shall allow the displaying of up to 20 Reference Points symbols at any hooked position on the TacPlot.
- The system shall provide the user with the capability to enter/update a Data Link Reference Point. Entries shall include latitude and longitude of the DLRP. Once entered, a DLRP cannot be terminated.
- The system shall allow the displaying of up to 50 Navigation Hazard symbols at once.

- The system shall allow the displaying of up to seven Man In Water symbols on the TacPlot at once.
- The system shall provide the capability to enter and reposition up to ten Man-In-Water special points. The position of Man-In-Water shall be geographically fixed.
- There may be up to two Formation Center symbols active in the system at one time.
- The Position And Intended Movement (PIM) function associated with Ownship and/or Formation Center shall provide the following navigational information based on destination time and place and associated base course and speed:
 - display the PIM symbol at the planned position along the steaming route based on current or user selected time.
 - PIM shall maintain a running calculation of distance and time ahead/behind plan based on current geographical position.
 - provide recommended course and speed for Ownship in order to regain PIM position.

13. Tableaux

- The LCCDS shall provide detailed information on all aspects of the tactical situation and system control, operating parameters, and status. This information shall be indexed for easy user access and presented in tableau form using Alphanumeric I/O windows.
- The Navigation tableau shall provide detailed information on ownship navigation including, but not limited to, the following:
 - time (standard GMT and local format - hh:mm:ss)
 - lat/long position (dd:mm:ss)
 - course (relative to true and magnetic north - 000.0 T/M)
 - speed (000.0 knots)
 - magnetic variation (degrees East/West - 00.0 E/W)

- set (degrees true - 000.0 T)
 - drift (speed in knots - 000.0)
 - wind direction (degrees true - 000.0 T)
 - wind speed (speed in knots - 000.0)
 - distance to next waypoint (nautical miles great circle or rhumb (direct distance))
 - ETA (Estimated Time of Arrival) at next waypoint
 - ETE (Estimated Time Enroute) to next waypoint (hh:mm:ss)
 - navigation source (inertial, SATNAV, Omega, Dead Reckoning (DR), etc.)
- The Float Plan tableau provides the user a tool for defining up to 50 steaming routes of up to 50 waypoints each. A steaming route shall consist of a numbered list (in order of entry) of waypoints. A waypoint is defined by latitude (dd:mm:ss N/S) and longitude (dd:mm:ss E/W).
 - The Active Tracks tableau shall provide a list of all tracks currently held in the Tactical Database along with time of last update.
 - The system shall provide the user with the capability to define both static and dynamic tableaux.
 - The differentiation between static and dynamic tableaux is the degree of complexity involved. Static tableaux contain information provided by the user and is not maintained, changed, or updated by program action. Procedures and checklists are examples of the type of information available through static tableaux. Dynamic tableaux are more complicated to define because it is intended that the program will somehow act upon or effect their contents.
 - The main source of on-line information to be made available for use in these tableaux shall be the Tactical Database.

C. TACTICAL PLOT GRAPHIC TOOLS

As previously mentioned, the Tactical Plot (TacPlot) is a graphical representation of the spatial relationship between Ownship and tactically significant objects. A TacPlot is a window for graphical input and output. [Ref. 1]

The availability of utility lines, circles, and other graphic objects is necessary for the user to manage the tactical picture and enhance tactical decision-making. [Ref. 1]

Also, according to [Ref. 1], graphic tools shall be considered a subset of special points managed and maintained in the Tactical Database. They shall be defined by lat/long position and dimension attributes. These graphics shall appear as background shapes on the TacPlot and shall not obscure tracks and other special points.

Graphic tools may be filled or unfilled, have variable width borders, and have text associated with each object. As a minimum, available border styles shall support creation of solid, dotted, and dashed lines. [Ref. 1]

A set of graphic tools that shall be stored in the Tactical Database and available for use on the TacPlot [Ref. 1] is described below.

1. Line

Lines may originate at any hooked position, symbol, or designated lat/long on the TacPlot. The Line function displays a straight line originating from the preceding hooked position to the current position of the Ball Tab, moving with the Ball Tab as the user moves the trackball.

2. Circle

Circles may originate at any hooked position, symbol, or designated lat/long on the TacPlot. The Circle function displays a circle centered on the preceding hooked position with radius equal to the distance between origin and current position of the Ball Tab, varying with the Ball Tab as the user moves the trackball.

3. Range Circle

This function displays a circle whose center is the Ball Tab with fixed, user entered, radius. The range circle will float in conjunction with the Ball Tab movement and may be fixed to any desired point on the TacPlot using the Mark function.

4. Expanding Circle

This function depicts a circular area of probability of a track/target associated with a time and position on the TacPlot. From origin, a circle expands at a rate determined by a user entered speed estimate for the track.

5. Moving Circle

This function generates a circular area of fixed radius which moves across the TacPlot from its origin (hooked position) with a user entered course and speed.

6. Ellipse

Ellipses may originate at any hooked position, symbol, or designated lat/long on the TacPlot. The Ellipse function displays an ellipse centered on the desired position with the user entered values for major and minor axes. This function may also operate in a "rubber band" fashion where the shape is stretched and sized based on Ball Tab movement in relation to a previously hooked position. The ellipse may be slewed to any

position on the TacPlot and rotated up to 180 degrees before being fixed using the Mark function.

7. Rectangle

Rectangle may originate at any hooked position, symbol, or designated lat/long on the TacPlot. The Rectangle function displays a square or rectangular shape centered on the desired position with user entered length/width values. If only one value is entered, a square will be generated. This function may also operate in a "rubber band" fashion where the shape is stretched and sized based on Ball Tab movement in relation to a previously hooked position. The rectangle may be slewed to any position on the TacPlot and rotate up to 180 degrees before being fixed using the Mark function.

8. Arc

Arcs may originate at any hooked position, symbol, or designated lat/long on the TacPlot. This function shall operate in a "rubber band" fashion where the shape is stretched and sized based on Ball Tab movement in relation to a previously hooked position. The arc may be slewed to any position on the TacPlot and rotated up to 180 degrees before being fixed using the Mark function.

III. DESIGN METHODOLOGY

A. OBJECT-ORIENTED TERMINOLOGY

The primitive element of an object-oriented programming language is the **object**. An object encapsulates both data (attributes or variables) and functions (operations or methods). That is, an object remembers certain information and knows how to perform certain operations. [Ref. 6]

Individual objects may have similarities. In any given application, some objects will behave differently from one another while other objects will behave in a like manner. Objects which share both the same behavior and structure are said to belong to the same class. A **class** is a generic specification for an arbitrary number of similar objects. A class can be thought of as a template for a specific kind of object. The objects of a class are called **instances** of that class. [Ref. 6]

The terms class and type are sometimes used interchangeably. Herein the term type will be used in accordance with more conventional programming languages. A **type** is any data type, whether it is provided by the system (e.g., integer, real, character, etc.) or user defined (i.e., a class or a set of possible values, such as (0..59), {N,S}, etc.).

A class may inherit both behavior and structure from another class, called a **superclass**. A class which inherits from some superclass is called a **subclass**.

An **abstract class** is one that is not intended to produce instances of itself. Abstract classes are designed only to be inherited from, that is, to provide some structure and/or

behavior to one or more subclasses. A class that is designed to be instantiated is called a **concrete class**. Concrete classes are designed primarily so that their instances are useful, and secondly so that they may also be inherited from. [Ref. 6]

Both abstract and concrete classes may make use of inheritance. However, abstract classes are generally used only to factor out attributes and/or operations that are common to more than one class, putting them in one place so that any number of subclasses may make use of them. [Ref. 6]

In the definition of a class it is necessary to describe the set of attributes that characterize the class and the set of operations that are defined for that class. As some attributes and operations may be inherited from other classes, it is also necessary to describe the class' parent classes (superclasses).

The terminology of object-oriented programming (OOP) can vary greatly from one system or language to another. In this work, the attributes that characterize a class (object) will be referred to as **variables** and the operations (or procedures) defined for that class (object) will be referred to as **methods**. [Ref. 7]

The variables that form a class can be divided into two categories: class variables and instance variables. A **class variable** is shared in both name and value by all instances (objects) of a class. An **instance variable** is shared in name only by all instances (objects) of a class. That is, each object has its own local version of an instance variable, while all the objects of the class access the same class variable. [Ref. 7]

A **composite object** is an object that contains other objects as instance variables or as class variables. This occurs when instance variables or class variables are themselves objects.

Inheritance can be formally (but simply) defined as a code sharing mechanism. It allows a new class to be defined based upon the definition of an existing class, without having to copy all of the existing code. A subclass inherits all variables and methods defined for its superclass (including those variables and methods which were inherited by that superclass from some other class). [Ref. 7]

Single inheritance is when a class is allowed to have only one superclass. **Multiple inheritance** allows a class to have several superclasses. **Selective inheritance** (also called **exclusive inheritance**) is a form of inheritance in which only some methods and/or some variables are inherited from a superclass or superclasses.

Nelson [Ref. 7] proposed the following way of representing class definitions in a language-independent manner (Figure 1):

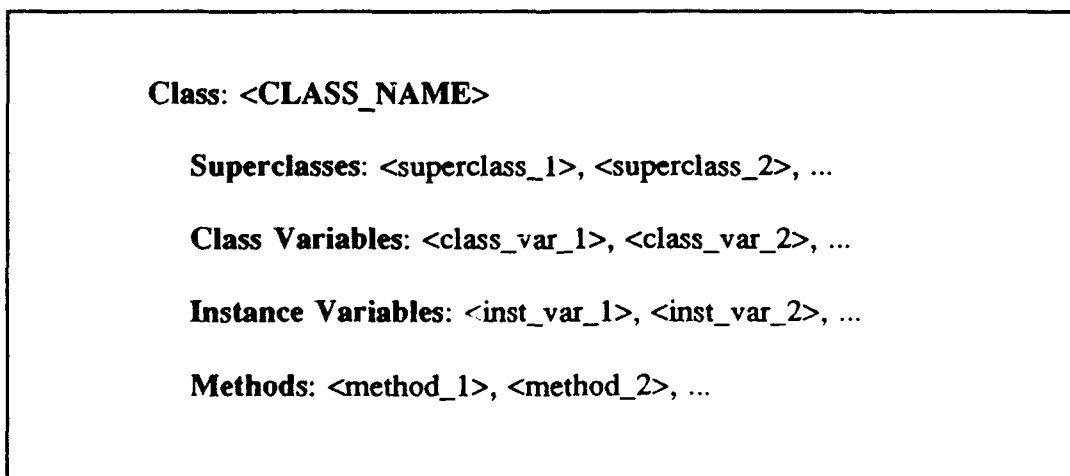


Figure 1: The Class Definition

This representation can be expanded in order to include the variables' types and default values, as follows (Figure 2):

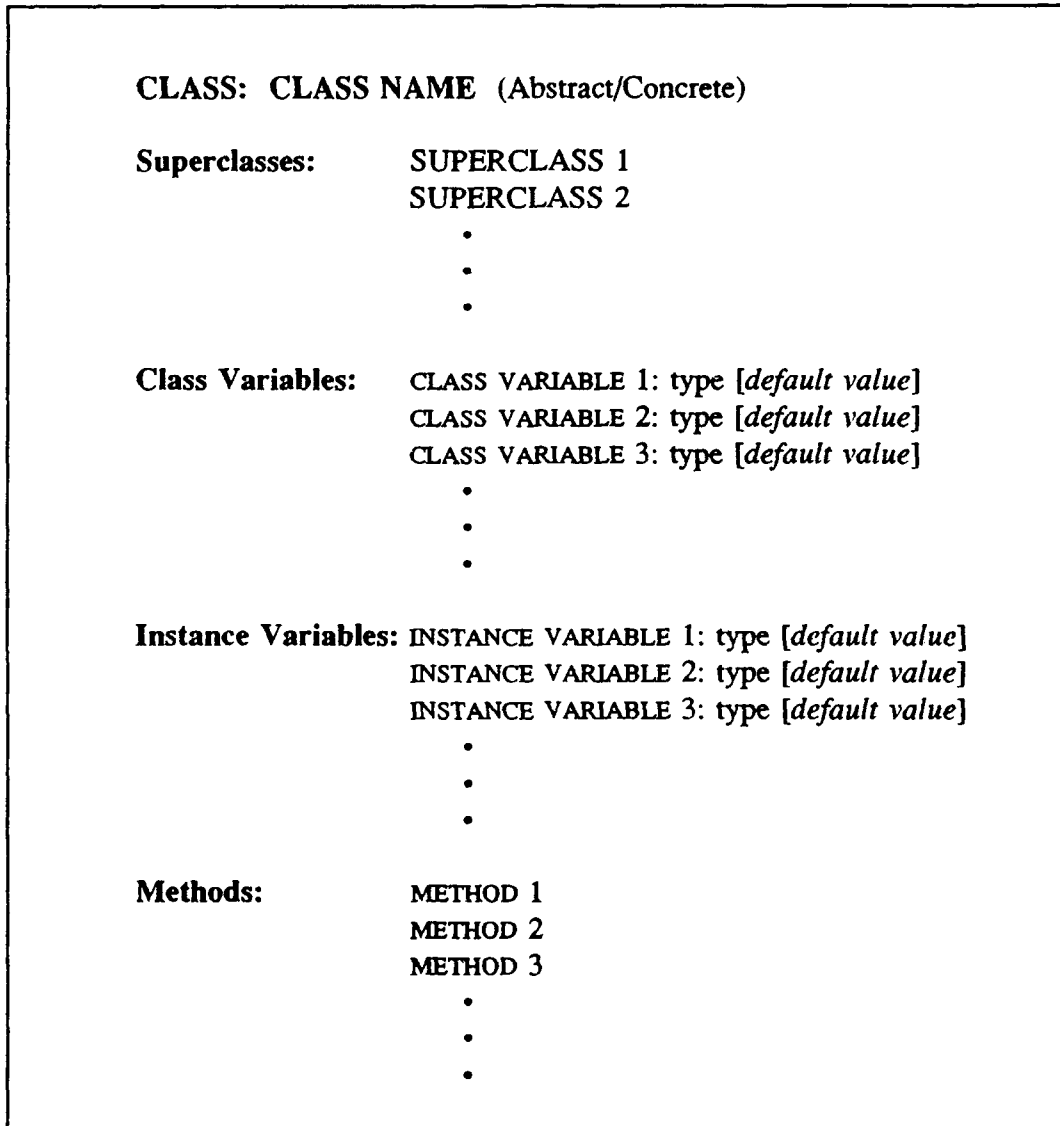


Figure 2: The Extended Class Definition

Note that the addition of the term (abstract/concrete) after CLASS NAME indicates whether or not the class was designed to be instantiated. Even though this information is

not strictly necessary for the class definition, it helps the user to better understand the role played by the class in the class hierarchy.

Note also the convention that class names are written in CAPITALS, variable and method names are written in SMALL CAPITALS, variable types are written in normal font, and variable values are written in *italics*. This form of representation will be used in this thesis.

B. OBJECT-ORIENTED DESIGN METHODOLOGIES

To date, there is no design methodology that is universally accepted by the object-oriented programming community. However, several such methodologies have been proposed. They are all somewhat similar in their approach to identifying and refining the objects and in organizing them into class hierarchies. Several of these methodologies will now be reviewed.

Booch [Ref. 8] suggests the following steps as a design method:

- Identify the objects and their attributes.
- Identify the operations that affect each object and the operations that each object must initiate.
- Establish the visibility of each object in relation to other objects.
- Establish the interface of each object.
- Implement each object.

Coad and Yourdon [Ref. 9] specify the following major steps of Object-Oriented Analysis:

- Identifying objects.
- Identifying structure.
- Identifying subjects.
- Defining attributes.
- Defining services (methods).

The purpose of the "identifying subjects" step is to identify mechanisms for controlling how much of a model a reader (analyst, manager, or customer) is able to consider and comprehend at one time. This step is not related directly to the design of the objects itself; rather it is a way to provide for better readability of the design.

Winblad, Edwards, and King [Ref. 10] propose four fundamental steps to guide the object-oriented design:

- Identifying and defining objects and classes.
- Organizing relationships among classes.
- Cultivating frameworks (abstract classes) in a hierarchy of classes.
- Building reusable class libraries and application frameworks.

Booch [Ref. 11] states that the process of object-oriented design generally tracks the following order of events:

- Identification of the classes and objects at a given level of abstraction.
- Identification of the semantics of these classes and objects.
- Identification of the relationships among these classes and objects.

- Implementation of the classes and objects.

The methodology used for the design of the objects and classes in this thesis is somewhat similar to the methodologies mentioned above. It does not, however, follow any of them in particular. The methodology used resulted from a research project in object-oriented design [Ref. 12] and from the experience acquired during the design of the Tactical Database.

C. THE DESIGN PROCESS

The process of object-oriented design starts with the discovery of the classes and objects that represent the problem domain. It stops whenever the designer finds that there are no new abstractions or mechanisms that might be used to modify the class hierarchy [Ref. 11]. The result of an object-oriented design is a hierarchy of classes [Ref. 10].

1. Why Organize the Classes in a Hierarchy

Classification is a basic human method of organization [Ref. 6]. Most practical activities, whether on an individual or social level, involve classification. In object-oriented programming, classification is achieved by defining classes and organizing them in a hierarchy. Class hierarchies are created by means of a mechanism provided by all object-oriented programming languages: inheritance.

Inheritance can also be considered as the ability of one class to define the behavior and data structure of its instances based upon the definition of another class (or classes). With inheritance it is possible to create a new class of objects as a refinement of another,

to factor out the similarities between classes, and to design and specify only the differences for the new class. In this way, new classes can be created quickly. [Ref. 6]

As previously defined, inheritance is a code sharing mechanism. It allows a new class to be defined based upon the definition of an existing class without having to copy all of the existing code [Ref. 7]. Inheritance can also be thought of as a key reusability mechanism [Ref. 13].

The main reason for organizing the classes in a hierarchy is that this type of organization allows the user to obtain simpler and more easily maintainable code while saving considerable storage space. It allows the developer to quickly and easily reuse existing code, therefore minimizing code duplication.

D. THE METHODOLOGY

The methodology used for the design of the Tactical Database consisted basically of the following steps:

- **Identification of the objects and classes.**
 - Initial definition of the objects and classes.
 - Analysis of the object's variables.
 - Analysis of the object's methods.
- **Refinement of the objects and classes.**
 - Global analysis of the set of objects and classes.
 - Individual analysis of each class (object).
- **Organization of the classes in a hierarchy.**

- Analysis of the implementation language.
- Construction of the hierarchies.
- Review of the classes' variables and methods.

It is important to realize that the design as a whole is an iterative process. That is, each step may have to be reviewed several times until the hierarchy of classes constructed is considered satisfactory.

1. Identification of the Objects and Classes

The primary motivation for identifying objects and classes is to match the representation of a system as closely as possible to the conceptual view of the real world [Ref. 9]. The design of an object-oriented system begins with the identification of the objects; finding these objects is perhaps the most challenging phase of an object-oriented design process [Ref. 10].

This step has been divided into three substeps: Initial Definition of the Objects and Classes, Analysis of the Object's Variables, and Analysis of the Object's Methods.

a. Initial Definition of the Objects and Classes

The purpose of this step is to come up with a list of potential objects and/or classes. In this phase of the design, the objects may not necessarily be represented in the format proposed in Section A of this chapter, as it is still too early to know the objects' superclasses, class and instance variables, variables' types, and variables' default values. The description of each object in the list contains only the object's initial set of variables (with their probable types) and methods.

Identification of the objects and classes requires a considerable knowledge of the problem domain (or problem space). The developer must study the problem requirements and learn the terminology and fundamentals of the problem domain as well.

In order to efficiently determine the objects required to model the system, it is necessary first to determine the goals of the system [Ref. 6]. Then, for each goal, find what objects are needed to accomplish it.

Shlaer and Mellor [Ref. 14] suggest that candidate classes and objects can usually be derived from the following sources:

- Tangible things: Cars, radar data, sensors, planes.
- Roles: Mother, engineer, systems analyst.
- Events: Landing, request, print.
- Interactions: Loan, meeting, intersection.

Ross [Ref. 15] offers a similar list of sources of potential classes and objects:

- People: Humans who carry out some function.
- Places: Areas set aside for people or things.
- Things: Physical objects that are tangible.
- Organizations: Formally organized collections of people, resources, facilities, and capabilities having a defined mission, whose existence is largely independent of individuals.
- Concepts: Principles or ideas not tangible per se.
- Events: Things that happen, usually to something else at a given date and time, or as steps in an ordered sequence.

Coad and Yourdon [Ref. 9] suggest that, in order to find objects and classes, the developer should look for:

- Structure: "Kind-of" and "part-of" relationships.
- Other Systems: External systems with which the application interacts.
- Devices: Devices which exchange data and control information with the system.
- Events Remembered: Point in time or historical event that must be observed and recorded by the system.
- Roles Played: The different roles human beings play in interacting with the system under study.
- Locations: Physical locations, offices, and sites important to the application.
- Organizational Units: Groups to which people belong.

All of these ideas are useful in finding the candidate objects. It should be noted, however, that sources of potential objects are virtually unlimited and the problem space is, obviously, the best place to look for these sources.

b. Analysis of the Object's Variables

The purpose of this step is to analyze the variables defined for the objects that resulted from step 1.a and list the observations that are significant to the construction of the class hierarchy.

The following guidelines are used during this phase:

- Analyze the object's variables to make sure that each variable is really necessary for the description of the object (class). Unnecessary variables should, obviously, be discarded, and missing variables should be added. Deciding which set of attributes (variables) best describe the object and/or which attributes are essential to a

representation of the object as close as possible to the real world situation is a task that requires a good working knowledge of the problem space.

- Look for variables that are common to any group of objects (classes). These commonalities are fundamental to the creation of the class hierarchy.
- Look for variables that have the same value for all objects (classes). Variables that have the same value for all objects of a class are potential class variables. Variables that do not change in value for all objects of all classes may be treated as constants. Analyze constants to decide whether they should be implemented as variables or methods.
- Observe variables that can be calculated or derived from other variables. These variables may be replaced by methods which calculate them.
- Observe variables that can be decomposed into more elementary variables. These variables may themselves be defined as individual objects. Objects that contain variables that can be decomposed are potential composite objects.
- Observe variables that can be combined to form an object. Some variables may represent the components of an object and, therefore, a new object can be defined including these variables.
- Observe variables that are defined for only one class. Classes that contain these variables may be classes without subclasses in the class hierarchy.

c. Analysis of the Object's Methods

The purpose of this step is to analyze the methods defined for the objects which resulted from step 1.a and list the observations that are important to the construction of the class hierarchy.

The following guidelines are used during this phase:

- Look for methods that are common to all or to a group of classes. Commonalities of methods are fundamental for the creation of the class hierarchy.
- Every concrete class should have, as a minimum, a set of methods whose purpose is to create, delete, maintain, and display the instances (objects) of that class. Abstract classes do not need these methods as they have no instances.

- In order to enforce encapsulation, it is necessary to define for every variable a method to access it (such as GET "VARIABLE") and a method to update it (such as SET "VARIABLE").

2. Refinement of the Objects and Classes

After the initial analysis of the object's variables and methods, it is necessary to refine them before the class hierarchy is constructed. The first task in this step is to analyze the two lists of observations obtained in steps 1.b and 1.c. This analysis is done in two substeps: global analysis of the set of objects and classes; and individual analysis of each class (object).

a. Global Analysis of the Set of Objects and Classes

The set of objects and classes is now analyzed as a whole, focusing mainly on the following three aspects: elimination of redundant information and addition of necessary (but lacking) information; identification of composite objects; and separation of the variables into class and instance variables.

(1) *Elimination of Redundant Information and Addition of Necessary Information.* Eliminate variables that can be derived from other variables. This can be done by defining a method that calculates the desired value using the information provided by the existing variables. It is not necessary to define a variable to store information that can be calculated.

With basis in the problem space, analyze the objects to make sure that all necessary information can be obtained through the set of methods defined for them; if not, add the necessary variables and/or methods, but be sure that the new methods/variables will not provide redundant information.

(2) Identification of Composite Objects. Define as individual objects those variables that can be decomposed into more elementary variables. The newly defined objects may themselves have variables that can be further decomposed into more elementary variables. This process continues until the object's variables can no longer be decomposed.

Objects that have other objects as its variables are composite objects. In a composite object, the relationship between the object and its variables is described as an "is-part-of" relationship, in which the object is viewed as a whole and its variables are viewed as its component parts.

(3) Separation of the Variables into Class and Instance Variables. A variable that has the same value for all objects (instances) of a class can be defined a class variable. Those variables whose values vary among objects are instance variables.

Some class variables may not be expected to change in value, in which case they may be more properly considered to be constants. As mentioned in Section 1.c, it is advisable (for encapsulation purposes) to define a method to access the value of each variable (GET "VARIABLE") and another method to update this value (SET "VARIABLE"). Since a constant value does not need to be updated, the method SET "VARIABLE" is not necessary, and this constant value could be simply provided by the method GET "VARIABLE". The class variable itself does not need to be defined. This saves some storage space.

Therefore, two approaches are suggested: the first approach is to define the class variable plus the methods GET "VARIABLE" and SET "VARIABLE"; the second

approach is to define only the method GET "VARIABLE", which is responsible for returning the constant value of what would have otherwise been the variable. The class variable itself is not defined.

When to use one approach or the other is a matter of design decision. However, when a class variable is defined for a class that has subclasses, a problem may occur. If the value of the class variable happens to change in one or more of the subclasses, the class variable can no longer be defined as a class variable for the superclass of these classes. In this case, the class variable has to be eliminated and new class variables have to be defined for each of the subclasses (note that this problem is implementation-dependent).

On the other hand, if a method were defined for the superclass to provide the constant value, the only change necessary would be to redefine the methods in each one of the affected subclasses. This approach makes maintenance easier.

b. Individual Analysis of Each Class (Object)

This step is essentially the same as step 2.a, except that now every class (object) is analyzed separately. Focus is on the same three aspects: elimination of redundant information and addition of necessary information; identification of composite objects; and separation of the variables into class and instance variables.

By analyzing each class individually, the developer is able to work on details that could not be detected or were too specific to be handled in step 2.a.

3. Organization of the Classes in a Hierarchy

a. Analysis of the Implementation Language

The design of the class hierarchy depends directly on the facilities provided by the implementation language or OODBMS used. In this phase, the following questions should be answered:

- Does the system provide single, multiple, or selective inheritance?
- If the system provides multiple inheritance, what are its conflict resolution rules (i.e., the way that the system handles the conflicts that may arise in the names of variables and methods)?
- Can methods inherited from a superclass can be overridden (redefined) in the subclasses?
- Can variables inherited from a superclass can be overridden (redefined) in the subclasses?
- How is the schema evolution handled within the system (i.e., the way that the system handles run-time changes in the class hierarchy)?
- Does the system offer early (compile-time) binding and/or late (run-time) binding?

(1) Does the system provide single, multiple, or selective inheritance? The form of inheritance provided by the system will have a direct influence on the design of the class hierarchy. For example, a design using a system which allows multiple inheritance will probably be different from a design using a system that only allows single inheritance.

The GemStone Data Management System [Ref. 16] and Object Pascal [Ref. 17] provide only single inheritance. The ORION Database System [Ref. 18], C++ [Ref. 19], Vbase Integrated Object System [Ref. 20], ONTOS Object Database [Ref. 21],

and the programming language Eiffel [Ref. 13] all provide multiple inheritance. CommonObjects [Ref. 22] provides selective inheritance of methods.

(2) If the system provides multiple inheritance, what are its conflict resolution rules? Multiple inheritance, even though giving more flexibility to the design, should be used carefully as conflicts involving the names of variables and methods may occur.

The approach used in many systems to resolve name conflicts among superclasses of a given class is as follows: if an instance variable or a method with the same name appears in more than one superclass of a class C, the one chosen by default is that of the first superclass in the list of (immediate) superclasses for C [Ref. 7]. Thus, this default conflict resolution scheme depends on the order in which the superclasses of a class are listed.

Unlike most systems, the ORION Database System allows the user to explicitly change this order at any time. Further, ORION provides two ways in which a user can override the default conflict resolution: the user may either explicitly inherit one instance variable or method from among several conflicting ones; or explicitly inherit one or more instance variables or methods that have the same name and rename them within the new class definition. [Ref. 18]

Other approaches are also possible. In the Eiffel programming language, for example, name conflicts are prohibited and result in a compilation error. However, name conflicts may be removed by renaming the conflicting variables and/or methods in the subclasses. [Ref. 13]

(3) Can methods inherited from a superclass be overridden (redefined) in the subclasses? Most systems allow the redefinition of inherited methods in the subclasses. Redefinition permits a method to be modified or customized to meet the specific needs of the subclass, while keeping the same name of the parent's class method.

(4) Can variables inherited from a superclass be overridden (redefined) in the subclasses? Some OOP languages, such as Object Pascal, do not allow the redefinition of inherited variables in the subclasses. A subclass cannot have a variable with the same name of an inherited variable. This results in a name conflict and, consequently, in a compilation error. Most systems, however, do permit variables to be renamed in the subclasses, so that name conflicts are eliminated.

(5) How is the schema evolution handled within the system? The developer has to be aware of how the system handles run-time changes in the class hierarchy such as addition, removal, reordering, modification, or renaming of classes, methods, and variables.

There are at least two approaches used in a schema evolution methodology which, even though they do not directly influence the design of the class hierarchy, are worth mentioning. The first approach, screening, is to defer (possibly indefinitely) modifying the persistent store; values are filtered or corrected as they are used. The second approach, conversion, immediately changes all instances of the class to the new class definition, ensuring that auxiliary definitions agree with the new definition. [Ref. 16]

These two approaches offer the choice of "pay me now or pay me later". In the screening approach, execution speed is compromised by screening. In the

conversion approach, much time can be consumed at the time a class is modified.

[Ref. 16]

It should be noted that there is also a third alternative: versioning. With this approach, existing objects are never converted to the new definition. This results in the existence of two (or more) kinds of objects: "old" objects which follow the original class definition and "new" objects which utilize the new (modified) class definition.

(6) Does the system offer early (compile-time) binding and/or late (run-time) binding? Binding is the point in a program's life when the caller of a method (procedure) is given the address of ("is bound to") the method (procedure). In early binding, this happens when the program is compiled and linked. Most traditional languages use only early binding. Early binding requires that the types of all variables be known at compile time and do not change after that (this is one of the characteristics of strongly-typed languages). [Ref. 23]

In late binding, by contrast, the caller is not bound to the method (procedure) until the method call is actually made. This allows variables to be assigned values of different types at different points of program execution. Performance degradation may occur, however, due to this dynamic (late) binding of methods with type checking completed at run-time. Modifying a class changes the environment and may invalidate bindings. Late binding will attempt to adapt to any changes in the environment. Late binding provides increased flexibility at the expense of efficiency. [Ref. 16]

b. Construction of the Hierarchies

The commonalities of methods and/or variables among classes is the fundamental factor in the construction of the class hierarchies. The following guidelines for the analysis of inheritance relationships between classes have been proposed [Ref. 6]:

- Model a "kind-of" hierarchy.
- Factor common methods as high as possible.
- Make sure that abstract classes do not inherit from concrete classes.
- Eliminate classes that do not add functionality.

(1) Model a "kind-of" hierarchy. A "kind-of" hierarchy means that every class should be a specific kind of its superclasses. Subclasses should support all of the methods and variables defined by their superclasses and possibly more. Ensuring that this is so will make the classes more reusable as it makes it easier to see where, in an existing hierarchy, a new class should be placed. [Ref. 6]

When a subclass includes only part of the methods and/or variables defined by its superclasses, it is good practice to create an abstract class with all of the methods and variables common to the class and superclass. The hierarchy is then restructured in such a way that the class and its former superclass both become subclasses of the newly created abstract class.

(2) Factor common methods as high as possible. Factoring common methods as high as possible is based on the principle that if a set of classes all support a common method and/or variable, they should inherit that method and/or variable from a common

superclass. If a common superclass does not already exist, then create one, and move the common methods and/or variables to it. This new superclass will probably be an abstract class. [Ref. 6]

(3) Make sure that abstract classes do not inherit from concrete classes. Abstract classes should not inherit from concrete classes because abstract classes, by their nature, support their methods and/or variables in implementation-independent ways. Concrete classes may specifically depend upon implementation. If the design happens to have such an inheritance, it is possible to solve the problem by creating another abstract class from which both the abstract and concrete classes can inherit their common methods and/or variables. [Ref. 6]

(4) Eliminate classes that do not add functionality. According to Wirfs-Brock, Wilkerson, and Wiener [Ref. 6], classes that have no methods should ordinarily be discarded. If a class inherits a method that it will implement in a unique way, then it adds functionality in spite of having no methods of its own, and should be kept. On the other hand, abstract classes that define no methods have no use [Ref. 6]. However, the statement "classes that have no methods should ordinarily be discarded" is questionable. A class, even though having no methods of its own, can still have several variables that are used to store important information about the objects. Therefore, only the classes that do not add functionality and/or do not store any information should be discarded.

c. Review of the Classes' Variables and Methods

During the construction of the class hierarchies, achieving a perfect match among the classes' variables and methods is highly unlikely. For example, when a class

is defined as a subclass of another class, the subclass inherits all the variables and methods of its superclass. However, the subclass may not need all these inherited variables and/or methods. If the system allows selective inheritance, these variables and/or methods can be excluded, but if the system does not allow selective inheritance, some classes will have to be slightly modified. These changes in the definition of certain classes are necessary for the achievement of similarity among classes. Some variables and/or methods that are not strictly necessary might be added to a class just to make it fit into the set of subclasses of a given class. This seems to contradict the philosophy of step 2.a.1 (Elimination of Redundant Information and Addition of Necessary Information). However, the purpose of these changes in the classes' variables and/or methods is to allow the design of the simplest class hierarchy possible while still keeping the semantics inherent to the problem space. The hierarchy of classes must reflect the real world situation, i.e., a contrived class hierarchy should not be created just for the sake of capturing certain commonalities [Ref. 9]. Therefore, the variables and/or methods added to a class should be meaningful in terms of the problem space.

E. COMMENTS

The goal of the proposed methodology is to provide a simple and systematic way to approach the problems of object-oriented design. Even though the steps that form the methodology were proposed in a certain logical order, the methodology is flexible in the sense that the order in which the steps are followed can be changed according to the

designer's preference. The iterative nature of the design process is evidence that the order in which each step is accomplished is flexible.

Presently, different object-oriented languages provide different facilities. This explains the necessity of an analysis of the implementation language before the construction of the class hierarchies.

However, some of these facilities are typical of object-oriented languages and, therefore, are provided by all of them. Therefore, in order to produce a design that is independent of the object-oriented language used it is reasonable to assume the following:

- All object-oriented languages allow, at least, single inheritance.
- The redefinition of inherited methods in the subclasses of a certain class is allowed in all object-oriented languages.
- The redefinition of inherited variables in the subclasses of a certain class is not allowed in all languages (Object Pascal [Ref. 17], for example, does not allow this form of redefinition).
- Not all object-oriented languages, including the OODBMSs considered herein, allow run-time changes in the definition of classes or in the class hierarchy.

It is expected, however, that these differences will eventually disappear with the evolution of the object-oriented languages. This will give the designer more freedom to produce a language-independent design.

IV. DESIGN OF THE TACTICAL DATABASE

The design of the Tactical Database was based on the basic requirements presented in [Ref. 1]. However, during the design process, some of these requirements were slightly modified in order to better fit (according to this author's point of view) the necessities of the Tactical Database in its role in the Low Cost Combat Direction System.

One of the requirements presented in [Ref. 1] is that the Tactical Database should manage and maintain a set of graphic tools (e.g., lines, circles, arcs, ellipses, rectangles, etc.) that would be used in the Tactical Plot. However, since a set of graphic tools does not directly represent information relative to a combat environment, it was decided not to include these graphic tools as part of the Tactical Database (it should be possible, however, to store the Tactical Database and the set of graphic tools in the same OODBMS, but as two independently designed databases). Therefore, graphic tools were not considered in the design of the Tactical Database.

A. IDENTIFICATION OF THE OBJECTS AND CLASSES

1. Initial Definition of the Objects and Classes

In [Ref. 1], a set of objects for the Tactical Database was identified. Thus, this initial phase of the design consisted basically of the analysis of the set of objects proposed, followed by an analysis of the problem space in order to determine if the initial set of objects was adequate to represent the real world situation.

The set of objects proposed in [Ref. 1] was listed in Chapter II, Section A.2 along with a brief description of each object. Some of these objects were modified in order meet additional requirements as specified in Chapter II, Section B. Variables were added (and some were just renamed) to the class OWNSHIP and methods were added to the classes OWNSHIP, TENTATIVE TRACK, WAYPOINT, and POSITION AND INTENDED MOVEMENT. Also, the classes AREA and ROUTE were added to the set of objects. This modified set of objects is presented in this section for further review.

Notice that the objects (classes) are not yet represented in the format proposed in Chapter III, Section A because, as previously mentioned, in this phase of the design the objects' superclasses, the class and instance variables, the composite classes, and the variables' types and default values are not completely defined yet. The purpose of this initial phase is to compile a potential list of objects without going into details on how the objects are defined. Following is the current version of the set of objects proposed in [Ref. 1]:

Class: OWNSHIP (Concrete)

Variables: Geographical Position {latitude:longitude}
Time of Position {hh:mm:ss}
Course {dd:mm:ss}
Speed {knots}
Origin {Local Manual}
Magnetic Variation {dd:mm:ss:east/west}
Ownship Set {dd:mm:ss:true/magnetic}
Ownship Drift {knots}
Wind Direction {dd:mm:ss:true/magnetic}
Wind Speed {knots}
Navigation Source {inertial, SATNAV, Omega, DR(dead reckoning)}
Julian Date {0001..9366}

Greenwich Mean Time (hh:mm:ss)
Marktime (hh:mm:ss)
Close CPA (distance range:time range)
Collision CPA (distance range:time range)
Tracknumber (0..9999)
Type {string of ten characters}
Track History {boolean}

Methods: Monitor Ownship Position
Navigational Computation
CPA Processing
Ownship Repositioning
Dead Reckoning
Bearing and Range from Position to Position

Description: An object of class OWNSHIP represents the ship that is operating the Low Cost Combat Direction System. It therefore follows that there should be only a single current instance of OWNSHIP for each Tactical Database. However, in order to store the ship's history information, other instances of OWNSHIP could be created to represent the ship at different times.

Class: TENTATIVE TRACK (Concrete)

Variables: Geographical Position (latitude and longitude)
Relative Position (bearing and range)
Time of Position (hh:mm:ss)
Course (dd:mm:ss)
Speed (knots)
Category (Tentative)
Identity (Unknown)
Origin (Local Manual, Local Auto)
Tracknumber (0..9999)
Type {string of ten characters}

Methods: New Track Establishment
Track Position Data
Track Course And Speed Determination
Dead Reckoning
Initial Category Assignment
Initial Identity Assignment
Track Termination

Description: An object of class TENTATIVE TRACK represents a new track generated by local sensors (radar). Once a valid course and speed is established for it, the Tentative Track will become a Firm Track (i.e., an Air Track, a Surface Track, or a Subsurface Track). This can happen as a result of any one of the following conditions being met: after three manual position updates by the user, when the category is entered manually before three position updates, or when the track is manually declared firm by the user.

Class: AIR TRACK (Concrete)

Variables: Geographical Position (latitude and longitude)
Relative Position (bearing and range)
Time of Position (hh:mm:ss)
Course (dd:mm:ss)
Speed (knots)
Height (feet)
Category (Air)
Identity (Unknown, Friendly, Hostile)
Origin (Local Manual, Local Auto, Remote)
Tracknumber (0..9999)
Type (string of ten characters)

Methods: CPA Processing
Track Position Update
Track Course And Speed Determination
Dead Reckoning
Track Position Prediction
Track History Processing
Manual Termination
Identification Function

Description: An object of class AIR TRACK represents a real-world object which is in the air and, consequently, has Height as one of its attributes to indicate its current altitude.

Class: SURFACE TRACK (Concrete)

Variables: Geographical Position (latitude and longitude)
Relative Position (bearing and range)
Time of Position (hh:mm:ss)
Course (dd:mm:ss)
Speed (knots)
Category (Surface)
Identity (Unknown, Friendly, Hostile)

Origin {Local Manual, Local Auto, Remote}
Tracknumber {0..9999}
Type {string of ten characters}

Methods: The same operations that apply to the class AIR TRACK.

Description: An object of class SURFACE TRACK represents a real-world object which is on the surface.

Class: SUBSURFACE TRACK (Concrete)

Variables: Geographical Position {latitude and longitude}
Relative Position {bearing and range}
Time of Position {hh:mm:ss}
Course {dd:mm:ss}
Speed {knots}
Depth {feet}
Category {Subsurface}
Identity {Unknown, Friendly, Hostile}
Origin {Local Manual, Local Auto, Remote}
Tracknumber {0..9999}
Type {string of ten characters}

Methods: The same operations that apply to the class AIR TRACK.

Description: An object of the class SUBSURFACE TRACK represents a real-world object which is underwater and, consequently, has Depth as one of its attributes.

Class: REFERENCE POINT (Concrete)

Variables: Geographical Position {latitude and longitude}
Relative Position {bearing and range}
Time of Position {hh:mm:ss}
Category {Special}
Identity {Refpoint}
Origin {Local Manual}

Methods: Enter/Update Reference Point
CPA Processing

Description: An object of class REFERENCE POINT represents a fixed point on the surface of the Earth which is used as a reference.

Class: NAVIGATION HAZARD (Concrete)

Variables: Geographical Position (latitude and longitude)
Relative Position (bearing and range)
Time of Position (hh:mm:ss)
Category (Special)
Identity (Navhaz)
Origin (Local Manual)

Methods: Enter/Update Navigation Hazard
CPA Processing

Description: An object of class NAVIGATION HAZARD represents a point which might be hazardous to navigation (icebergs, shallow waters, reefs, mines, etc.) and should therefore be avoided.

Class: MAN IN WATER (Concrete)

Variables: Geographical Position (latitude and longitude)
Relative Position (bearing and range)
Time of Position (hh:mm:ss)
Category (Special)
Identity (Man In Water)
Origin (Local Manual)

Methods: Enter/Update Man in Water
CPA Processing

Description: An object of class MAN IN WATER represents a point on the surface of the water on which a person (or a group of persons) is believed to be.

Class: WAYPOINT (Concrete)

Variables: Geographical Position (latitude and longitude)
Relative Position (bearing and range)
Time of Position (hh:mm:ss)
Steaming Route (1..6)
Category (Special)
Identity (Waypoint)
Origin (Local Manual)

Methods: Waypoint Geometry
Ownship Distance to Waypoint
Estimated Time of Arrival at Waypoint
Estimated Time Enroute to Waypoint

Description: An object of class WAYPOINT represents a destination point on the surface. Each waypoint can be viewed as a "node" in a predetermined route.

Class: DATA LINK REFERENCE POINT (Concrete)

Variables: Geographical Position (latitude and longitude)
Time of Position (hh:mm:ss)
Category (Special)
Identity (Dlrp)
Origin (Local Manual)

Methods: Enter/Update Data Link Reference Point
CPA Processing

Description: An object of class DATA LINK REFERENCE POINT represents a fixed geographic reference position common to all Link 11 participating units.

Class: FORMATION CENTER (Concrete)

Variables: Geographical Position (latitude and longitude)
Time of Position (bearing and range)
Course (dd:mm:ss)
Speed (knots)
Category (Special)
Identity (Fc)
Origin (Local Manual)

Methods: Enter/Update Formation Center
Dead Reckoning
CPA Processing

Description: An object of class FORMATION CENTER represents a moving geographic position representing the center of a group of ships steaming in formation.

Class: POSITION AND INTENDED MOVEMENT (Concrete)

Variables: Geographical Position (latitude and longitude)
Relative Position (bearing and range)
Time of Position (hh:mm:ss)
Course (dd:mm:ss)
Speed (knots)
Category (Special)
Identity (Pim)
Origin (Local Manual)

Methods: Enter/Update Position And Intended Movement
Dead Reckoning
CPA Processing
Ownship Distance to PIM
Estimated Time of Arrival at PIM
Estimated Time Enroute to PIM

Description: An object of class POSITION AND INTENDED MOVEMENT represents the Ownship or formation planned position based on a pre-computed base course and speed to arrive at a destination at the required time.

Class: USERDEFINED (Concrete)

Variables: A minimal set is:
Geographical Position (latitude and longitude)
Time of Position (hh:mm:ss)
C (character/string)
N (numeric)
F (floating value)
D (date)
T (time)
L (logical)
DD (condition)

Methods: Not specified.

Description: The Tactical Database shall allow the user to define new classes of objects at run-time and then instantiate objects of that class. A new class shall be defined either by selection from a given set of attributes and operations or by definition of new attributes or operations. [Ref. 1]

One important observation that resulted from the analysis of the proposed objects was that, except for the class USERDEFINED, all classes have a common characteristic: each one of them represents a **point** (either a fixed or nonfixed position) in the context of the LCCDS scenario. This characteristic was useful for the creation of new objects during the phase of identification of the class structure (class hierarchy).

However, in the combat environment, it is also necessary for a ship to have information about **areas** (regions) that are considered dangerous (hot areas) due, for example, to the presence of enemy warships, mines, or antiship artillery. Information about protected and unprotected areas is also important. These areas may be fixed or nonfixed. In order to store and handle this type of information, the class AREA was initially identified as follows:

Class: AREA (Concrete)

Variables: Geographical Position {latitude and longitude}
Relative Position {bearing and range}
Time of Position {hh:mm:ss}
North Limit {latitude and longitude}
South Limit {latitude and longitude}

East Limit {latitude and longitude}
West Limit {latitude and longitude}
Course {dd:mm:ss}
Speed {knots}
Identity {Hot, Protected, Unprotected}
Origin {Local Manual, Local Auto, Remote}
Area Number {integer}

Methods: Dead Reckoning
CPA Processing
Draw Area

Description: An object of class AREA represents a fixed or nonfixed geographical region within whose limits navigation may be considered dangerous (hot areas), protected, or unprotected.

Another type of information that is fundamental for a ship's navigation is the information about routes. A route is the way taken or planned from one place to another. In general, a route is composed of a set of intermediate points (destinations or waypoints). A route can then be defined as the line connecting the start point to the destination point, which results from drawing straight lines connecting all of the intermediate points.

When going from one place to another, it is necessary for the safety of the ship to know whether the route taken crosses dangerous (hot) areas or not. Thus, the routes can be classified as hot, protected, or unprotected.

One of the requirements mentioned in [Ref. 1] is that the LCCDS has to allow the specification of up to six routes with up to 50 destinations (waypoints) per route. In general, a route is entered manually by the user. However, information about known routes can also be received through the Link 11 system.

Therefore, in order to store information about routes, the class ROUTE was initially defined as follows:

Class: ROUTE (Concrete)

Variables: Route Number {1..6}
Start Point {latitude and longitude}
Waypoints {array 1..48}
End Point {latitude and longitude}
Identity {Hot, Protected, Unprotected}
Origin {Local Manual, Remote}

Methods: Route Geometry
Draw Route

Description: An object of class ROUTE represents a set of points (waypoints) that describe the way taken or planned from one place (start point) to another (end point or final destination).

With the addition of the classes AREA and ROUTE, the set of classes (objects) initially defined for the Tactical Database should be adequate to represent the real world scenario of a combat situation. However, since the design as a whole is an iterative process, classes could be added to this set or even eliminated from it, as a consequence of the other phases of the design.

2. Analysis of the Objects' Variables

This step consists of listing observations about the proposed classes' variables which could be considered as important or significant to the construction of the class hierarchy. The following observations fit into this category:

- GEOGRAPHICAL POSITION, TIME OF POSITION, and ORIGIN are common to all classes (objects).
- GEOGRAPHICAL POSITION has two components: Latitude and Longitude.
- Latitude and Longitude can be further decomposed into Degree, Minute, Second, and Hemisphere.
- TIME OF POSITION has four components: Hour, Minute, Second, and Time Zone.
- RELATIVE POSITION is common to all classes except OWNERSHIP and DATA LINK REFERENCE POINT.
- RELATIVE POSITION has two components: Range and Bearing.

- Bearing can be further decomposed into Angle and Reference North (True or Magnetic).
- OWNSHIP SET and WIND DIRECTION can be decomposed into Angle and Reference North (True or Magnetic).
- MAGNETIC VARIATION can be decomposed into Angle and Hemisphere (East or West).
- CLOSE CPA and COLLISION CPA can both be decomposed into distance range and time range.
- The geographical position of any point in the Tactical Database scenario can be calculated from its relative position with respect to Ownship's geographical position. Alternatively, the relative position of any point could be calculated from its geographical position plus Ownship's position.
- ORIGIN has the value *Local Manual* for all instances of classes OWNSHIP, REFERENCE POINT, NAVIGATION HAZARD, MAN IN WATER, WAYPOINT, DATA LINK REFERENCE POINT, POSITION AND INTENDED MOVEMENT, and FORMATION CENTER.
- CATEGORY is common to all classes except OWNSHIP and AREA.
- CATEGORY has the value *Special* for all instances of classes REFERENCE POINT, NAVIGATION HAZARD, MAN IN WATER, WAYPOINT, DATA LINK REFERENCE POINT, POSITION AND INTENDED MOVEMENT, and FORMATION CENTER.
- CATEGORY has the value *Tentative* for all instances of class TENTATIVE TRACK, the value *Air* for all instances of class AIR TRACK, the value *Surface* for all instances of class SURFACE TRACK, and the value *Subsurface* for all instances of the class SUBSURFACE TRACK.
- IDENTITY is common to all classes except OWNSHIP.
- IDENTITY has the value *Unknown* for all instances of class TENTATIVE TRACK, the value *Refpnt* for all instances of class REFERENCE POINT, the value *Navhaz* for all instances of class NAVIGATION HAZARD, the value *Man in Water* for all instances of class MAN IN WATER, the value *Waypoint* for all instances of class WAYPOINT, the value *Dlrp* for all instances of class DATA LINK REFERENCE POINT, the value *Fc* for all instances of class FORMATION CENTER, and the value *Pim* for all instances of class POSITION AND INTENDED MOVEMENT.

- TRACK NUMBER and TYPE are common to the classes OWNSHIP, TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK.
- COURSE and SPEED are common to the classes OWNSHIP, TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, SUBSURFACE TRACK, AREA, FORMATION CENTER, and POSITION AND INTENDED MOVEMENT.
- TRACK HISTORY is defined only for class OWNSHIP.
- HEIGHT is defined only for class AIR TRACK.
- DEPTH is defined only for class SUBSURFACE TRACK.
- STEAMING ROUTE is defined only for class WAYPOINT.
- Except for HEIGHT and DEPTH, the classes TENTATIVE TRACK, SURFACE TRACK, AIR TRACK, and SUBSURFACE TRACK all have the same variables.
- Except for RELATIVE POSITION, the classes REFERENCE POINT, NAVIGATION HAZARD, MAN IN WATER, and DATA LINK REFERENCE POINT all have the same variables.
- The classes FORMATION CENTER and POSITION AND INTENDED MOVEMENT both have the same variables.

3. Analysis of the Object's Methods

This step consists of listing observations about the proposed methods that may be relevant to the construction of the class hierarchy. The following observations were made:

- CPA PROCESSING is common to all classes except USERDEFINED, TENTATIVE TRACK, ROUTE, and WAYPOINT.
- DEAD RECKONING is common to the classes TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, SUBSURFACE TRACK, AREA, POSITION AND INTENDED MOVEMENT, and FORMATION CENTER.
- TRACK COURSE AND SPEED DETERMINATION is common to the classes TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK.

- The classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK all have the same methods.
- Except for DEAD RECKONING, the classes REFERENCE POINT, NAVIGATION HAZARD, MAN IN WATER, DATA LINK REFERENCE POINT, POSITION AND INTENDED MOVEMENT, and FORMATION CENTER all have the same methods.
- Only the classes TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK have methods to delete objects, namely TRACK TERMINATION and MANUAL TERMINATION.
- Only the class TENTATIVE TRACK has a method to display one or more objects, namely TRACK POSITION DATA.
- The methods OWNERSHIP DISTANCE TO WAYPOINT and OWNERSHIP DISTANCE TO PIM perform similar tasks. The same is true of the methods ESTIMATED TIME OF ARRIVAL AT WAYPOINT and ESTIMATED TIME OF ARRIVAL AT PIM, and also of the methods ESTIMATED TIME ENROUTE TO WAYPOINT and ESTIMATED TIME ENROUTE TO PIM.
- The methods WAYPOINT GEOMETRY and ROUTE GEOMETRY perform similar tasks.

B. REFINEMENT OF THE OBJECTS AND CLASSES

1. Global Analysis of the Set of Objects and Classes

Every database should provide the user with the means to create, delete, maintain, and display objects. Therefore, the methods CREATE INSTANCE, DELETE INSTANCE, UPDATE INSTANCE, and DISPLAY INSTANCE were defined for all concrete classes (abstract classes do not need these methods as they have no instances). Also, to provide a user-friendly set of options, the method GET MENU was defined for all classes.

In order to enforce encapsulation, a separate method to access each variable's value was defined (e.g., GET "VARIABLE"). Also, a separate method to update each variable's value was defined (e.g., SET "VARIABLE"). The SET "VARIABLE" method is also responsible for checking if the value being updated is in the correct range.

In [Ref. 1], the variable TYPE was defined for the classes OWNERSHIP, TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK. However, the only information provided is that TYPE is a string of ten characters; nothing is said about its purpose. For this reason, this variable was not included in the set of variables defined for the classes above. Further research about the real world situation will be necessary before TYPE can be included as a meaningful variable. It could, however, be easily added at any time.

2. Individual Analysis of Objects and Classes

a. *Geographical Position*

The analysis of the object's variables step showed that the variable GEOGRAPHICAL POSITION has two components: latitude and longitude. Latitude and longitude, in turn, each have four components: degree, minute, second, and hemisphere (north/south or east/west, respectively).

Notice, however, that the components degree, minute, and second can also be used, in real world terms, to form another object (class): ANGLE. Therefore, latitude and longitude can both be represented by an angle plus a hemisphere. In fact, latitude and longitude are the geographical coordinates of a point.

Also notice that latitude is a particular case of geographical coordinate whose hemisphere can assume only the values *North* or *South* and whose angle is restricted to the range 0..90 degrees. Longitude, in turn, is also a particular case of geographical coordinate whose hemisphere can assume only the values *East* or *West* and whose angle is restricted to the range 0..180 degrees.

These classes were initially defined as follows:

CLASS: ANGLE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: DEGREE: (0..359) [0]
MINUTE: (0..59) [0]
SECOND: (0..59) [0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET DEGREE
SET DEGREE
GET MINUTE
SET MINUTE
GET SECOND
SET SECOND

CLASS: LATITUDE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: ANGLE: ANGLE [0:0:0]
HEMISPHERE: {N, S} [N]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET ANGLE

SET ANGLE
GET HEMISPHERE
SET HEMISPHERE

CLASS: LONGITUDE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: ANGLE: ANGLE [0:0:0]
HEMISPHERE: (E, W) [W]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET ANGLE
SET ANGLE
GET HEMISPHERE
SET HEMISPHERE

The methods SET ANGLE and SET HEMISPHERE are responsible for the checking if the values of the angles and hemispheres being assigned to LATITUDE and LONGITUDE are in the correct range.

Since LATITUDE and LONGITUDE are themselves objects, the variable GEOGRAPHICAL POSITION was defined as a composite object, as follows:

CLASS: GEOGRAPHICAL POSITION (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: LATITUDE: LATITUDE [0:0:0:M]
LONGITUDE: LONGITUDE [0:0:0:W]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET LATITUDE
SET LATITUDE
GET LONGITUDE
SET LONGITUDE

b. Relative Position

The variable RELATIVE POSITIVE has two components: bearing and range. Thus, the class RELATIVE POSITION was defined, having two variables: BEARING and RANGE. A bearing can be represented by an angle plus a reference north (true or magnetic). Since an angle plus a reference north can more generally be considered as representing a direction, the class DIRECTION was defined. BEARING was then defined as a variable of type DIRECTION. Since ANGLE is also an object, DIRECTION is a composite object. The class RELATIVE POSITION is also a composite object. The classes DIRECTION and RELATIVE POSITION were defined as follows:

CLASS: DIRECTION (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: ANGLE: ANGLE [0:0:0]
REFERENCE NORTH: {T, M} [T]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET ANGLE
SET ANGLE
GET REFERENCE NORTH
SET REFERENCE NORTH

CLASS: RELATIVE POSITION (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: BEARING: DIRECTION [0:0:0:T]
RANGE: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET BEARING
SET BEARING
GET RANGE
SET RANGE

c. Wind Direction and Wind Speed

The variables WIND DIRECTION and WIND SPEED can be combined to form the class WIND. Note that the variable WIND DIRECTION can be represented by an angle plus a reference north (true or magnetic). Since the class DIRECTION has an angle and a reference north as its component variables, WIND DIRECTION was defined as a variable of type DIRECTION. Also, since WIND DIRECTION and WIND SPEED were both defined for the class WIND, these two variables can simply be named, respectively, DIRECTION and SPEED. Therefore, the class WIND was defined as follows:

CLASS: WIND (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: DIRECTION: DIRECTION [0:0:0:T]
SPEED: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET DIRECTION
SET DIRECTION
GET SPEED
SET SPEED

d. Ownship Set and Magnetic Variation

The variable OWNSHIP SET can be represented by an angle plus a reference north (true or magnetic). Since the class DIRECTION has an angle and a reference north as its component variables, it was decided to define OWNSHIP SET as a variable of type DIRECTION.

The variable MAGNETIC VARIATION can be represented by an angle plus a hemisphere, with the restrictions that the hemisphere can only assume the values *East* or *West* and the angle is supposed to assume only small values (even though not mandatory). Since the class LONGITUDE is also represented by an angle (with values in the range 0..180) plus a hemisphere (with values *East* or *West*), MAGNETIC VARIATION could be defined as a variable of type LONGITUDE, in order to reuse existing code. However, to avoid problems of semantics, it was decided to define the class MAGNETIC VARIATION to represent this variable. MAGNETIC VARIATION was then defined as follows:

CLASS: MAGNETIC VARIATION (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: ANGLE: ANGLE [0:0:0]
HEMISPHERE: {E, W} [W]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE

GET ANGLE
SET ANGLE
GET HEMISPHERE
SET HEMISPHERE

e. Time of Position

The variable TIME OF POSITION has four simple components: hour, minute, second, and time zone. The components hour, minute, and second represent the local time of a place. These three components together can form the class: HMS. The class HMS plus a time zone can form another class: TIME. These two classes were defined as follows:

CLASS: HMS (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: HOUR: (00..23) [00]
MINUTE: (00..59) [00]
SECOND: (00..59) [00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET HOUR
SET HOUR
GET MINUTE
SET MINUTE
GET SECOND
SET SECOND

CLASS: TIME (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: LOCAL TIME: HMS [00:00:00]
TIME ZONE: (A..Z) [Z]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET LOCAL TIME
SET LOCAL TIME
GET TIME ZONE
SET TIME ZONE

f. Close CPA and Collision CPA

One of the requirements presented in [Ref. 1] is that when the CPA of a track meets specified time and range criteria, the user shall be notified of a Close CPA or Collision CPA, as appropriate. The system shall allow the user to manually adjust the criteria for Close and Collision CPA calculations and alert. Preset parameters for CPA calculations shall be requested from the user upon system initialization. The following are the default criteria for Close and Collision CPA alerts:

- Close CPA Range: 200..9999 yards.
- Close CPA Time: 5..99 minutes.
- Collision CPA Range: 1..199 yards.
- Collision CPA Time: 5..99 minutes.

Notice that the CPA criteria can be divided into Close CPA and Collision CPA. Close CPA can be further divided into distance range and time range. Similarly, Collision CPA can be further divided into distance range and time range. Distance range, in turn, can be further divided into minimum distance and maximum distance and, analogously, time range can be divided into minimum time and maximum time, with time being expressed in minutes. Notice that time can be better represented in the format hours:minutes:seconds, rather than just in minutes. Therefore, in order to describe the CPA criteria, the following classes were created:

CLASS: DISTANCE RANGE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: MINIMUM DISTANCE: real [0.0]
MAXIMUM DISTANCE: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET MINIMUM DISTANCE
SET MINIMUM DISTANCE
GET MAXIMUM DISTANCE
SET MAXIMUM DISTANCE

CLASS: TIME RANGE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: MINIMUM TIME: HMS [00:00:00]
MAXIMUM TIME: HMS [00:00:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET MINIMUM TIME
SET MINIMUM TIME
GET MAXIMUM TIME
SET MAXIMUM TIME

CLASS: CLOSE CPA (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: DISTANCE RANGE: DISTANCE RANGE [200:9999]
TIME RANGE: TIME RANGE [00:05:00:01:39:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET DISTANCE RANGE
SET DISTANCE RANGE
GET TIME RANGE
SET TIME RANGE

CLASS: COLLISION CPA (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: DISTANCE RANGE: DISTANCE RANGE [1:199]
TIME RANGE: TIME RANGE [00:05:00:01:39:00]

Methods: GET MENU
 CREATE INSTANCE
 DELETE INSTANCE
 UPDATE INSTANCE
 DISPLAY INSTANCE
 GET DISTANCE RANGE
 SET DISTANCE RANGE
 GET TIME RANGE
 SET TIME RANGE

g. Ownship

The OWNSHIP's variable ORIGIN has the value *Local Manual* for all instances of class OWNSHIP. For this reason, it is not necessary to repeat this variable's value *Local Manual* every time a new instance of OWNSHIP is created. ORIGIN could then be represented as a class variable. Another alternative would be to treat it as a constant and simply define the method GET ORIGIN which returns the constant value *Local Manual*.

As previously mentioned, deciding whether a variable with constant value should be defined as a class variable or as a method is a matter of design decision. Both alternatives work equally well. It is good practice in object-oriented programming to access the value of a variable only by means of a method (and not directly). Therefore, in general, methods to access and update a variable's value are necessary. For the class OWNSHIP, instead of defining ORIGIN as a class variable, it was decided to define the method GET ORIGIN which is used to return origin information.

Notice, however, that an instance variable in general has different values for different instances (objects) of a given class and, in this case, a method defined to display that variable's value would display its current value. The instance variable itself is

necessary to store the variable's value so that this value can be used in calculations during the program execution.

In the real world situation, the class OWNSHIP has only one instance, which is the ship that is operating the Tactical Database. However, it may be convenient to have the trajectory followed by ship available at times. One way to maintain this trajectory history is to store the position of the ship at different times. This can be done by assigning a number to the position of the ship every time a new geographical position and a new time of position are stored. This can be done by using the variable TRACK NUMBER. Since there is only one object (track), it is not necessary to store a number for it. Thus, the variable TRACK NUMBER is otherwise unused and can therefore be used to store an identifier (number) for each selected previous position of the ship.

In [Ref. 1], the variable TRACK HISTORY is specified for the class OWNSHIP. This variable is supposed to store a boolean value (true or false) to indicate if a record of previous positions of the ship is to be kept or not. However, it was decided to keep a record of Ownship's previous positions at all times. Consequently, the variable TRACK HISTORY would always have the constant value *True*. Thus, this variable became unnecessary. TRACK HISTORY was therefore eliminated and the method TRACK HISTORY PROCESSING was defined to provide track history information.

The advantage of using TRACK NUMBER to store the position numbers is that the variable TRACK NUMBER is also defined for the classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK and this similarity of variables is useful for the organization of the class hierarchy.

In order to store wind information (wind direction and speed), the variable WIND INFO was defined. Since the method CPA PROCESSING is defined for OWNSHIP, the variables CLOSE CPA and COLLISION CPA were also defined for this class, in order to store the criteria of Close and Collision CPAs.

After this refinement, the class OWNSHIP was defined as follows:

CLASS: OWNSHIP (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
TIME OF POSITION: TIME [00:00:00:Z]
GREENWICH MEAN TIME: TIME [00:00:00:Z]
MARKTIME: HMS [00:00:00]
JULIAN DATE: (0001..9366) [0001]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
MAGNETIC VARIATION: MAGNETIC VARIATION [0:0:0:W]
OWNSHIP SET: DIRECTION [0:0:0:T]
OWNSHIP DRIFT: real [0.0]
WIND INFO: WIND [0:0:0:T:0.0]
NAVIGATION SOURCE: {Inertial, SATNAV, Omega, DR} [DR]
CLOSE CPA : CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]
TRACK NUMBER: integer [0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
MONITOR OWNSHIP POSITION
NAVIGATIONAL COMPUTATION
CPA PROCESSING
OWNSHIP REPOSITIONING
DEAD RECKONING

BEARING AND RANGE FROM POSITION TO POSITION
TRACK HISTORY PROCESSING
GET ORIGIN
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET GREENWICH MEAN TIME
SET GREENWICH MEAN TIME
GET MARKTIME
SET MARKTIME
GET JULIAN DATE
SET JULIAN DATE
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET MAGNETIC VARIATION
SET MAGNETIC VARIATION
GET OWNERSHIP SET
SET OWNERSHIP SET
GET OWNERSHIP DRIFT
SET OWNERSHIP DRIFT
GET WIND INFO
SET WIND INFO
GET NAVIGATION SOURCE
SET NAVIGATION SOURCE
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA
GET TRACK NUMBER
SET TRACK NUMBER

h. Tentative Track

The variables CATEGORY and IDENTITY were specified (in [Ref. 1]) to have the values *Tentative* and *Unknown*, respectively, for all instances of this class. However, it would be useful to know if a tentative track is either in the air, or on the surface, or

underwater. Therefore, instead of assuming the value *Tentative*, the variable `CATEGORY` was allowed to assume the values *Tent-Air*, *Tent-Surface*, or *Tent-Subsurface*. `CATEGORY` was then defined as an instance variable. Instead of defining `IDENTITY` as a class variable, the method `GET IDENTITY` was defined to return the value *Unknown*, which is constant.

In a real world situation, information about a track (air, surface, or subsurface) is obtained by means of a tracking radar. A radar directly reads the relative position of the target being tracked with respect to its own position. When the information is received from a distant radar, a transformation of coordinates is necessary. Once the track position relative to Ownship is obtained, the track geographical position can be calculated.

Since the radar directly reads the relative position, the variable `RELATIVE POSITION` was kept and the variable `GEOGRAPHICAL POSITION` was replaced by a method, `CALCULATE GEOGRAPHICAL POSITION`, which calculates the geographical position of a track from its relative position with respect to Ownship (it is, of course, assumed that the Ownship's geographical position is known). Notice that, for encapsulation purposes, the method `CALCULATE GEOGRAPHICAL POSITION` could instead be named `GET GEOGRAPHICAL POSITION`. The user does not need to know whether the geographical position is calculated or stored directly in the database. However, in order to have this method's name as close as possible to its purpose, it was decided to name it `CALCULATE GEOGRAPHICAL POSITION`. Notice also that only one of the variables `RELATIVE POSITION` or `GEOGRAPHICAL POSITION` needs to be defined, since one can be calculated from the other.

The methods `TRACK POSITION DATA` and `TRACK TERMINATION` (both specified in [Ref. 1]) are used for displaying track data and deleting a track. They were renamed

DISPLAY INSTANCE and DELETE INSTANCE, respectively, because these last two methods were defined for all classes and similarities are desirable for the organization of the class hierarchy (note that if the user desires, these methods can easily be made available under both names).

The class TENTATIVE TRACK was therefore defined as follows:

CLASS: TENTATIVE TRACK (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: RELATIVE POSITION: RELATIVE POSITION [0:0:0:T:0.0]
TIME OF POSITION: TIME [00:00:00:Z]
CATEGORY: {Tent-Air, Tent-Surface, Tent-Subsurface} [*Tent-Air*]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
ORIGIN: {Local Manual, Local Auto} [*Local Manual*]
TRACK NUMBER: integer [0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
NEW TRACK ESTABLISHMENT
TRACK COURSE AND SPEED DETERMINATION
DEAD RECKONING
INITIAL CATEGORY ASSIGNMENT
INITIAL IDENTITY ASSIGNMENT
CATEGORY CHANGE PROCESSING
CALCULATE GEOGRAPHICAL POSITION
GET IDENTITY
GET RELATIVE POSITION
SET RELATIVE POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET CATEGORY

SET CATEGORY
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET ORIGIN
SET ORIGIN
GET TRACK NUMBER
SET TRACK NUMBER

i. Air Track, Surface Track, and Subsurface Track

These three classes were analyzed together due to their similarity to one another. CATEGORY has the value *Air* for all instances of class AIR TRACK, the value *Surface* for all instances of class SURFACE TRACK, and the value *Subsurface* for all instances of class SUBSURFACE TRACK. In order to return these values, the method GET CATEGORY was defined for each of these classes.

In order to maintain history positional data on all tracks, the variable POSITION NUMBER was defined. Therefore, every time a track (air, surface, or subsurface) position is updated, the old instance is saved and a new instance of this track is created with a new position number.

Again, for the purpose of obtaining a better similarity of methods, the methods TRACK POSITION UPDATE and MANUAL TERMINATION (specified in [Ref. 1]) were renamed, respectively, UPDATE INSTANCE and DELETE INSTANCE.

The geographical position is obtained via the method CALCULATE GEOGRAPHICAL POSITION, which uses the variable RELATIVE POSITION.

These three classes were, then, defined as follows:

CLASS: AIR TRACK (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: RELATIVE POSITION: RELATIVE POSITION [0:0:0:T:0.0]
TIME OF POSITION: TIME [00:00:00:Z]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
HEIGHT: real [0.0]
IDENTITY: {Unknown, Friendly, Hostile} [Unknown]
ORIGIN: {Local Manual, Local Auto, Remote} [Local Manual]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00.05:00:01:39:00]
TRACK NUMBER: integer [0]
POSITION NUMBER: integer [0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CPA PROCESSING
TRACK COURSE AND SPEED DETERMINATION
DEAD RECKONING
TRACK HISTORY PROCESSING
IDENTIFICATION FUNCTION
CALCULATE GEOGRAPHICAL POSITION
GET CATEGORY
GET RELATIVE POSITION
SET RELATIVE POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET HEIGHT
SET HEIGHT
GET IDENTITY
SET IDENTITY
GET ORIGIN
SET ORIGIN

GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA
GET TRACK NUMBER
SET TRACK NUMBER
GET POSITION NUMBER
SET POSITION NUMBER

CLASS: SURFACE TRACK (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: RELATIVE POSITION: RELATIVE POSITION [0:0:0:T:0.0]
TIME OF POSITION: TIME [00:00:00:Z]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
IDENTITY: {Unknown, Friendly, Hostile} [Unknown]
ORIGIN: {Local Manual, Local Auto, Remote} [Local Manual]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]
TRACK NUMBER: integer [0]
POSITION NUMBER: integer [0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CPA PROCESSING
TRACK COURSE AND SPEED DETERMINATION
DEAD RECKONING
TRACK HISTORY PROCESSING
IDENTIFICATION FUNCTION
CALCULATE GEOGRAPHICAL POSITION
GET CATEGORY
GET RELATIVE POSITION
SET RELATIVE POSITION
GET TIME OF POSITION
SET TIME OF POSITION

GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET IDENTITY
SET IDENTITY
GET ORIGIN
SET ORIGIN
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA
GET TRACK NUMBER
SET TRACK NUMBER
GET POSITION NUMBER
SET POSITION NUMBER

CLASS: SUBSURFACE TRACK (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: RELATIVE POSITION: RELATIVE POSITION [0:0:0:T:0.0]
TIME OF POSITION: TIME [00:00:00:Z]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
DEPTH: real [0.0]
IDENTITY: {Unknown, Friendly, Hostile} [Unknown]
ORIGIN: {Local Manual, Local Auto, Remote} [Local Manual]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]
TRACK NUMBER: integer [0]
POSITION NUMBER: integer [0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CPA PROCESSING
TRACK COURSE AND SPEED DETERMINATION

DEAD RECKONING
TRACK HISTORY PROCESSING
IDENTIFICATION FUNCTION
CALCULATE GEOGRAPHICAL POSITION
GET CATEGORY
GET RELATIVE POSITION
SET RELATIVE POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET DEPTH
SET DEPTH
GET IDENTITY
SET IDENTITY
GET ORIGIN
SET ORIGIN
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA
GET TRACK NUMBER
SET TRACK NUMBER
GET POSITION NUMBER
SET POSITION NUMBER

j. Waypoint

The variables IDENTITY, CATEGORY, and ORIGIN have the values *Waypoint*, *Special*, and *Local Manual*, respectively, for all instances of these classes. The methods GET IDENTITY, GET CATEGORY, and GET ORIGIN were then defined to return their respective constant values.

A waypoint represents each of the intermediate destination points used during the planning of a route. Each waypoint is, in fact, a geographical position in the planning

chart. Therefore, it is convenient to have this geographical position stored directly in the Tactical Database. GEOGRAPHICAL POSITION is then defined as an instance variable.

The variable STEAMING ROUTE indicates the routes to which the waypoint belongs. According to [Ref. 1], the Tactical Database must allow for the specification of up to six steaming routes, each of which may have up to 50 waypoints. Since a waypoint can belong to more than one route, the type of the variable STEAMING ROUTE was defined as a subset (either proper or improper) of the set {1,2,3,4,5,6}.

The method WAYPOINT GEOMETRY was eliminated because the method ROUTE GEOMETRY (defined for the class ROUTE) performs the same task and has a name that is semantically more significant.

The relative position of each waypoint with respect to Ownship can be calculated once the geographical positions of the waypoints and of Ownship are known. The method CALCULATE RELATIVE POSITION is used to accomplish this task. Notice that, for encapsulation purposes, this method could have been named GET RELATIVE POSITION, since the user does not need to know whether the relative position is calculated or stored directly in the database. However, in order to have this method's name as close as possible to its purpose, it was named CALCULATE RELATIVE POSITION.

The methods OWNSHIP DISTANCE TO WAYPOINT, ESTIMATED TIME OF ARRIVAL AT WAYPOINT, and ESTIMATED TIME ENROUTE TO WAYPOINT were renamed, respectively, OWNSHIP DISTANCE TO POSITION, ESTIMATED TIME OF ARRIVAL AT POSITION, and ESTIMATED TIME ENROUTE TO POSITION. Consequently, these methods can now be reused for other classes.

The class WAYPOINT was then defined as follows:

CLASS: WAYPOINT (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
TIME OF POSITION: TIME [00:00:00:Z]
STEAMING ROUTE: subset of {1,2,3,4,5,6} [{]}

Methods:

- GET MENU
- CREATE INSTANCE
- DELETE INSTANCE
- UPDATE INSTANCE
- DISPLAY INSTANCE
- CALCULATE RELATIVE POSITION
- OWNSHIP DISTANCE TO POSITION
- ESTIMATED TIME OF ARRIVAL AT POSITION
- ESTIMATED TIME ENROUTE TO POSITION
- GET IDENTITY
- GET CATEGORY
- GET ORIGIN
- GET GEOGRAPHICAL POSITION
- SET GEOGRAPHICAL POSITION
- GET TIME OF POSITION
- SET TIME OF POSITION
- GET STEAMING ROUTE
- SET STEAMING ROUTE

k. Reference Point, Navigation Hazard, and Man in Water

These three classes have identical variables and similar methods, and are therefore analyzed together.

The variable CATEGORY has the value *Special* for all instances of these classes. Also, the variable IDENTITY has the value *Refpoint* for all instances of class REFERENCE POINT, *Navhaz* for all instances of class NAVIGATION HAZARD, and *Man in water* for all instances of class MAN IN WATER. Consequently, the methods GET IDENTITY and GET CATEGORY were defined to return the appropriate constant value.

The variable ORIGIN was specified in [Ref. 1] to assume only the value *Local Manual* for these three classes. However, it is also important to receive information about these classes from a remote source. Therefore, ORIGIN will be allowed to assume the values *Local Manual* and *Remote*.

A reference point and a navigation hazard are, in general, points whose geographical positions are known. Therefore, their relative positions with respect to Ownship can be calculated by means of the method CALCULATE RELATIVE POSITION.

When a situation of men in water occurs (airplane crash, for example), the information transmitted by the radar is their relative position. Their geographical position can then be calculated. However, if the information about men in water is received from a remote station, the information received is their relative position with respect to a data link reference point. Their geographical position can also be calculated. In order to obtain similarity with the classes REFERENCE POINT and NAVIGATION HAZARD, the variable RELATIVE POSITION was eliminated, the variable GEOGRAPHICAL POSITION was kept, and the method CALCULATE RELATIVE POSITION was added to the definition of the class MAN IN WATER.

The variables COURSE and SPEED are defined for the classes NAVIGATION HAZARD and MAN IN WATER to account for eventual displacements due to ocean currents, for example. Even though displacements such as these are, in general, negligible in comparison to the characteristic speeds involved in the LCCDS scenario, the variables COURSE and SPEED were defined for completeness.

Each of the methods ENTER/UPDATE MAN IN WATER, ENTER/UPDATE NAVIGATION HAZARD, and ENTER/UPDATE REFERENCE POINT (which were specified in [Ref. 1]) were replaced by two methods: CREATE INSTANCE and UPDATE INSTANCE. This was done to keep as much commonality as possible among methods in this group of classes.

After these refinements, these classes were defined as follows:

CLASS: REFERENCE POINT (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
TIME OF POSITION: TIME [00:00:00:Z]
ORIGIN: {Local Manual, Remote} [Local Manual]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CPA PROCESSING
CALCULATE RELATIVE POSITION
GET IDENTITY

GET CATEGORY
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET ORIGIN
SET ORIGIN
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA

CLASS: NAVIGATION HAZARD (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:M]
TIME OF POSITION: TIME [00:00:00:Z]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
ORIGIN: {Local Manual, Remote} [*Local Manual*]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CPA PROCESSING
CALCULATE RELATIVE POSITION
GET IDENTITY
GET CATEGORY
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET COURSE
SET COURSE
GET SPEED

SET SPEED
GET ORIGIN
SET ORIGIN
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA

CLASS: MAN IN WATER (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:M]
TIME OF POSITION: TIME [00:00:00:Z]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
ORIGIN: {Local Manual, Remote} [*Local Manual*]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CPA PROCESSING
CALCULATE RELATIVE POSITION
GET IDENTITY
GET CATEGORY
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET ORIGIN
SET ORIGIN
GET CLOSE CPA

SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA

1. Data Link Reference Point

The variables CATEGORY and IDENTITY have the values *Special* and *Dlrp*, respectively, for all instances of this class. Therefore, the methods GET CATEGORY and GET IDENTITY were defined to return these values.

The variable ORIGIN was allowed to assume not only the value *Local Manual*, but also the value *Remote*.

A data link reference point is essentially a fixed geographical position used as a reference. For this reason, it is necessary to store its value directly in the Tactical Database. The variable GEOGRAPHICAL POSITION has been added for this purpose.

All position information received by Ownship from remote units is with respect to a data link reference point. The position information with respect to Ownship is obtained by means of a transformation of coordinates. For this transformation of coordinates, the relative position of Ownship with respect to the given data link reference point is necessary; the method CALCULATE RELATIVE POSITION provides this capability.

Once again, trying to achieve as high of degree as possible of commonalities among the methods and/or variables of classes, the methods ENTER DLRP and UPDATE DLRP (both specified in [Ref. 1]) were renamed CREATE INSTANCE and UPDATE INSTANCE, respectively.

The class DATA LINK REFERENCE POINT was then defined as follows:

CLASS: DATA LINK REFERENCE POINT (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
TIME OF POSITION: TIME [00:00:00:Z]
ORIGIN: {Local Manual, Remote} [*Local Manual*]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CPA PROCESSING
CALCULATE RELATIVE POSITION
GET IDENTITY
GET CATEGORY
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET ORIGIN
SET ORIGIN
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA

m. Formation Center and Position And Intended Movement

The variable CATEGORY has the value *Special* for all instances of these two classes. Also, the variable IDENTITY has the value *Fc* for all instances of class FORMATION CENTER and the value *Pim* for all instances of class POSITION AND

INTENDED MOVEMENT. Therefore, the methods GET CATEGORY and GET IDENTITY were defined to return the appropriate constant value.

The variable ORIGIN was allowed to assume the values *Local Manual* and *Remote* for the instances of the class FORMATION CENTER and only the value *Local Manual* for all instances of the class POSITION AND INTENDED MOVEMENT. Therefore, the method GET ORIGIN was defined only for the class POSITION AND INTENDED MOVEMENT in order to return the value *Local Manual*.

An object of class FORMATION CENTER or of class POSITION AND INTENDED MOVEMENT represents a geographical position. Thus, it is necessary to define the variable GEOGRAPHICAL POSITION for both classes in order to have this information promptly available in the Tactical Database. The relative position is calculated by the method CALCULATE RELATIVE POSITION.

In order to obtain a better similarity of methods, the methods OWNERSHIP DISTANCE TO PIM, ESTIMATED TIME OF ARRIVAL AT PIM, and ESTIMATED TIME ENROUTE TO PIM were renamed, respectively, OWNERSHIP DISTANCE TO POSITION, ESTIMATED TIME OF ARRIVAL AT POSITION, and ESTIMATED TIME ENROUTE TO POSITION.

The methods for creation, maintenance, and display of objects (i.e., GET MENU, CREATE INSTANCE, DELETE INSTANCE, UPDATE INSTANCE, and DISPLAY INSTANCE) were also added to the definition of these two classes, which were defined as follows:

CLASS: FORMATION CENTER (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:M]
TIME OF POSITION: TIME [00:00:00:Z]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
ORIGIN: {Local Manual, Remote} [Remote]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
DEAD RECKONING
CPA PROCESSING
CALCULATE RELATIVE POSITION
GET IDENTITY
GET CATEGORY
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET ORIGIN
SET ORIGIN
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA

CLASS: POSITION AND INTENDED MOVEMENT (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
TIME OF POSITION: TIME [00:00:00:Z]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
CLOSE CPA : CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]

Methods:

- GET MENU
- CREATE INSTANCE
- DELETE INSTANCE
- UPDATE INSTANCE
- DISPLAY INSTANCE
- DEAD RECKONING
- CPA PROCESSING
- OWNSHIP DISTANCE TO POSITION
- ESTIMATED TIME OF ARRIVAL AT POSITION
- ESTIMATED TIME ENROUTE TO POSITION
- CALCULATE RELATIVE POSITION
- GET IDENTITY
- GET CATEGORY
- GET ORIGIN
- GET GEOGRAPHICAL POSITION
- SET GEOGRAPHICAL POSITION
- GET TIME OF POSITION
- SET TIME OF POSITION
- GET COURSE
- SET COURSE
- GET SPEED
- SET SPEED
- GET CLOSE CPA
- SET CLOSE CPA
- GET COLLISION CPA
- SET COLLISION CPA

n. Area

An area can be described by the geographical coordinates of its geometric center and its north, south, east, and west limits. Therefore, in order to keep the semantics as close as possible to the real world, the variable GEOGRAPHICAL POSITION was renamed CENTER GEO POSITION. Since the center relative position can be calculated from the variable CENTER GEO POSITION, the variable RELATIVE POSITION was eliminated and the method CALCULATE RELATIVE POSITION was defined. The class AREA was then defined as follows:

CLASS: AREA (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: CENTER GEO POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
TIME OF POSITION: TIME [00:00:00:Z]
NORTH LIMIT: GEOGRAPHICAL POSITION [0:0:0:N]
SOUTH LIMIT: GEOGRAPHICAL POSITION [0:0:0:S]
EAST LIMIT: GEOGRAPHICAL POSITION [0:0:0:E]
WEST LIMIT: GEOGRAPHICAL POSITION [0:0:0:W]
IDENTITY: {Hot, Protected, Unprotected} [*Unprotected*]
ORIGIN: {Local Manual, Local Auto, Remote} [*Local Manual*]
CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
AREA NUMBER: integer [0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
DRAW TDB AREA

CPA PROCESSING
DEAD RECKONING
CALCULATE RELATIVE POSITION
GET CENTER GEO POSITION
SET CENTER GEO POSITION
GET TIME OF POSITION
SET TIME OF POSITION
GET NORTH LIMIT
SET NORTH LIMIT
GET SOUTH LIMIT
SET SOUTH LIMIT
GET EAST LIMIT
SET EAST LIMIT
GET WEST LIMIT
SET WEST LIMIT
GET IDENTITY
SET IDENTITY
GET ORIGIN
SET ORIGIN
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET AREA NUMBER
SET AREA NUMBER

o. Route

Since the variables START POINT and END POINT can also be considered as being waypoints, these variables were eliminated and the meaning of the variable WAYPOINTS was extended to include these two points. Thus, the class ROUTE was defined as follows:

CLASS: ROUTE (Concrete)

Superclasses: To be defined in Section C.2 and C.3

Class Variables: None

Instance Variables: ROUTE NUMBER: (1..6) [1]
WAYPOINTS: array 1..50 of WAYPOINT [0:0:0:N:00:00:00:Z:{1}]
IDENTITY: {Hot, Protected, Unprotected} [Unprotected]
ORIGIN: {Local Manual, Remote} [Local Manual]

Methods:

- GET MENU
- CREATE INSTANCE
- DELETE INSTANCE
- UPDATE INSTANCE
- DISPLAY INSTANCE
- ROUTE GEOMETRY
- GET ROUTE NUMBER
- SET ROUTE NUMBER
- GET WAYPOINTS
- SET WAYPOINTS
- GET IDENTITY
- SET IDENTITY
- GET ORIGIN
- SET ORIGIN

p. Userdefined

Due to its peculiar nature, this class has no similarities with the other classes. In order to allow the user to create a new class, the following methods were defined: CREATE NEW CLASS, DEFINE NEW VARIABLE, and DEFINE NEW METHOD. It was decided that no variables would be provided, giving the user both the freedom and the responsibility to define the necessary variables and/or methods.

The class USERDEFINED was then defined as follows:

CLASS: USERDEFINED (Concrete)

Superclasses: To be defined in Sections C.2 and C.3

Class Variables: None

Instance Variables: None

Methods:

- GET MENU
- CREATE INSTANCE
- DELETE INSTANCE
- UPDATE INSTANCE
- DISPLAY INSTANCE
- CREATE NEW CLASS
- DEFINE NEW VARIABLE
- DEFINE NEW METHOD

C. ORGANIZATION OF THE CLASSES IN A HIERARCHY

1. Analysis of the Implementation Language

Ross [Ref. 24], after analyzing three OODBMSs, namely Vbase [Ref.20], Gemstone [Ref. 25], and Iris [Ref. 26], concluded that the system that best fits the requirements of the LCCDS Project is the Gemstone Database Management System, a product of Servio Logic Corporation, Alameda, CA.

Assuming that Gemstone is the system to be used in the LCCDS Project, the following questions about the system can be answered:

- Does the system provide single, multiple, or selective inheritance? In Gemstone, each class, with the exception of the root, has a unique superclass. Therefore, only single inheritance is allowed.
- Can methods inherited from a superclass be overridden (redefined) in the subclasses? Yes.

- Can variables inherited from a superclass be overridden (redefined) in the subclasses? Yes, but the constraint on an inherited variable must be consistent with the constraint in the superclass. That is, the constraint on an inherited variable must be the same as, or a subset of, the constraint in the superclass.
- How is the schema evolution handled within the system? Gemstone approaches the schema evolution by means of a conversion mechanism ("pay me now"). Modifications do not need to be recompiled (i.e, classes can be modified and new classes can be created at run-time).
- Does the system offer early (compile-time) binding and/or late (run-time) binding? Gemstone is not strongly typed. This allows variables to be assigned values of different types at different points of execution, thus achieving late (dynamic) binding.

2. Construction of the Hierarchies

As previously mentioned, the similarities of methods and/or variables among classes is the fundamental factor for the construction of the class hierarchies.

However, it is important to keep in mind that the hierarchies constructed should represent the real world (problem space) as closely as possible. The resulting hierarchy (or hierarchies) should not be just a number of similar classes organized in a meaningless manner.

In the design process, the similarities of methods and/or variables among classes were listed in the steps Analysis of the Object's Variables and Analysis of the Object's Methods. Additionally, new methods and variables were defined during the step Refinement of the Objects and Classes. The resulting set of classes can then be organized into one or more hierarchies.

It was observed that, except for the class USERDEFINED, each of the classes defined for the Tactical Database represents either a point (fixed or nonfixed position),

a line (remember that a route can be represented by a line connecting the starting point to the end point), or an area (region) in the LCCDS scenario. The classes can thus be initially divided into two groups: those that are predefined and those that are defined by the user at run-time. The predefined classes can be further divided into three groups: those that represent points, those that represents lines, and those that represent areas. Five classes were, therefore, defined: PREDEFINED OBJECT, USERDEFINED OBJECT, TDB POINT (Tactical Database Point), TDB LINE (Tactical Database Line), and TDB AREA (Tactical Database Area). Considering that all classes represent real world objects, the root class of the Tactical Database hierarchy was named TDB OBJECT (Tactical Database Object) and the classes PREDEFINED OBJECT and USERDEFINED OBJECT were defined as its subclasses. The classes TDB POINT, TDB LINE, and TDB AREA, in turn, were defined as subclasses of the class PREDEFINED OBJECT. This initial class hierarchy is shown in Figure 3.

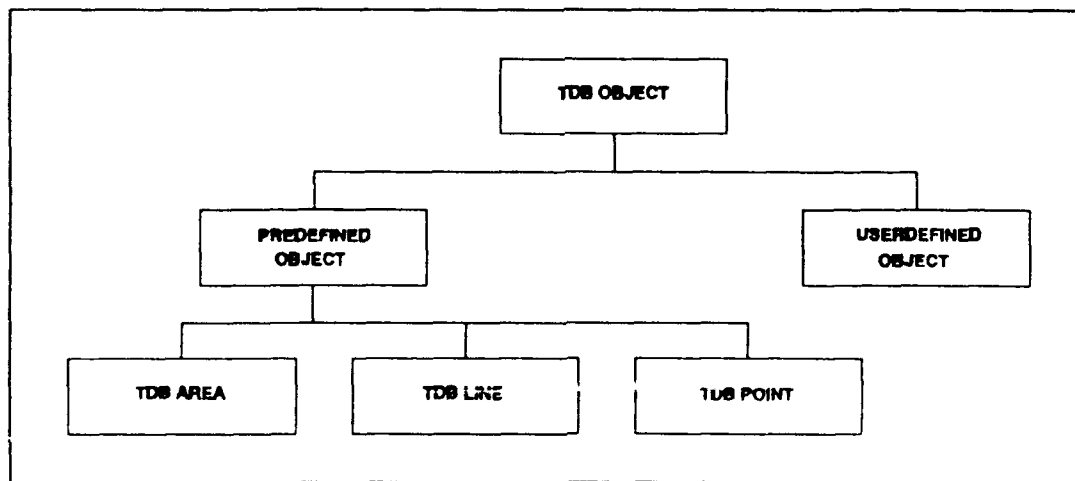


Figure 3: TACTICAL DATABASE Initial Class Hierarchy

The analysis of the commonalities among the classes AIR TRACK, SURFACE TRACK, SUBSURFACE TRACK, and TENTATIVE TRACK showed that all these classes could be defined as subclasses of a class TRACK.

While the classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK represent, respectively, objects that are in the air, on the surface, or underwater, the class TENTATIVE TRACK represents an object which can be either in the air, or on the surface, or underwater. Objects of classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK are considered firm tracks because their positions can be updated by the system in regular intervals of time. On the other hand, an object of class TENTATIVE TRACK only becomes a firm track after a valid course and speed established for it.

In terms of the real world situation, it is meaningful then to create the class TRACK with two subclasses: FIRM TRACK and TENTATIVE TRACK. The class FIRM TRACK, in turn, is divided into three subclasses: AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK.

Note also that the class OWNERSHIP, even though not representing an object that is being tracked by the ship's sensors, can be considered as a particular case of the class SURFACE TRACK for which the relative position is equal to zero (remember that the relative position is calculated with respect to Ownship). The class OWNERSHIP was, therefore, defined as a subclass of the class SURFACE TRACK.

Thus, considering the class TRACK as the root, the hierarchy of classes shown in Figure 4 was defined:

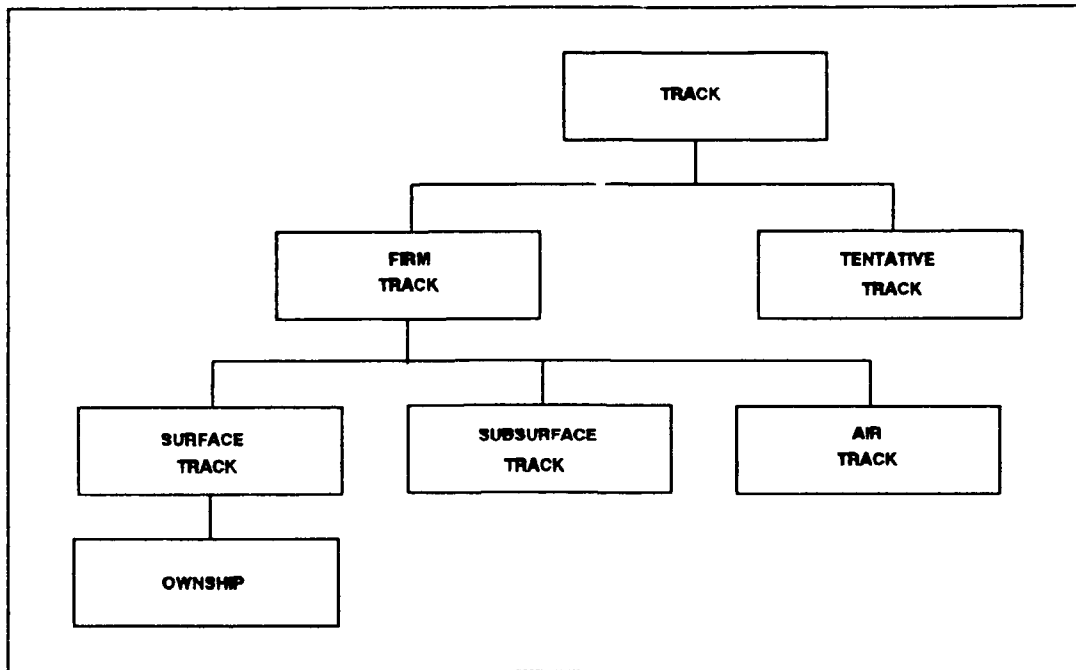


Figure 4: TRACK Class Hierarchy

The analysis of the commonalities among the classes REFERENCE POINT, DATA LINK REFERENCE POINT, WAYPOINT, FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT showed that these classes can be defined as subclasses of a class SPECIAL POINT. However, the variables COURSE and SPEED were not defined for the classes REFERENCE POINT, WAYPOINT, and DATA LINK REFERENCE POINT, as these are fixed points. Based on this difference, these classes can be divided into two groups: fixed and nonfixed special points.

Therefore, two subclasses of the class SPECIAL POINT were created: FIXED SPECIAL POINT and NONFIXED SPECIAL POINT. The classes REFERENCE POINT, DATA LINK REFERENCE POINT, and WAYPOINT were defined as subclasses of

FIXED SPECIAL POINT and the classes FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT were defined as subclasses of NONFIXED SPECIAL POINT. Therefore, considering SPECIAL POINT as the root, the hierarchy of classes shown in Figure 5 was defined.

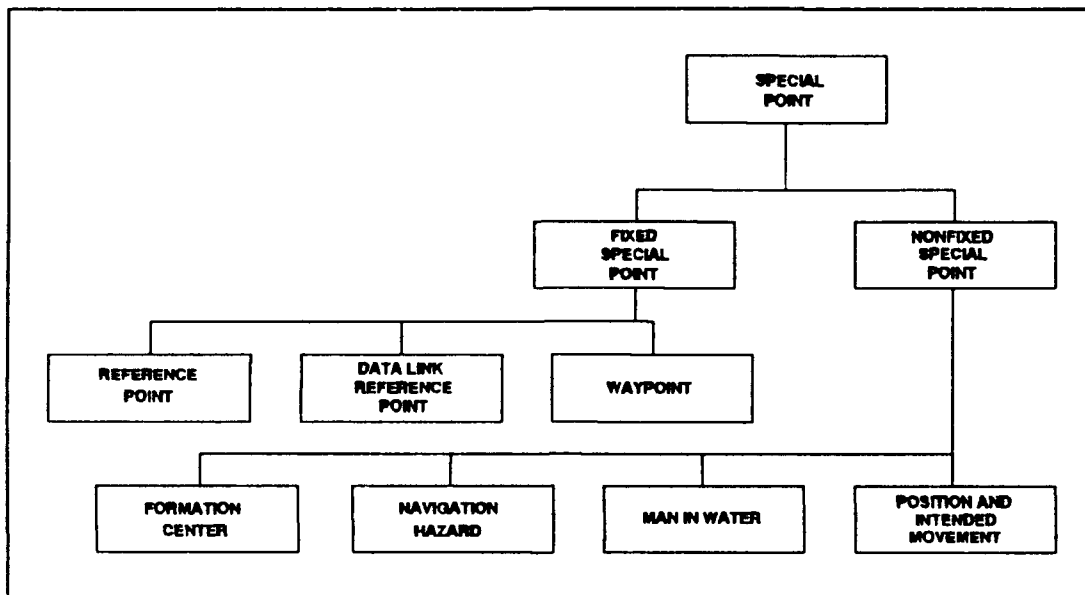


Figure 5: SPECIAL POINT Class Hierarchy

Since the classes SPECIAL POINT and TRACK represent points in the LCCDS scenario, the two hierarchies corresponding to these classes were combined by defining them as subclasses of the class TDB POINT. The portion of the class hierarchy with TDB POINT as the root is shown in Figure 6.

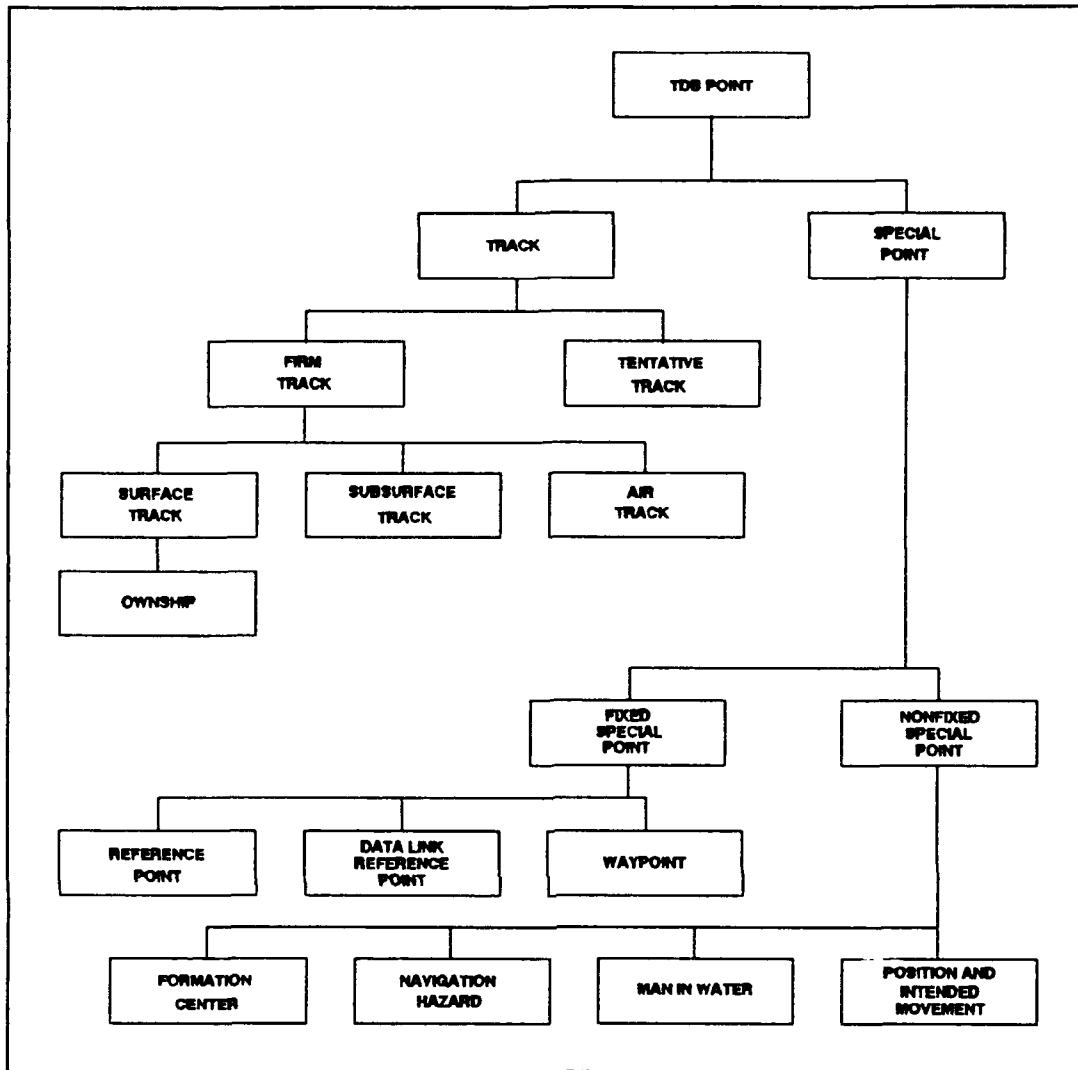


Figure 6: TDB POINT Class Hierarchy

Since the class ROUTE represents a line (or a set of lines), it was defined as a subclass of the class TDB LINE. No subclasses were defined for the class TDB AREA.

With the specification of all descendant classes of the classes TDB POINT, TDB LINE, and TDB AREA, the entire hierarchy was completed. This hierarchy is shown in Figure 7.

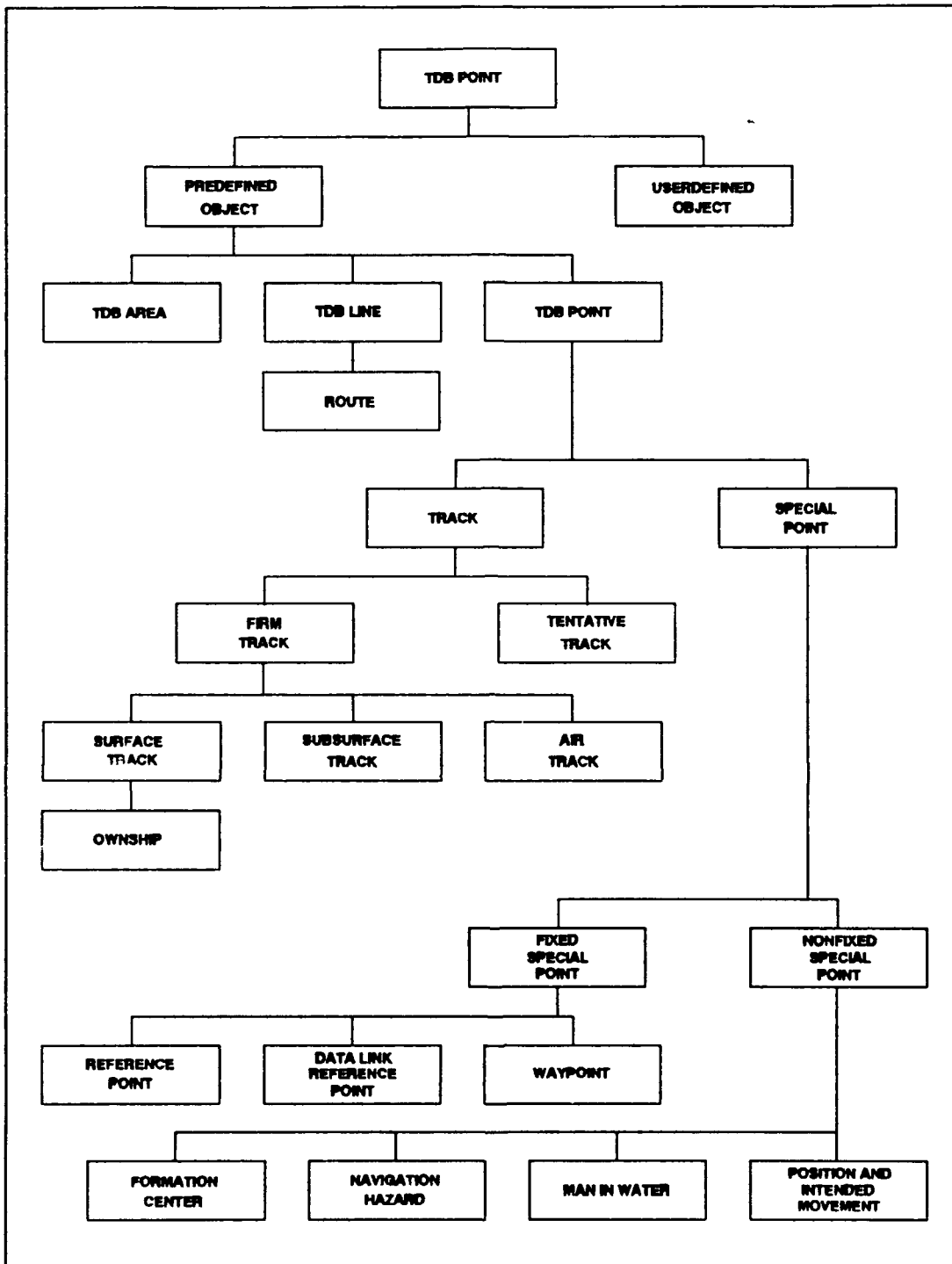


Figure 7: TACTICAL DATABASE Class Hierarchy

Notice that, for the creation of the class hierarchy, nine abstract classes were defined: TDB OBJECT, PREDEFINED OBJECT, TDB POINT, TDB LINE, TRACK, FIRM TRACK, SPECIAL POINT, FIXED SPECIAL POINT, and NONFIXED SPECIAL POINT.

The set of concrete classes consists of the classes ANGLE, LATITUDE, LONGITUDE, GEOGRAPHICAL POSITION, DIRECTION, RELATIVE POSITION, WIND, MAGNETIC VARIATION, HMS, TIME, DISTANCE RANGE, TIME RANGE, CLOSE CPA, and COLLISION CPA, which are used as variables (i.e., component classes of the composite classes), and the classes TDB AREA, ROUTE, USERDEFINED OBJECT, TENTATIVE TRACK, SURFACE TRACK, SUBSURFACE TRACK, AIR TRACK, OWNERSHIP, REFERENCE POINT, DATA LINK REFERENCE POINT, WAYPOINT, FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT, which represent the main objects of interest in the Tactical Database.

3. Review of the Classes' Variables and Methods

As the class hierarchy was constructed, the fact that the system allows only single inheritance was taken into consideration. In order to fit nicely into the class hierarchy, some classes had to be modified slightly. These modifications are described below.

a. TDB Object

The class TDB OBJECT was defined as the root of the class hierarchy of the LCCDS. Therefore, TDB OBJECT is an ancestor of each of the classes that form the Tactical Database.

In order to identify uniquely every instance (object) of every class, the instance variable TDB OBJECT NUMBER was defined for this class.

The only variable that is common to all classes is ORIGIN. This variable was then defined for the class TDB OBJECT, so that every class can inherit it. Also, the method GET MENU is common to all classes. It was therefore defined for this class.

One of the requirements listed in [Ref. 1] is that the user should be allowed to assign additional information to any type of track. By defining the variable ADDITIONAL INFORMATION for the class TDB OBJECT, this requirement was extended in such a way that the user is allowed to assign additional information to any object (instance) in the Tactical Database.

Since the type of information that is to be stored in the variable ADDITIONAL INFORMATION is not known, it was decided that this variable should be of type string, because numeric data can also be stored as strings. However, numeric data stored as strings cannot be used in calculations. Thus, the methods CONVERT STRING TO NUMERIC and CONVERT NUMERIC TO STRING were defined in order to allow the user to make type conversions.

Notice that the definition of ADDITIONAL INFORMATION as a variable of type string (of length 30, for example), might cause a problem. This variable, in most cases, will not be used, but will still occupy storage space. If the number of objects is large, which is the case of the Tactical Database, a considerable amount of memory (both primary and secondary) will be wasted.

One possible solution to this problem can be achieved by defining the class **ADDITIONAL INFO** and then defining **ADDITIONAL INFORMATION** as a variable of type **ADDITIONAL INFO**, as follows:

CLASS: ADDITIONAL INFO

Superclasses: None

Class Variables: None

Instance Variables: **INFORMATION LENGTH:** integer [0]
INFORMATION: string(information length) [null]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET INFORMATION LENGTH
SET INFORMATION LENGTH
GET INFORMATION
SET INFORMATION

The definition of the variable **ADDITIONAL INFORMATION** as a variable of type **ADDITIONAL INFO**, allows the user to specify the length of the string necessary to store the information. However, this solution can only be implemented if the OODBMS allows late (dynamic) binding, since the length of the string will only be known at run-time. GemStone allows late binding, but in order to keep the design of the Tactical Database as independent as possible of the OODBMS used, it was decided not to adopt this solution at this time.

Another possible solution involves the concept of subobject. A **subobject** is an object (class) that, even though being a component of a composite object, can exist independently. In general, the component objects of a composite object are dependent objects. A **dependent object** cannot be created (or accessed) unless the composite object to which it belongs already exists (i.e., its existence depends on the existence of the composite object). If **ADDITIONAL INFO** were defined as a subobject (i.e., defined independently of any other object), then the type of the variable **ADDITIONAL INFORMATION** could be defined as a pointer to one of the instances of **ADDITIONAL INFO**. If, for example, the composite object to which **ADDITIONAL INFORMATION** belongs were brought into main memory, only the pointer to the instance of **ADDITIONAL INFO** would be brought into the main memory as part of the object. The instance containing the additional information would remain stored in another location. This mechanism would avoid the problem of wasting storage space. However, since not all OODBMSs allow the creation of subobjects, it was also decided not to adopt this solution at this time.

Therefore, it will be up to the implementor to decide which is the best approach to this problem. For now, the variable **ADDITIONAL INFORMATION** is simply defined as a string of length 30.

Thus, the class **TDB OBJECT** was defined as follows:

CLASS: TDB OBJECT (Abstract)

Superclasses: None

Class Variables: None

Instance Variables: TDB OBJECT NUMBER: integer [0]
ORIGIN: {Local Manual, Local Auto, Remote} [*Local Manual*]
ADDITIONAL INFORMATION: string(30) [*None*]

Methods: GET MENU
CONVERT STRING TO NUMERIC
CONVERT NUMERIC TO STRING
GET TDB OBJECT NUMBER
SET TDB OBJECT NUMBER
GET ORIGIN
SET ORIGIN
GET ADDITIONAL INFORMATION
SET ADDITIONAL INFORMATION

b. Userdefined Object

Considering that the instantiation of the objects of the class USERDEFINED OBJECT depends on the decisions made by the user at run-time, this class was placed in the class hierarchy as a subclass of the class TDB OBJECT, so that it does not depend on other classes, except the root class. For this class, no variables were defined locally.

Thus, the class USERDEFINED OBJECT was defined as follows:

CLASS: USERDEFINED OBJECT (Concrete)

Superclasses: TDB OBJECT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CREATE NEW CLASS

DEFINE NEW VARIABLE
DEFINE NEW METHOD

c. Predefined Object

The method CPA PROCESSING and the variables CLOSE CPA and COLLISION CPA were defined for all classes of the Tactical Database hierarchy, except TENTATIVE TRACK, USERDEFINED OBJECT, ROUTE, and WAYPOINT. Adding them to the definition of the classes TENTATIVE TRACK, ROUTE, and WAYPOINT, makes CPA PROCESSING, CLOSE CPA, and COLLISION CPA common to all subclasses of the class PREDEFINED OBJECT. Therefore, the method CPA PROCESSING and the variables CLOSE CPA and COLLISION CPA can be defined only for this class, rather than having to be defined separately for each of the classes for which they were previously defined. Thus, the class PREDEFINED OBJECT was defined as follows:

CLASS: PREDEFINED OBJECT (Abstract)

Superclasses: TDB OBJECT

Class Variables: None

Instance Variables: CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]

Methods: GET MENU
CPA PROCESSING
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA

d. TDB Area

The only modification to this class was the elimination of the instance variable AREA NUMBER, which was replaced by the inherited variable TDB OBJECT NUMBER. Note that the method SET AREA NUMBER could easily be added to the system if the user prefers to have this information available under the originally specified variable name. The name of the method does not have to match the name of the variable that is actually used. Also note that, in general, this same approach can be taken for any other variable in the system which has been renamed or redefined.

The class TDB AREA was then defined as follows:

CLASS: TDB AREA (Concrete)

Superclasses: PREDEFINED OBJECT

Class Variables: None

Instance Variables: CENTER GEO POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
TIME OF POSITION: TIME [00:00:00:Z]
NORTH LIMIT: GEOGRAPHICAL POSITION [0:0:0:N]
SOUTH LIMIT: GEOGRAPHICAL POSITION [0:0:0:S]
EAST LIMIT: GEOGRAPHICAL POSITION [0:0:0:E]
WEST LIMIT: GEOGRAPHICAL POSITION [0:0:0:W]
IDENTITY: {Hot, Protected, Unprotected} [Unprotected]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
DEAD RECKONING
DRAW TDB AREA
GET CENTER GEO POSITION

SET CENTER GEO POSITION
GET NORTH LIMIT
SET NORTH LIMIT
GET SOUTH LIMIT
SET SOUTH LIMIT
GET EAST LIMIT
SET EAST LIMIT
GET WEST LIMIT
SET WEST LIMIT
GET IDENTITY
SET IDENTITY
GET COURSE
SET COURSE
GET SPEED
SET SPEED

e. TDB Line

In order to provide means of displaying the TDB lines, the method DRAW TDB LINE was defined for this class. Therefore, TDB LINE was defined as follows:

CLASS: TDB LINE (Abstract)

Superclasses: PREDEFINED OBJECT

Class Variables: None

Instance Variables: None

Methods: GET MENU
DRAW TDB LINE

f. Route

For this class, the variable ORIGIN could only assume the values *Local Manual* and *Remote*. However, instead of redefining locally the variable ORIGIN that was inherited

from the class TDB OBJECT, which allows also the value *Local Auto*, it was decided to allow this variable to assume also the value *Local Auto* for this class. This decision can be explained by the fact that future versions of the LCCDS could provide "Optimum Route Planning", which would calculate the optimum route from one point to another, taking into consideration factors such as necessity to avoid detection, position and plans of other battle group units, rapidly changing intelligence information, and satellite and aircraft reconnaissance coverage. Once the optimum route was calculated, it could be automatically stored in the Tactical Database.

The method DRAW ROUTE can be defined using the method DRAW LINE, which was inherited from the class TDB LINE.

Therefore, the class ROUTE was defined as follows:

CLASS: ROUTE (Concrete)

Superclasses: TDB LINE

Class Variables: None

Instance Variables: ROUTE NUMBER: (1..6) [1]
WAYPOINTS: array 1..50 of WAYPOINT [0:0:0:N:00:00:00:Z:{1}]
IDENTITY: {Hot, Protected, Unprotected} [Unprotected]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
DRAW ROUTE
ROUTE GEOMETRY
GET ROUTE NUMBER
SET ROUTE NUMBER
GET WAYPOINTS

SET WAYPOINTS
GET IDENTITY
SET IDENTITY

g. TDB Point

The class TDB POINT was defined as the superclass of the classes TRACK and SPECIAL POINT, because these two classes represent points in the LCCDS scenario. The only methods defined locally for TDB POINT are PLOT TDB POINT and GET MENU (which was redefined). Since the variable TIME OF POSITION is common to all subclasses of the class TDB POINT, TIME OF POSITION was defined for this class. Thus, TDB POINT was defined as follows:

CLASS: TDB POINT (Abstract)

Superclasses: TDB OBJECT

Class Variables: None

Instance Variables: TIME OF POSITION: TIME [00:00:00:Z]

Methods:
GET MENU
PLOT TDB POINT
GET TIME OF POSITION
SET TIME OF POSITION

h. Track

The class TRACK was defined as the superclass of the classes TENTATIVE TRACK and FIRM TRACK. The class FIRM TRACK, in turn, was defined as the superclass of the classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK.

All methods and variables that were common to these classes were then defined for the class TRACK. This class was then defined as follows:

CLASS: TRACK (Abstract)

Superclasses: TDB POINT

Class Variables: None

Instance Variables: RELATIVE POSITION: RELATIVE POSITION [0:0:0:T:0.0]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
IDENTITY: {Unknown, Friendly, Hostile} [Unknown]

Methods: GET MENU
CALCULATE GEOGRAPHICAL POSITION
TRACK COURSE AND SPEED DETERMINATION
DEAD RECKONING
GET RELATIVE POSITION
SET RELATIVE POSITION
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET IDENTITY
SET IDENTITY

i. Tentative Track

The class TENTATIVE TRACK was defined as a subclass of the class TRACK. Since the variable IDENTITY was defined for the class TRACK, it was inherited by the class TENTATIVE TRACK. The methods GET IDENTITY and SET IDENTITY were also inherited. Since the variable IDENTITY has the constant value *Unknown* for all instances

of this class, it was decided to "disable" the method SET IDENTITY by redefining it locally to do nothing.

Since the initial identity of a track of class TENTATIVE TRACK is always *Unknown*, it was decided to eliminate the method INITIAL IDENTITY ASSIGNMENT. The responsibility of assigning an initial identity to a new track was transferred to the method NEW TRACK ESTABLISHMENT. The method CLASS CHANGE PROCESSING was defined in order to transform of a tentative track into a firm track when appropriate.

In order to store the speed value above which an unknown track is considered an air track and below which an unknown track is considered a surface track, the variable CATEGORY SPEED was defined. Also, the variable TRACK NUMBER was replaced by variable TDB OBJECT NUMBER, which was inherited indirectly from the class TDB OBJECT. The class TENTATIVE TRACK was then defined as follows:

CLASS: TENTATIVE TRACK (Concrete)

Superclasses: TRACK

Class Variables: None

Instance Variables: CATEGORY: {Tent-Air, Tent-Surface, Tent-Subsurface} [*Tent-Air*]
CATEGORY SPEED: real [50.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
NEW TRACK ESTABLISHMENT
INITIAL CATEGORY ASSIGNMENT
CLASS CHANGE PROCESSING
SET IDENTITY

GET CATEGORY
SET CATEGORY
GET CATEGORY SPEED
SET CATEGORY SPEED

j. Firm Track

The class FIRM TRACK was defined as a superclass of the classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK. The only methods that were common to these three classes and were not defined for the superclass TRACK are IDENTIFICATION FUNCTION and TRACK HISTORY PROCESSING. The method IDENTIFICATION FUNCTION was renamed TRACK IDENTIFICATION as this new name better suggests the purpose of the method. The only instance variable defined for this class is POSITION NUMBER.

Therefore, FIRM TRACK was defined as follows:

CLASS: FIRM TRACK (Abstract)

Superclasses: TRACK

Class Variables: None

Instance Variables: POSITION NUMBER: integer [0]

Methods:
GET MENU
IDENTIFICATION FUNCTION
TRACK HISTORY PROCESSING
GET POSITION NUMBER
SET POSITION NUMBER

k. Air Track, Surface Track, and Subsurface Track

The only modification in each of these classes was the elimination of the variable TRACK NUMBER, which was replaced by the inherited variable TDB OBJECT NUMBER. These classes were then defined as follows:

CLASS: AIR TRACK (Concrete)

Superclasses: FIRM TRACK

Class Variables: None

Instance Variables: HEIGHT: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET CATEGORY
GET HEIGHT
SET HEIGHT

CLASS: SURFACE TRACK (Concrete)

Superclasses: FIRM TRACK

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET CATEGORY

CLASS: SUBSURFACE TRACK (Concrete)

Superclasses: FIRM TRACK

Class Variables: None

Instance Variables: DEPTH: real [0.0]

Methods:
GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET CATEGORY
GET DEPTH
SET DEPTH

1. Ownship

The class OWNSHIP was defined as a subclass of the class SURFACE TRACK and, consequently, inherited the variables RELATIVE POSITION, ORIGIN, IDENTITY, and POSITION NUMBER which were not initially defined for it. Even though these variables contain information that is not strictly necessary, they are meaningful in terms of the problem space. RELATIVE POSITION has the value *0:0:0:T:0.0*, ORIGIN has the value *Local Manual*, and IDENTITY has the value *Friendly*.

It was stated previously that the variable TRACK NUMBER (which was later replaced by the variable TDB OBJECT NUMBER) would be used to store a number for each position of the ship, thus allowing the system to maintain history positional data for Ownship. Since the class OWNSHIP also inherited the variable POSITION NUMBER, it is better, for uniformity, to use this variable to store the position number than to use the variable TDB OBJECT NUMBER (which replaced the variable TRACK NUMBER). The variable

TDB OBJECT NUMBER will then be used as a unique identifier for Ownship within the set of instances of the classes defined for the Tactical Database.

Also, OWNSHIP inherited the methods CALCULATE GEOGRAPHICAL POSITION, TRACK COURSE AND SPEED DETERMINATION, TRACK IDENTIFICATION, and the methods used to access and update variables GET RELATIVE POSITION, SET RELATIVE POSITION, GET IDENTITY, SET IDENTITY, and GET CATEGORY, which were not originally defined for it.

Since the method GET GEOGRAPHICAL POSITION was defined locally for the class OWNSHIP, the inherited method CALCULATE GEOGRAPHICAL POSITION is not strictly necessary. A solution to this redundancy is to define the method GET GEOGRAPHICAL POSITION in terms of the method CALCULATE GEOGRAPHICAL POSITION. The method TRACK COURSE AND SPEED DETERMINATION can replace the method NAVIGATIONAL COMPUTATION (specified in [Ref. 1]) and the methods TRACK IDENTIFICATION and GET CATEGORY simply return the values *Friendly* and *Surface*, respectively. The method OWNSHIP REPOSITIONING was eliminated because its functions can be executed by the method MONITOR OWNSHIP POSITION.

The class OWNSHIP was then defined as follows:

CLASS: OWNSHIP (Concrete)

Superclasses: SURFACE TRACK

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
GREENWICH MEAN TIME: TIME [00:00:00:Z]
JULIAN DATE: (0001..9366) [0001]
MARKTIME: HMS [00:00.00]

MAGNETIC VARIATION: MAGNETIC VARIATION [0:0:0:W]
OWNSHIP SET: DIRECTION [0:0:0:T]
OWNSHIP DRIFT: real [0.0]
WIND INFO: WIND [0:0:0:T:0.0]
NAVIGATION SOURCE: (Inertial, SATNAV, Omega, DR) [DR]

Methods:

GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
MONITOR OWNSHIP POSITION
BEARING AND RANGE FROM POSITION TO POSITION
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET GREENWICH MEAN TIME
SET GREENWICH MEAN TIME
GET JULIAN DATE
SET JULIAN DATE
GET MARKTIME
SET MARKTIME
GET MAGNETIC VARIATION
SET MAGNETIC VARIATION
GET OWNSHIP SET
SET OWNSHIP SET
GET OWNSHIP DRIFT
SET OWNSHIP DRIFT
GET WIND INFO
SET WIND INFO
GET NAVIGATION SOURCE
SET NAVIGATION SOURCE

m. Special Point

The class SPECIAL POINT was created to abstract the commonalities among the classes REFERENCE POINT, WAYPOINT, FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, DATA LINK REFERENCE POINT, and POSITION AND INTENDED MOVEMENT. The variable GEOGRAPHICAL POSITION, as well as the methods

CALCULATE RELATIVE POSITION, GET GEOGRAPHICAL POSITION, SET GEOGRAPHICAL POSITION, GET IDENTITY, and GET CATEGORY, are common to all of these classes.

As previously mentioned, according to [Ref. 1], the variable ORIGIN could assume only the value *Local Manual* for the classes WAYPOINT, POSITION AND INTENDED MOVEMENT, REFERENCE POINT, DATA LINK REFERENCE POINT, FORMATION CENTER, NAVIGATION HAZARD, and MAN IN WATER. However, it was decided that the variable ORIGIN can also assume the value *Remote* for the classes REFERENCE POINT, DATA LINK REFERENCE POINT, FORMATION CENTER, NAVIGATION HAZARD, and MAN IN WATER. This variable, which was inherited from the class TDB OBJECT, was then redefined for the class SPECIAL POINT, being allowed to assume only the values *Local Manual* and *Remote*.

The method GET IDENTITY, even though common to all of the above classes, returns a different constant value for each one of them. Therefore, this method was not defined for the class SPECIAL POINT because it would have to be redefined for each one of the classes. It is more appropriate, then, to define GET IDENTITY locally for each one of the classes.

The class SPECIAL POINT was then defined as follows:

CLASS: SPECIAL POINT (Abstract)

Superclasses: TDB POINT

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
ORIGIN: {Local Manual, Remote} [*Local Manual*]

Methods: GET MENU
CALCULATE RELATIVE POSITION
GET CATEGORY
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET ORIGIN
SET ORIGIN

n. Fixed Special Point

The class FIXED SPECIAL POINT was defined as the superclass of the classes REFERENCE POINT, DATA LINK REFERENCE POINT, and WAYPOINT. In the LCCDS scenario, an object of class FIXED SPECIAL POINT represents a special point which is stationary, that is, does not have the variables COURSE and SPEED defined for it. The only method defined (redefined) locally is the method GET MENU. FIXED SPECIAL POINT is an abstract class (i.e., it was not designed to be instantiated). This class was defined as follows:

CLASS: FIXED SPECIAL POINT (Abstract)

Superclasses: SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU

o. Reference Point

For the class REFERENCE POINT, the only methods defined locally are the methods RETURN IDENTITY and GET MENU. GET IDENTITY returns the constant value *Refpnt* for all instances of this class and GET MENU is redefined in order to meet the requirements of this class. Therefore, REFERENCE POINT was defined as follows:

CLASS: REFERENCE POINT (Concrete)

Superclasses: FIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

p. Data Link Reference Point

Similarly, for the class DATA LINK REFERENCE POINT, the only methods defined locally are the methods GET IDENTITY and GET MENU. GET IDENTITY returns the value *Dlrp* for all instances of this class and GET MENU is redefined to fit this class. DATA LINK REFERENCE POINT was then defined as follows:

CLASS: DATA LINK REFERENCE POINT (Concrete)

Superclasses: FIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

q. Waypoint

For the class WAYPOINT, the method GET MENU was redefined locally and the method GET IDENTITY was defined to return the constant value *Waypoint* for all instances of this class. The variable ORIGIN was redefined in order to allow only the constant value *Local Manual* for all instances of this class. Therefore, WAYPOINT was defined as follows:

CLASS: WAYPOINT (Concrete)

Superclasses: FIXED SPECIAL POINT

Class Variables: None

Instance Variables: STEAMING ROUTE: subset of {1,2,3,4,5,6} [{ }]
ORIGIN: {Local Manual} [*Local Manual*]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE

DISPLAY INSTANCE
OWNSHIP DISTANCE TO POSITION
ESTIMATED TIME OF ARRIVAL AT POSITION
ESTIMATED TIME ENROUTE TO POSITION
GET IDENTITY
GET STEAMING ROUTE
SET STEAMING ROUTE
GET ORIGIN
SET ORIGIN

r. Nonfixed Special Point

The class NONFIXED SPECIAL POINT was defined as the superclass of the classes FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT. The variables COURSE and SPEED characterize the objects of this class.

The method DEAD RECKONING was not defined initially for the classes MAN IN WATER and NAVIGATION HAZARD. However, since the geographical positions of the instances of these classes can change with time, it is important to provide a method to estimate future positions for these two classes. Therefore, DEAD RECKONING was defined for the class NONFIXED SPECIAL POINT, so that the classes MAN IN WATER and NAVIGATION HAZARD can also inherit it.

NONFIXED SPECIAL POINT was then defined as follows:

CLASS: NONFIXED SPECIAL POINT (Abstract)

Superclasses: SPECIAL POINT

Class Variables: None

Instance Variables: COURSE: ANGLE [0:0:0]
SPEED: real [0.0]

Methods: GET MENU
DEAD RECKONING
GET COURSE
SET COURSE
GET SPEED
SET SPEED

s. Formation Center, Navigation Hazard, and Man in Water

The method GET IDENTITY is defined locally for each one of these three classes. This method returns the constant values *Fc*, *Navhaz*, and *Man in Water* for all instances of the classes FORMATION CENTER, NAVIGATION HAZARD, and MAN IN WATER, respectively. Since these three classes are concrete classes, the methods CREATE INSTANCE, DELETE INSTANCE, UPDATE INSTANCE, and DISPLAY INSTANCE were defined locally for all of them.

These three classes were defined as follows:

CLASS: FORMATION CENTER (Concrete)

Superclasses: NONFIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

CLASS: NAVIGATION HAZARD (Concrete)

Superclasses: NONFIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

CLASS: MAN IN WATER (Concrete)

Superclasses: NONFIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

t. Position And Intended Movement

The method GET IDENTITY is defined locally and returns the constant value *Pim* for all instances of this class. The variable ORIGIN was redefined to allow only the value *Local Manual*.

The class POSITION AND INTENDED MOVEMENT was then defined as

follows:

CLASS: POSITION AND INTENDED MOVEMENT (Concrete)

Superclasses: NONFIXED SPECIAL POINT

Class Variables: None

Instance Variables: ORIGIN: {Local Manual} [*Local Manual*]

Methods:

- GET MENU
- CREATE INSTANCE
- DELETE INSTANCE
- UPDATE INSTANCE
- DISPLAY INSTANCE
- OWNSHIP DISTANCE TO POSITION
- ESTIMATED TIME OF ARRIVAL AT POSITION
- ESTIMATED TIME ENROUTE TO POSITION
- GET IDENTITY
- GET ORIGIN
- SET ORIGIN

D. DESCRIPTION OF THE METHODS DEFINED

The purpose of this section is to present a description of each of the methods defined for the Tactical Database. In describing a method, the following information is provided:

- Name of the method (the name of the method should suggest its purpose, as much as possible).
- Class(es) for which the method is defined.
- Brief description of how the method works.

Following are the methods that were defined for the Tactical Database:

1. Get Menu

Classes: All classes, either concrete or abstract.

Description: This method presents a list of options for user selection. For a class that has subclasses, the list of options includes that class' subclasses plus the methods defined locally for that class. For a class that does not have subclasses, the list of options includes only the methods defined locally for that class. Since each class has its own structure, this method should be redefined (i.e., defined locally) for each class.

2. Create Instance

Classes: All concrete classes.

Description: This method asks the user to enter the values of the variables defined for the class to which this instance is to belong. After these values are entered, a new instance of the class is created. It is recommended that this method be implemented by means of the OODBMS predefined operations for instance creation. Since each concrete class has its own structure, this method should be redefined (i.e., defined locally) for each concrete class.

3. Delete Instance

Classes: All concrete classes.

Description: This method asks the user to enter the name or identification number of the instance (object) to be deleted. It is recommended that this method be implemented by means of the OODBMS predefined operations for instance deletion. It is also recommended that user confirmation be requested before the deletion is executed. Since each concrete class has its own structure, this method should be redefined (i.e., defined locally) for each concrete class.

4. Update Instance

Classes: All concrete classes.

Description: This method displays a list of the variables (and their respective current values) defined for the instance. The user is allowed to modify any of these values. It is recommended that this method be implemented by means of the OODBMS predefined operations for instance update. Since each concrete class has its own structure, this method should be redefined (i.e., defined locally) for each concrete class.

5. Display Instance

Classes: All concrete classes.

Description: This method displays a list of the variables, and their respective current values, defined for the instance. The user is not allowed to modify these values. It is also recommended that this method be implemented by means of the OODBMS predefined operations for instance display. Since each concrete class has its own structure, this method should be redefined (i.e., defined locally) for each concrete class.

6. Get "Variable"

Classes: All classes for which at least one variable was defined, and all classes for which at least one method was used to store a constant value (rather than defining a class variable).

Description: The term "variable" represents the name of the variable that is to be accessed. This method returns the value of the variable requested.

7. Set "Variable"

Classes: All classes for which at least one variable was defined, and all classes for which at least one method was used to store a constant value (rather than defining a class variable).

Description: The term "variable" represents the name of the variable that is to be updated. This method is generally used in conjunction with the method GET "VARIABLE". The user is allowed to modify the value of the requested variable.

8. CPA Processing

Classes: PREDEFINED OBJECT

Description: This method should access the geographical position, course, and speed of the two instances (objects) and, by means of extrapolation, calculate the new geographical positions in which the instances (objects) will be closest to each other. Information about range, bearing, and time between the objects should also be provided.

This method should use the methods GET CLOSE CPA and GET COLLISION CPA in order to issue notifications when an object meets one of the cpa criteria.

9. Convert String to Numeric

Classes: TDB OBJECT

Description: The method converts a string to numeric format. It is recommended that this method be implemented by means of the operations for type conversion provided by the OODBMS used.

10. Convert Numeric to String

Classes: TDB OBJECT

Description: The method converts a number to string format. It is recommended that this method be implemented by means of the operations for type conversion provided by the OODBMS used.

11. Create New Class

Classes: USERDEFINED OBJECT

Description: The method should allow the user to list the variables and methods that will compose the new class. This method should be used in conjunction with the methods CREATE NEW VARIABLE and CREATE NEW METHOD. It is recommended that this method be implemented by means of the operations for class definition (creation) provided by the OODBMS used.

12. Create New Variable

Classes: USERDEFINED OBJECT

Description: The method should allow the user to specify the variable's name, type, and default value. It is recommended that this method be implemented by means of the operations for variable definition provided by the OODBMS used.

13. Create New Method

Classes: USERDEFINED OBJECT

Description: The method should allow the user to define a method containing any of the operations provided by the OODBMS used. It is recommended that this method be implemented by means of the operations for method definition provided by the OODBMS used.

14. Dead Reckoning

Classes: TDB AREA, TRACK, and NONFIXED SPECIAL POINT.

Description: Based on the input parameters and assuming constant course and speed, the method calculates by extrapolation the geographical position of the object (instance) at the desired time (or after a desired period of time). The user should be allowed to enter either the time of the desired position, or the period of time after which the position is to be calculated.

15. Plot TDB Point

Classes: TDB POINT

Description: The method retrieves the geographical position of the instance requested. This method should interact directly with the Man-Machine Interface, which is responsible for plotting the desired point on the screen.

16. Draw TDB Line

Classes: TDB LINE

Description: The method retrieves the geographical positions of the points that defined the desired line. This method should interact directly with the Man-Machine Interface, which is responsible for drawing the desired line on the screen.

17. Draw TDB Area

Classes: TDB AREA

Description: This method retrieves the values of the variables NORTH LIMIT, SOUTH LIMIT, EAST LIMIT, and WEST LIMIT which are the limits of a rectangular area. Since the contour of an area can be drawn by means of lines, this method can be implemented using the method DRAW LINE. This method should interact directly with the Man-Machine Interface, which is responsible for drawing the desired area on the screen.

18. Calculate Geographical Position

Classes: TRACK

Description: Based on Ownship's geographical position and the relative position of the object (instance) with respect to Ownship, the method calculates the desired object's geographical position.

19. Track Course And Speed Determination

Classes: TRACK

Description: This method should retrieve at least two previous relative positions of the track with respect to Ownship and their respective time of position and, based on these data, calculate the track's speed and course.

20. New Track Establishment

Classes: TENTATIVE TRACK

Description: The method should allow the user to manually enter the new track's relative position, course, and speed. The newly entered track should be initiated as an instance of the class TENTATIVE TRACK, and the value *Unknown* should be assigned to the new track's identity. Another function of this method is to call (activate) the method INITIAL CATEGORY ASSIGNMENT.

21. Initial Category Assignment

Classes: TENTATIVE TRACK

Description: This method should access the value of the variable **CATEGORY SPEED** and assign the category to the new track according to the following criterion:

- track speed less than category speed: *Surface*
- track speed greater than category speed: *Air*

The preset value of the category speed shall be 50.0 knots.

22. Class Change Processing

Classes: TENTATIVE TRACK

Description: This method should delete an instance of the class **TENTATIVE TRACK** and create an instance of one of the classes **AIR TRACK**, **SURFACE TRACK**, or **SUBSURFACE TRACK**, as a result of one of the following conditions:

- After three manual updates by the user.
- When the category is entered manually before three position updates.
- When the track is manually declared firm.

23. Track Identification

Classes: FIRM TRACK

Description: This method uses the methods **GET IDENTITY** and **SET IDENTITY** in order to access and update the identity of the requested track. The new identity of the track is entered manually by the user.

24. Track History Processing

Classes: FIRM TRACK

Description: This method retrieves the previous positions and respective times of position of the requested track, starting at a time or geographical position specified by the user. This method should interact with the Man-Machine Interface, which is responsible for displaying the retrieved points. It is recommended that this method be implemented by means of the operations for query processing provided by the OODBMS used.

25. Monitor Ownship Position

Classes: OWNSHIP

Description: This method should allow the user to manually enter the Ownship's geographical position, course, speed, time of position, GMT, Julian Date, and marktime.

26. Bearing and Range from Position to Position

Classes: OWNSHIP

Description: Based on the two given geographical positions, this method calculates the range and bearing from the first to the second selected point.

27. Calculate Relative Position

Classes: TDB AREA and SPECIAL POINT

Description: Based on Ownship's geographical position, this method calculates the relative position of the object (instance) with respect to Ownship.

28. Ownship Distance to Position

Classes: WAYPOINT and POSITION AND INTENDED MOVEMENT

Description: Based on the geographical positions of Ownship and the instance, the distance between the two points is calculated.

29. Estimated Time Enroute to Position

Classes: WAYPOINT and POSITION AND INTENDED MOVEMENT

Description: This method can be implemented by means of a call to the method OWNSHIP DISTANCE TO POSITION, whose output is the distance from Ownship to the desired instance. Therefore, based on this distance and on Ownship's speed, the time enroute is calculated.

30. Estimated Time of Arrival at Position

Classes: WAYPOINT and POSITION AND INTENDED MOVEMENT

Description: This method can be implemented by means of a call to the method ESTIMATED TIME ENROUTE TO POSITION, whose output is the time enroute to the desired instance. Based on this time and on the current time, the time of arrival at the instance is calculated.

31. Route Geometry

Classes. ROUTE

Description: For each waypoint that forms the selected route, this method calculates the course at the desired speed that Ownship or a selected track must take in order to intercept the waypoint. Estimated time of arrival at the waypoint and estimated time enroute to the waypoint are also calculated. The route geometry calculation should be updated every four seconds. The estimated times of arrival and estimated times enroute can be calculated by means of the methods ESTIMATED TIME OF ARRIVAL AT POSITION and ESTIMATED TIME ENROUTE TO POSITION, respectively.

V. ADDITIONAL DESIGN CONSIDERATIONS

This chapter presents some considerations about topics that complement the design, having a direct influence on data reliability and database performance. These topics are: physical storage management (clustering), concurrency control, crash recovery, garbage collection, and database security.

A. PHYSICAL STORAGE MANAGEMENT

The purpose of this section is to present some preliminary ideas about how the objects should be clustered on disk. Even though the ideas proposed represent the present opinion of this author, it is important to realize that further research is still necessary.

1. Guidelines for Object Clustering

The way in which objects are stored on disk affects the overall performance of the system. In general, when objects that are often used together are placed in contiguous areas of the disk, overall retrieval time is reduced.

In order to determine how the objects should be clustered on disk, the following guidelines are proposed:

- Analyze the clustering facilities provided by the OODBMS used.
- Answer the following question: In the system used, can instances of the same class be stored in separate clusters?

- Place all instances of a class that is "not" a component of several composite classes in the same disk cluster, if possible.
- Do not cluster together instances of a class that is a component of several composite classes, if possible.
- Place all component classes of a composite class in the same disk cluster, if possible.
- Cluster only concrete classes.

a. Analyze the Clustering Facilities provided by the OODBMS used

The physical storage of the objects on disk depends on the system used. Different systems use different default clustering methods. Additionally, they may or may not allow user-defined clustering.

In ONTOS [Ref. 21], all elements (components) of an object are stored together so that they can be retrieved efficiently. Related objects can be clustered on disk in any arbitrary, programmer-defined grouping, thereby allowing objects to be clustered according to the anticipated application usage. Clustering two or more objects together means storing them in the same segment. Segments are variable-sized atomic units of transfer between secondary storage and main memory [Ref. 20]. Whenever one of the objects is accessed from disk and brought into main memory, all objects in that cluster are also brought into main memory. Subsequent access to any of these objects is then a main memory access rather than a disk access. Thus, when done properly, clustering can be used to great advantage in application performance. [Ref. 20]

In ORION [Ref. 18], all instances of a class are placed in the same segment¹. Thus a class is associated with a single segment and all its instances reside in that segment. The user does not have to be aware of segments as ORION automatically allocates a separate segment for each class. However, for clustering composite objects it is often necessary to store instances of multiple classes in the same segment. The user is required to determine which classes should be stored together in a segment. [Ref. 18]

One of the original goals of GemStone [Ref. 25] was that the system should provide features for managing the placement of objects on disk. The database administrator, or an application programmer, should be able to specify how certain objects should be clustered on the disk. GemStone provides two clustering methods: one basic and one more sophisticated.

The basic clustering method simply assigns each class to a disk page. It does not attempt to cluster the class' instance variables. If the user wishes to cluster the instance variables of an object, a special method must be defined to do so, using the basic clustering method as a tool.

The more sophisticated method does a depth-first traversal of the tree representing the class' instance variables, that is, it writes to the disk the class' first instance variable, then the first instance variable of that instance variable, then the first instance variable of that instance variable, and so on to the bottom of the tree. It then

¹ Even though the term "segment" is not defined in [Ref. 18], it is likely that it refers to the same type of logical storage unit defined in [Ref. 20], as previously discussed.

backs up and visits the variables it missed before, repeating the process until the entire tree has been written. [Ref. 25]

b. In the System Used, Can Instances of the Same Class be Stored in Separate Clusters?

If the system permits instances of the same class to be stored in separate clusters, then the user has total freedom to decide how to cluster the objects. In this case, all components of a composite object can be stored together. However, if the system requires that instances of the same class be stored only in the same cluster (i.e., if instances of the same class cannot be stored in separate clusters), then the user is limited to creating only clusters of classes.

c. Place All Instances of a Class that Is Not a Component of Several Composite Classes in the Same Cluster

The placement of all instances of a class that is not a component of several composite classes in the same cluster allows the retrieval of all instances of a class in only one disk access. However, if a class has a large number of instances, the cluster may become too large (possibly even larger than the main memory) to be efficiently read into main memory. In this case, the only solution is to store the instances in more than one cluster.

d. Do Not Cluster Together Instances of a Class that Is a Component of Several Composite Classes

When a class is a component of several composite classes, the number of classes to be stored together may also become too large to be handled efficiently by the

system. If the system permits, a solution to this problem may be obtained by not clustering the instances of that class together. Instead, the clustering is done in such a way that every one of the instances of the component class is stored together with its corresponding composite object.

e. Place All Component Classes of a Composite Class in the Same Cluster

Ideally, the components of a composite object (class) should be stored together at all times [Ref. 18]. However, when a class has several component classes, a cluster including all of these classes might be too large to be handled efficiently by the system.

f. Cluster Only Concrete Classes

Since abstract classes have no instances, the only storage space occupied by these classes is the space necessary to store their class definitions. On the other hand, concrete classes may have a large number of instances (objects) and, consequently, besides the space necessary for their class definitions, may require a large amount of storage space for their instances. The purpose of clustering objects on disk is to reduce the time required to retrieve information, which is contained (stored) in the instances of the class, not in the class definition itself. Therefore, it is not necessary to cluster abstract classes.

2. Clustering in the Tactical Database

a. Clustering of the Composite Classes

In the Tactical Database, the variable TIME OF POSITION, which is an instance (object) of class TIME, was defined for most of the classes. Consequently, most of the objects (classes) in the Tactical Database are composite objects (classes) which have

TIME as one of their component classes. In order to store TIME together with all these classes, it would be necessary to create a cluster that would include most of the classes defined for the Tactical Database (i.e., most of the classes would have to be clustered together on disk). The resulting cluster would certainly be too large to be used efficiently by the system.

Therefore, it was decided that the instances of class TIME should not be clustered together. The clustering should be done in such a way that instances of TIME (represented by the variable TIME OF POSITION) are clustered together with their corresponding composite object.

A similar situation occurs with the classes ANGLE, LATITUDE, LONGITUDE, GEOGRAPHICAL POSITION, DIRECTION, RELATIVE POSITION, CLOSE CPA, COLLISION CPA, and HMS. These classes are component classes of several different composite classes (objects). Storing them with their respective composite classes would result in the creation of large and inefficient clusters. Thus, it was decided not to cluster the instances of these classes together, but to store the instances of these classes with their corresponding composite objects.

On the other hand, the classes WIND and MAGNETIC VARIATION are component classes only of the composite class OWNERSHIP. Therefore, it was decided to cluster them together with the class OWNERSHIP. Similarly, the classes DISTANCE RANGE and TIME RANGE are components only of the classes CLOSE CPA and COLLISION CPA. Therefore, it was decided to cluster these four classes together. Also,

since the class WAYPOINT is a component class only of the class ROUTE, it was decided to cluster these two classes together.

b. Clustering of Classes that Are Used Together

In the Tactical Database scenario, every new track is first classified as a tentative track and later, after a valid course and speed is established for it, becomes either an air track, a surface track, or a subsurface track. Thus, the class TENTATIVE TRACK is often used together with the classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK. Therefore, the instances of these four classes should be clustered together.

Also, all the tracks' relative positions and closest points of approach (CPAs) are calculated based on Ownship's geographical position and, consequently, the classes TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK are in general used together with the class OWNSHIP. However, the same fact is true for all special points. That is, their relative positions and closest points of approach (CPAs) are calculated based on Ownship's geographical position and, thus, the classes REFERENCE POINT, DATA LINK REFERENCE POINT, WAYPOINT, FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT are also used together with the class OWNSHIP.

Therefore, there are four main clustering possibilities: cluster OWNSHIP with all tracks; cluster OWNSHIP with all special points; cluster OWNSHIP with both tracks and special points; or do not cluster OWNSHIP with either of them.

Clustering OWNSHIP with both tracks and special points would result in a cluster that would probably be too large to be handled efficiently by the system, since most of the classes defined for the Tactical Database are either subclasses of the class TRACK or subclasses of the class SPECIAL POINTS.

Clustering OWNSHIP with all special points or not clustering OWNSHIP with either tracks or special points are viable alternatives. However, considering that the time to retrieve information about a track is critical for the safety of the ship, it is necessary to have all information about a track available as fast as possible. The fastest way of retrieving information about a track is achieved when the class OWNSHIP is clustered with all tracks (assuming that there are no problems of memory size).

Therefore, it was decided that the classes TENTATIVE TRACK, AIR TRACK, SURFACE TRACK, SUBSURFACE TRACK, and OWNSHIP should be clustered together on disk. Remember that the classes WIND and MAGNETIC VARIATION should also be clustered together with the class OWNSHIP.

c. Clustering of the Other Classes

The remaining concrete classes defined for the Tactical Database are TDB AREA, USERDEFINED OBJECT, REFERENCE POINT, DATA LINK REFERENCE POINT, FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT. The only class that is often used together with each one of these classes is the class OWNSHIP. However, it was decided that OWNSHIP should be clustered with all the classes that represent tracks.

Thus, there is no strong reason for storing any of these classes together. However, it is almost always practical to cluster together all instances of the same class. In order to achieve this, the classes TDB AREA, USERDEFINED OBJECT, MAN IN WATER, REFERENCE POINT, DATA LINK REFERENCE POINT, FORMATION CENTER, NAVIGATION HAZARD, and POSITION AND INTENDED MOVEMENT should each be stored in a separate cluster.

B. CONCURRENCY CONTROL

In a database management system (DBMS) that provides concurrent access, each user should see a consistent version of the database, even when other users are running simultaneously. When a user logs in, the DBMS establishes for him or her a logical entity called a **session**, which is analogous to an operating system session, job, or process. A separate session is created each time a user logs in, and the DBMS monitors, serves, and protects each session independently. In general, the DBMS prevents inconsistencies by encapsulating a session's operations (computations, stores, and fetches) in units called **transactions**. The operations that make up a transaction act on what appears to be a private copy of the objects. This "copy" is called a **workspace**. It is only when the user tells the DBMS to **commit** the current transaction that it tries to merge the modified objects in the user's workspace with the main, shared object store. [Ref. 25]

In general, when the user tells the DBMS to commit a transaction, it checks for two conditions that would indicate conflict with the activities of other concurrent users:

- First, it finds out whether other concurrent sessions have committed transactions of their own, modifying objects that were read or modified during the user's transaction. If they have, then the user may have used outdated values in some computations.
- Second, it checks for "locks" set by other sessions that would indicate their intention to modify objects that the user has read or read objects that the user has modified. The presence of such locks would mean that committing the user's changes might invalidate another user's work.

If neither of these conditions is found, the transaction is committed. This not only makes the new and changed objects visible to other users as a permanent part of the shared database, but also makes visible to the user any new and modified objects that have been committed by other users. [Ref. 25]

If, on the other hand, the system finds a conflict or potential conflict, then it refuses to commit the user's modifications. When a transaction fails to commit, it leaves the user's workspace intact with all of the new and modified objects it contains. [Ref. 25]

The user can then abort the transaction and start a new one. This discards all of the new objects and modifications from the aborted transaction. Depending on the activities of other users, the user may be able to repeat the operations using the new values from the database and commit this new transaction without encountering conflicts. [Ref. 25]

It should be realized that the process of committing a transaction is an all-or-nothing method of posting the user's updates to the main object store. The system either commits all of the modifications encapsulated in a transaction at once, or it commits none of them. Because of this property, transactions are said to be "atomic". All permanent object modifications are encapsulated in transactions and are therefore atomic. The system moves from one internally consistent state to another, as users commit their changes, and no

inconsistent data can be introduced as a result of concurrent changes that conflict with one another. One can think of the entire set of operations encapsulated within a transaction as occurring in the instant when the transaction is committed. [Ref. 25]

1. Concurrency Management Mechanisms

Most of the concurrency control mechanisms can be categorized into two main approaches: one that **prevents** conflicting actions during transaction execution (pessimistic concurrency control); and one that **discards** conflicting updates when a transaction attempts to commit (optimistic concurrency control). [Ref. 16]

a. Optimistic Concurrency Control

In the optimistic mechanism, the user simply reads and writes objects at will as if he were the only user. The system searches for and detects conflicts with other sessions only at the time the user tries to commit the transaction. Although relatively easy to implement, this mechanism entails the risk that the user will lose all of the work that was done if conflicts are detected and the transaction cannot be committed. [Ref. 25]

b. Pessimistic Concurrency Control

In the pessimistic mechanism, there are two main techniques: one based on locks and one based on timestamps.

A **lock** is a variable associated with an object in the database and describes the status of that object with respect to possible operations that can be applied to it. Generally, there is one lock for each object in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database. In the locking technique, the user acts as early as possible to detect and prevent conflicts by explicitly

requesting locks that signal his intentions to read or write objects. If the user succeeds in locking an object, then other users will henceforth be unable to use the object in a way that would conflict with the locking user's purposes. If the user is unable to acquire a lock, then he knows that someone else has already locked the object, and therefore he cannot use the object and then commit. The user can then abort the transaction immediately, instead of wasting time on work that can't be committed. [Ref. 25]

A **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the DBMS, so a timestamp can be thought of as the "transaction start time". Transactions can therefore be ordered according to their timestamps to ensure serializability. [Ref. 3]

Timestamps can be generated in several ways. One possibility is to have a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps will be numbered 1,2,3,... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current value of the system clock. [Ref. 3]

Since timestamps are automatically created by the DBMS, the user in general has little or no control over the concurrency control mechanism. On the other hand, in the locking mechanism, it is up to the user whether or not to request a lock on an object, thus being able to control the concurrency control mechanism. Consequently, in the pessimistic concurrency control approach, emphasis will be given to the locking mechanism.

2. Optimistic versus Pessimistic Concurrency Control

Optimistic concurrency control is the most efficient mode of operation if [Ref. 25]:

- the user is not sharing data with other sessions, or
- the user is reading data but not writing, or
- the user is writing a limited amount of shared data and can tolerate not being able to commit his work sometimes.

In the optimistic mode, the system only looks for conflicts at commit time. Therefore the user's chances of being in conflict with other users increase with the time between commits and the size of the user's read set.

Controlling concurrent access with locks (pessimistic concurrency control) is the most efficient mode of operation if [Ref. 25]:

- there is a lot of competition for shared data in the user's application, or
- the user cannot tolerate even an occasional inability to commit.

It is important to keep in mind that the locking mechanism improves one user's chances of committing only at the expense of other users. This means that the user should use locks sparingly to prevent an overall degradation of system performance.

3. Concurrency Control in OODBMSs

The EXODUS [Ref. 27] and ORION [Ref. 28] systems both use a locking technique (pessimistic approach) for concurrency control. In particular, ORION provides a mechanism that allows the locking of a composite object along with its component objects as a unit. The AVANCE object management system [Ref. 29] uses the timestamp

model (pessimistic approach) for concurrency control. The Vbase [Ref. 20], ONTOS [Ref. 21], and GemStone [Ref. 25] systems all provide both pessimistic and optimistic concurrency control.

Since, according to [Ref. 24], GemStone is the system that was recommended for use in the LCCDS, it is important to mention some details about its concurrency control mechanisms. The default mechanism is optimistic concurrency control. That is, this mechanism is always in effect for objects which the user has not locked.

With GemStone in optimistic concurrency control mode, a transaction which fails to commit leaves the user's workspace intact with all of the new and modified objects it contains. The user may then either take some action to save the values of those objects in a file outside GemStone, or abort the transaction altogether (in GemStone, when a transaction is aborted, the system automatically starts a new transaction). [Ref. 25]

In the pessimistic concurrency control mode, three kinds of locks are provided: read, write, and exclusive. A session may hold only one kind of lock on an object at a time.

Holding a **read lock** on an object means that the user can make computations based on the object's value and then commit without fear that some other transaction might have committed a new value for that object after the read lock was obtained. Another way of saying this is that holding a read lock on an object guarantees that other sessions cannot:

- acquire a write or exclusive lock on the object, or
- commit if they have written the object.

Multiple sessions may hold read locks on the same object. Therefore, read locks are also known as "shared" locks.

Holding a **write lock** on an object guarantees that the user can write the object and commit. That is, it ensures that the user will not find that someone else has prevented him from committing by writing the object and committing it before him during the transaction. Another way of looking at this is that holding a write lock on an object guarantees that the other sessions cannot:

- acquire any kind of lock on the object, or
- commit if they have written the object.

Write locks differ from read locks in that only one session may hold a write lock on an object. In fact, if a session holds a write lock on an object, then no other session may hold any kind of lock on the object. This prevents other sessions from receiving the assurance implied by a read lock that the value of the object it sees in its workspace will not be out of date when it attempts to commit a transaction. Other sessions may, however, still read the object without acquiring any type of lock of their own.

An **exclusive lock** is like a write lock in that it guarantees the user's ability to write an object. It goes beyond a write lock by guaranteeing that other sessions cannot:

- acquire any kind of lock on the object, or
- commit if they have written or read the object.

GemStone's exclusive locks correspond to what traditional data management systems call exclusive locks or sometimes just write locks. By contrast, GemStone's write locks are not exclusive in the conventional sense, since other sessions may be able to read a write-locked object optimistically (i.e., without holding a lock) and still commit.

In addition to the locks for single objects, GemStone also allows locks on collections of objects.

4. Concurrency Control in the Tactical Database

The Tactical Database will interact with four external interfaces: the Radar System (sensors interface), the Link 11 System (data link interface), the Navigation System (navigation interface), and the user or LCCDS operator (man-machine interface).

Data will be written to the TDB through these four external interfaces, but only the user (through the man-machine interface) will be allowed to read data from the TDB.

In addition, some updates to the TDB will occur internally. That is, the system will automatically read data from the TDB, perform some calculations (for example, dead reckoning or waypoint maneuvering geometries) and then write the results back to the TDB.

According to [Ref. 1], the system shall update the Ownship position and velocity, all waypoint maneuvering geometries, and all track data at least every four seconds. Also, the system shall automatically perform dead reckoning computations for all tracks if no update is received within four seconds.

Thus, in order to meet these requirements, all automatic operations performed by the system will have to be short duration transactions (at most four seconds). For short

duration transactions, the optimistic concurrency control technique is the most efficient approach.

Therefore, it is recommended that optimistic concurrency control be used for all automatic updates (either Local Auto or Remote) performed by the Tracking, Data Link, and Navigation Systems. When a transaction (update) fails to commit, it is recommended that the transaction be aborted (discarding the data), because the next update, with new data, will occur within the next four seconds.

On the other hand, updates (transactions) performed manually by the user (Local Manual) will certainly take longer than four seconds and, therefore, will require locks, otherwise they would have little chance to commit, because the automatic updates would commit first.

Therefore, it is recommended that pessimistic concurrency control be used for all updates performed by the user (Local Manual updates). Once the user acquires a lock on an object, the transaction is then guaranteed to commit.

Transactions that involve schema modification (e.g., changes in the definition of a class) will require pessimistic concurrency control. In such cases, to avoid inconsistencies, it is recommended that the user request exclusive locks not only on the classes being modified, but also on all descendant classes of these classes.

C. CRASH RECOVERY

Transaction recovery means preservation of the atomicity property of a transaction. This means that, ideally, despite all possible failures of the computer system, all updates

of a committed transaction will be stored in the database, and all updates of an aborted transaction will be purged from the database. Of course, no database system can support transaction recovery against all possible failures of the computer system, which may include simultaneous failures of the processor, main memory, secondary memory, and communication medium between processors. Most commercial database systems support database recovery from **soft crashes** (which leave the contents of disk intact) and **hard crashes** (which destroy the contents of disk). [Ref. 28]

1. Recovery in OODBMSs

In ORION [Ref. 28], ONTOS [Ref. 21], Vbase [Ref. 20], and GemStone [Ref. 25], as well as in most multi-user systems, the unit of recovery from soft crashes is the transaction. This means that changes made by committed transactions are kept, and changes not yet committed are lost.

To guard against hard crashes, GemStone and Vbase allow the user to create backup files of the database. However, in Vbase the backup copy of the database is immutable: it can be deleted or replaced, but not modified [Ref. 20]. On the other hand, in GemStone replicates of the database can be created. That is, the system copies all objects from the file that contains the database to a new file, and afterward stores newly-committed objects in both files. Subsequent damage to one file leaves all objects intact in the other one, allowing the system to continue to function normally with no loss of data. Only authorized users are allowed to create database replicates. GemStone allows the creation of up to 6 replicates. Of course, maintaining replicates of the database on line has a cost in both time and space.

2. Recovery in the Tactical Database

Soft crashes in the Tactical Database could result in the loss of the data not yet committed to the database. Since most transactions in the TDB are of short duration (automatic transactions are expected to take less than four seconds, while manual transactions, in general, take just a few minutes), only recent information will be lost. Therefore, unless the soft crash is caused by hardware failure (which will cause the system to be "down" for a considerable amount of time), the database can be updated rapidly.

Since GemStone (the system recommended in [Ref. 24]) allows authorized users to create replicates of the database, at least one backup copy of the database should be kept on disk (preferably on a separate disk). This will greatly reduce the chances of losing all data due to hard crashes.

D. GARBAGE COLLECTION

Each time a transaction commits, the database pages containing the old version of data changed by the transaction become inaccessible. Such pages are considered "garbage" since they are not part of free space and do not contain usable information. Garbage may be created also as a side-effect of crashes. Periodically, it is necessary to find all the garbage pages and add them to the list of free pages. This process is called **garbage collection** and, of course, imposes additional overhead and complexity on the system.

[Ref. 30]

1. Garbage Collection in OODBMSs

In an OODBMS, it is important to perform garbage collection periodically, in order to reclaim the storage space consumed by unreferenced objects (objects that are no longer used).

In AVANCE [Ref. 29], garbage collection is performed on user request. In GemStone [Ref. 25], garbage collection is executed in single-user mode and can only be performed by authorized users.

2. Garbage Collection in the Tactical Database

As previously mentioned, only authorized users can perform garbage collection in GemStone. Since all qualified LCCDS operators should be capable of fully operating the system, they should all be authorized to perform periodic garbage collection.

The frequency in which garbage collection should be performed will depend, in general, on the number of objects deleted (unreferenced) during LCCDS operation. Since garbage collection may be a relatively long process which requires that no interfaces be logged in (sending data) to the database, it is recommended that it be performed only in situations of low risk to the ship.

E. DATABASE SECURITY

The data stored in a database needs to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. The term **database security** usually refers to security from malicious access, while **database integrity** refers to the avoidance of accidental loss of consistency. [Ref. 30]

Accidental loss of consistency may result from [Ref. 30]:

- Crashes during transaction processing.
- Anomalies due to concurrent access to the database.
- Anomalies due to the distribution of data over several computers.
- A logical error that violates the assumption that transactions preserve the database consistency constraints.

The techniques of recovery and concurrency control are used to protect the database against accidental loss of consistency. It is easier to protect the database against loss of data consistency than to protect against malicious access to the database. Among the forms of malicious access are the following [Ref. 30]:

- Unauthorized reading of data (theft of information).
- Unauthorized modification of data.
- Unauthorized destruction of data.

Absolute protection of the database from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access the database without proper authority. [Ref. 30]

Therefore, in order to protect the database, security measures must be taken at several levels [Ref. 30]:

- **Physical.** The site or sites containing the computer systems must be physically secured against armed or surreptitious entry of intruders.

- **Human.** Authorization of users must be done carefully to reduce the chance of an authorized user giving access to an intruder in exchange for a bribe or other favors.
- **Operating system.** No matter how secure the database system is, weakness in operating system security may serve as a means of unauthorized access to the database. Since almost all database systems allow remote access through terminals or networks, software-level security within the operating system is as important as physical security.
- **Database system.** Some authorized database system users may be authorized to access only a limited portion of the database. Other users may be allowed to issue queries, but may be forbidden to modify the data. It is the responsibility of the database system to ensure that restrictions such as these are not violated.

It is worthwhile in many applications to devote a considerable effort to the preservation of the integrity and security of the database. Loss of data, whether via accident or fraud, may seriously impair the ability of a corporation to function or, in the case of the Tactical Database, could result in severe or catastrophic consequences.

1. Security in OODBMSs

The System Documentation Manuals of Vbase [Ref. 20] and ONTOS [Ref. 21] do not mention any security mechanism. This suggests that these two systems do not provide mechanisms to avoid malicious access to the database.

GemStone [Ref. 25] provides the following kinds of security mechanisms to help control access to sensitive code and data: login authorization, name hiding, procedural protection, and nonprocedural protection (authorization and privileges). The user may choose to employ any or all of these mechanisms.

a. Login Authorization

Login authorization provides GemStone's first line of protection. "Logging in" is the process of establishing a logical connection with GemStone that permits further interaction. It is analogous to logging in to a timeshared operating system. The GemStone system administrator, or someone with equivalent privileges, establishes user ID's and passwords for authorized users. [Ref. 25]

b. Name Hiding

The system administrator assigns to each user a symbol list, which contains the names of all the system-defined objects that the user might need. Although the decision about which objects to include is entirely up to the system administrator, the user's symbol list probably contains [Ref. 25]:

- A system dictionary which contains some or all of the system-defined classes, and any other objects to which all GemStone users may need to refer. Although the user can read the objects in this dictionary, he is generally not permitted to modify them.
- A private dictionary in which user-defined objects that are not to be shared with other users are stored.
- One or more special-purpose dictionaries which can be shared with other users.

It can be difficult or even impossible for a user to refer to global objects that are not in his symbol list. Consequently, just omitting off-limits objects from the user's symbol list provides a certain amount of security. However, determined users may still find ways to circumvent this, since it is difficult to ensure that all indirect paths to an object are eliminated. [Ref. 25]

c. Procedural Protection

If the user's program accesses its objects only through methods, he can control the use of those objects by including user identity checks in their methods. Obviously, this kind of checking on a large scale would require a lot of code, and, consequently, might be troublesome to maintain and modify. [Ref. 25]

d. Nonprocedural Protection

GemStone provides two kinds of mechanisms that are global and nonprocedural: authorization and privileges [Ref. 25].

(1) Authorization. GemStone's **authorization** mechanism protects objects from access by users who have not been explicitly given permission to use them. Every user can employ the authorization mechanism to protect both data and code objects selectively.

GemStone enables the user to endow both other users and objects with authorization attributes. Whenever a program tries to read or write an object, GemStone compares the object's authorization attributes with those of the user whose program is attempting to do the reading or writing. If the two share some attributes, the operation may proceed. If not, GemStone returns an error notification.

The authorization mechanism allows the user to authorize reading an object without giving authority to write it. Authorizations are easy to change, and they don't require extra code in the objects they protect.

(2) Privileges. The **privilege** mechanism, which is entirely independent of the authorization system, enables the system administrator to control who can send certain powerful messages, such as those that halt the system or change passwords. Privileges are

associated only with a few such methods, and the mechanism cannot be extended to control other methods.

Therefore, while authorization can be used to control access to any kinds of objects, the privilege mechanism only guards a small but crucial set of methods. The system administrator determines who shall be privileged to invoke each of a small group of messages which, together, exert life-and-death control over all the GemStone's objects and functions.

2. Security in the Tactical Database

The Tactical Database will be used in a multi-user form, being shared by the Track System, Link 11 System, Navigation System, and LCCDS operator (user). However, since all transactions performed by the Track System, Link 11 System, and Navigation System are automatic (i.e., following a programmed scheme), the only manual access to the database will be performed by the LCCDS operator. Since all LCCDS operators should be capable of fully operating the system in all circumstances, there should be very little need to control access to sensitive code and data. Thus, all qualified LCCDS operators should be given "login authorization" and they should all have access to all objects in the Tactical Database (i.e., they should all have the privileges of a system administrator).

VI. COMMENTS AND CONCLUSIONS

A. DESIGN APPROACH

According to [Ref. 31], object-oriented design practices may take either of two different approaches: a data-driven approach or a responsibility-driven approach. Wirfs-Brock and Wilkerson [Ref. 31] argue that data-driven approaches fail to maximize encapsulation because they focus too quickly on the implementation of objects. A responsibility-driven approach, on the other hand, can increase encapsulation by deferring implementation issues until a later stage.

1. Data-Driven Design

In a data-driven design, the primary focus is on the structure of the data being represented in the system. Objects are designed by asking the two following questions [Ref. 31]:

- What (structure) does this object represent?
- What operations can be performed by this object?

The main benefit of the data-driven approach is that it is a familiar process for programmers experienced with traditional procedural languages. It is relatively easy for such programmers to adapt their previous experience to the design of object-oriented systems. [Ref. 31]

Even though the goal of data-driven design is to encapsulate data and algorithms, it inherently violates that encapsulation by making the structure of an object part of its definition. This in turn leads to the definition of operations that reflect that structure (because they were designed with the structure in mind). Attempts to transparently change the structure of an object are destined to fail because other classes rely on that structure. This is the antithesis of encapsulation. [Ref. 31]

2. Responsibility-Driven Design

The goal of responsibility-driven design is to improve encapsulation. It does so by viewing a program in terms of the client/server model. [Ref. 31]

The **client/server model** is a description of the interaction between two entities: the client and the server. A *client* makes requests of the server to perform services. A *server* provides a set of services upon request. The ways in which the client can interact with the server are described by a *contract*: a specification of the requests that can be made of the server by the client. Both must fulfill the contract, the client by making only those requests specified, and the server by responding to those requests. [Ref. 31]

Thus, the **responsibilities** of an object are the services that it provides for the contracts it supports. In the terminology used in this thesis, the responsibilities of an object are represented by its methods.

In object-oriented programming, both client and server are either classes or instances of classes. Any object can act as either a client or a server at any given time. The advantage of the client/server model is that it focuses on *what* the server does for the

client, rather than *how* the server does it. The implementation of the server is encapsulated (locked away from the client). [Ref. 31]

Responsibility-driven design is inspired by the client/server model. It focuses on the contract by asking the two following questions [Ref. 31]:

- What actions is this object responsible for?
- What information does this object share?

An important point is that information shared by an object may or may not be part of the structure of that object. That is, the object could store the information, compute it, or delegate the request to another object. [Ref. 31]

Encapsulation is compromised when the structural details of an object become part of the interface to that object. This can only occur if the designer uses knowledge of those structural details. The strength of the responsibility-driven design is that it maximizes encapsulation when the designer intentionally ignores the structural details of an object. [Ref. 31]

3. Data-Driven versus Responsibility-Driven Design

The data-driven approach to object-oriented design focuses on the structure of the data. This results in the incorporation of structural information in the definitions of classes. Doing so violates encapsulation. [Ref. 31]

On the other hand, the responsibility-driven approach focuses on the contractual responsibilities of a class, emphasizing the encapsulation of both the structure and

behavior of objects. Object structure is then specified during the implementation phase.
[Ref. 31]

The reason a data-driven design violates encapsulation is that the variables defined for a class are explicitly declared in the class definition, which is public. In some systems, this permits these variables to be accessed directly, rather than only being accessed by means of methods.

In addition, in this author's point of view, given a set of objects, the class hierarchy that would result from a design based on a data-driven approach would be similar, or even identical, to the one that would result from a design based on a responsibility-driven approach. This is so because, in the responsibility-driven approach, an object's responsibilities include two key items [Ref. 6:p. 61]:

- the actions the object can perform, and
- the knowledge the object maintains.

In a data-driven approach, the actions the object can perform are represented by its methods and the knowledge the object maintains are represented by its variables. In a responsibility-driven approach, these two items are represented by the object's methods (responsibilities).

Thus, in both approaches the final set of objects is essentially the same. They are just represented differently. Therefore, using one approach or the other, the organization of these objects in a hierarchy are likely to produce the similar results. However, the responsibility-driven approach could provide better encapsulation.

4. Tactical Database Design Approach

In the design of the Tactical Database, a data-driven approach was used. That is, class structures were explicitly defined during the design phase. Therefore, the objects (classes) defined for the Tactical Database are composed of variables and methods (if a responsibility-driven approach were used, the initial version of the objects would have been composed only of methods).

In order to improve encapsulation, two methods were defined for every variable: one to access the variable (GET "VARIABLE") and one to update the variable (SET "VARIABLE").

In the case of the Tactical Database, a data-driven approach was considered more appropriate for two reasons:

- a design based on a data-driven approach is more readable, and
- some database performance constraints might be achieved only by violating the encapsulation of the objects.

A design based on a data-driven approach is more readable because the variables and the methods that form an object are shown explicitly in its class definition. This allows the reader to distinguish promptly "the knowledge an object maintains" (its variables) from "the actions this object can perform" (its methods). Readability is especially important when the designer and the implementor are not the same person (or team), which is the case of the Tactical Database.

Even though encapsulation provides better software maintainability, it may result in a reduction of the system overall performance. In situations which require short

response times, it is sometimes more efficient to access a variable directly, rather than accessing it by means of one or more procedure (method) calls. Of course, the designer has to be aware of the consequences of sacrificing maintainability to obtain better performance.

Despite the fact that the design of the Tactical Database has been based on a data-driven approach, the same implementation could also have been arrived at by using a responsibility-driven approach. This could be achieved by defining every variable of a class inside a method (such as "GET VARIABLE"). Consequently, these variables could no longer be accessed directly. This would improve encapsulation, and the resultant class would have no variables declared in its definition, only methods. This is basically the result of a design based on a responsibility-driven approach.

B. TACTICAL DATABASE ISSUES

1. Object Space

The number of tracks and special points stored in the Tactical Database shall be constrained only by physical storage limits of the system [Ref. 1:p. 33]. A GemStone system can support 2^{32} objects and an object can have up to 2^{31} instance variables [Ref.16]. Therefore, this requirement can be fulfilled by GemStone.

2. Performance

The Tactical Database overall performance is a result of both software and hardware performances.

According to [Ref. 1:p. 33], all tracks shall be updated at least every four seconds, and the system shall support up to 1000 active tracks (or special points) displayed on the TacPlot. Also, the response time for menu selections, track information requests, and tactical display aids shall be less than one second.

Considering that up to 1000 tracks (or special points) may be stored in the Tactical Database, an update time of four milliseconds per track (or special point) would be required. If every track (or special point) kept in secondary storage (hard disk) is accessed separately, then the system will require a disk access time of at most two milliseconds, since an update operation requires two disk accesses: one access to bring the object into main memory and one to write the object back to disk. However, through the use of clustering, it is possible to update an entire group of objects with only two disk accesses, instead of two disk accesses per object. This would improve the database performance. Several recommendations as to how to cluster the classes in the Tactical Database were made in Chapter V, Section A.2. In addition, GemStone provides an indexed associative access mechanism which could be used to improve the efficiency in retrieving collections of objects from the database.

Note that the requirement that the system shall support up to 1000 active tracks (or special points) displayed on the TacPlot is, in this author's opinion, overkill in that the display of 1000 symbols would clutter up the screen, making it extremely difficult for the LCCDS operator to operate the system properly.

Another fact that might influence the database performance is the concurrency control mechanism. The Tactical Database will be shared by four interfaces: Track

System, Data Link System, Navigation System, and Man-Machine Interface. Except for the Man-Machine Interface, these interfaces will be sending data automatically to the database every four seconds. In order to reduce the number of possible transaction conflicts, it is recommended that these three interfaces be synchronized in a round robin schedule.

3. Display Doctrine

As mentioned in Chapter II, Section B.5, the LCCDS shall provide, via a Display Doctrine, the capability for user defined conditional statements, in IF-THEN form, which control data filtering and specify presentation parameters for displayed data. The Display Doctrine shall, as a minimum, apply to and operate on the following types of displayed information:

- All tracks qualifying for display.
- All tracks of a particular track category qualifying for display.
- Any single track or special point identified using the Hook function.

Rather than defining methods to display the requested information, it is recommended that the Display Doctrine requirement be achieved by formulating queries using the GemStone query protocol.

4. Menus

The method GET MENU was defined for the class TDB OBJECT and redefined for each of the classes in the Tactical Database. The purpose of this set of methods is to allow the user to access any part of the database without having to memorize the class

hierarchy in which the TDB was organized and also without having to memorize the structure of each class (i.e., which variables and methods were defined for each class).

Therefore, for each class, GET MENU will present a list of options which include that class's methods and subclasses. By selecting one of the options, the user will be able to either execute a method or access a subclass of that class. When a subclass is selected, another menu containing a list of similar options for the subclass is presented.

In order to reduce the time to access the TDB, these menus could be made part of the Man-Machine Interface (MMI) software. However, this would imply that the MMI designer would have to be aware of the organization of the TDB. Also, any modifications in the TDB would require modifications in the MMI. This is a violation of encapsulation between two interacting software programs. The LCCDS implementer will have to decide if the performance improvement obtained by this violation of encapsulation is worth the reduction in maintainability.

C. FUTURE IMPROVEMENTS

1. Multiple Users

One of the possible improvements in the LCCDS is the replacement of the system CPU by networked workstations [Ref. 1:p. 213]. Since GemStone is a multi-user OODBMS, the only implication of this modification would be the introduction of additional interfaces to allow several operators to operate the LCCDS simultaneously.

However, since several operators would then be using the Tactical Database simultaneously, it is recommended to assign one of them the function of database

administrator. Only the database administrator would have authorization to execute certain sensitive operations such as garbage collection and database backups.

2. Automatic Multiple CPA Function

The CPA (Closest Point of Approach) function specified for the basic LCCDS calculates the CPA of a track on user request. It issues a warning of a Close CPA or Collision CPA as appropriate when the track is within certain specified time and range criteria. A possible improvement is the Automatic Multiple CPA Function. This Function would automatically calculate CPA values for all surface tracks within a specified range and issue warnings when the criteria are met. It should also be possible to enable or disable this function. [Ref. 1:p. 209]

The Automatic Multiple CPA Function could be added to the Tactical Database by defining, for the class SURFACE TRACK, the method AUTOMATIC CPA PROCESSING. This method would be called every time a surface track was updated. The method would be responsible for checking the range (distance) from the track to Ownship and, if the track was within the specified range, then a call to the method CPA PROCESSING (which was defined for the class TDB OBJECT and, consequently, inherited by all classes of the Tactical Database) would be executed and a warning issued. If the track was not within the specified range, then CPA PROCESSING would not be called.

D. FINAL COMMENTS

The two most desired characteristics in a database are data reliability and good performance. Data reliability depends directly on the database mechanisms to ensure

security and data integrity, while performance depends on three factors: database design; database management system; and system hardware.

1. Database Security and Data Integrity

In the Tactical Database, in principle, database security is not a critical factor, since all individuals that could have access to the database are typically members of the ship's crew, which are supposed to be trustworthy people.

The integrity of the database is maintained, as much as possible, by means of the concurrency control and crash recovery mechanisms. While the concurrency control mechanism depends entirely on the OODBMS used, it is possible to improve the crash recovery mechanism by using two distinct hard disks: one for the data itself and one for database backups. This is a simple and cost effective way of reducing the risk of loss of data due to hard crashes. Ideally, for the Tactical Database, at least two hard disks should be used.

2. Performance

Good performance is absolutely necessary for the Tactical Database. One of the factors that affect performance is the database design; and a good design depends not only on the designer's skills, but also on the facilities provided by the OODBMS. GemStone provides only single inheritance. Ideally, an OODBMS should also provide multiple and selective inheritance. These two features together would make the design easier, resulting in simpler class hierarchies.

The clustering techniques provided by the system also greatly influence the overall database performance. Ideally, an OODBMS should allow the user to create clusters of classes, of instances, or both classes and instances. GemStone provides these facilities.

Another factor that affects the database performance is the system hardware. A relatively recent advance in technology are multi-backend databases. In these systems, the database is divided into parts and each part is processed independently and stored in a separate hard disk. A central processor is responsible for the integration of the parts. Since different portions of the database can be accessed in parallel, these systems tend to reduce the database average access time. Ideally, the system used for the Tactical Database should be a multi-backend OODBMS. At this time, however, no commercial multi-backend OODBMS is available.

E. CONCLUSION

This thesis consists of three main parts: the design methodology (Chapter III), the design itself (Chapter IV), and the additional design considerations (Chapter V).

The goal of the design methodology proposed (Chapter III) is to provide a simple and systematic way of approaching the problems of object-oriented design, focusing on the production of a design that is, as much as possible, independent of the specific language used for implementation. This gives the implementer much freedom in choosing the OODBMS to be used.

The design itself (Chapter IV) resulted in the definition of a set of classes organized in a hierarchy, which are intended to serve as the basis for the implementation of the

Tactical Database for the LCCDS. A data-driven design approach was used in order to give the reader a better understanding of the structure of each object (class). However, the implementation could have been produced utilizing a responsibility-driven approach without changes in the class hierarchy.

The additional design considerations (Chapter V), even though not considered an integral part of the design methodology, directly affect the two most desired characteristics in a database: data reliability and good performance. These additional considerations complement the design and should not be neglected during the implementation phase.

Finally, it is hoped that the design proposed is clear enough to facilitate the implementation, which is a tough and exhaustive phase of a software development cycle.

APPENDIX

A. TACTICAL DATABASE CLASS HIERARCHY

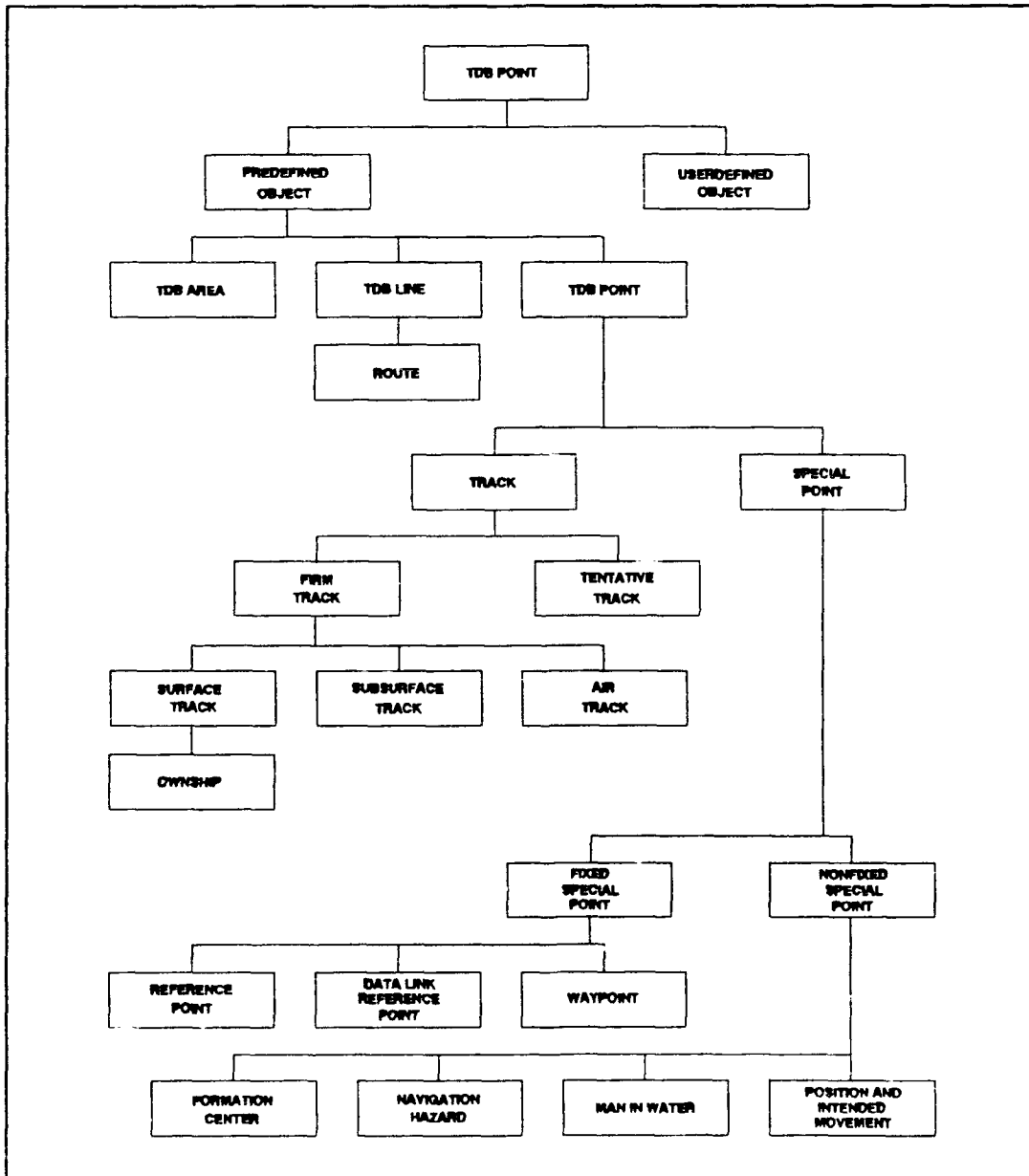


Figure 8: The Hierarchy of Classes Defined for the TACTICAL DATABASE

B. CLASS REPRESENTATION FORMAT

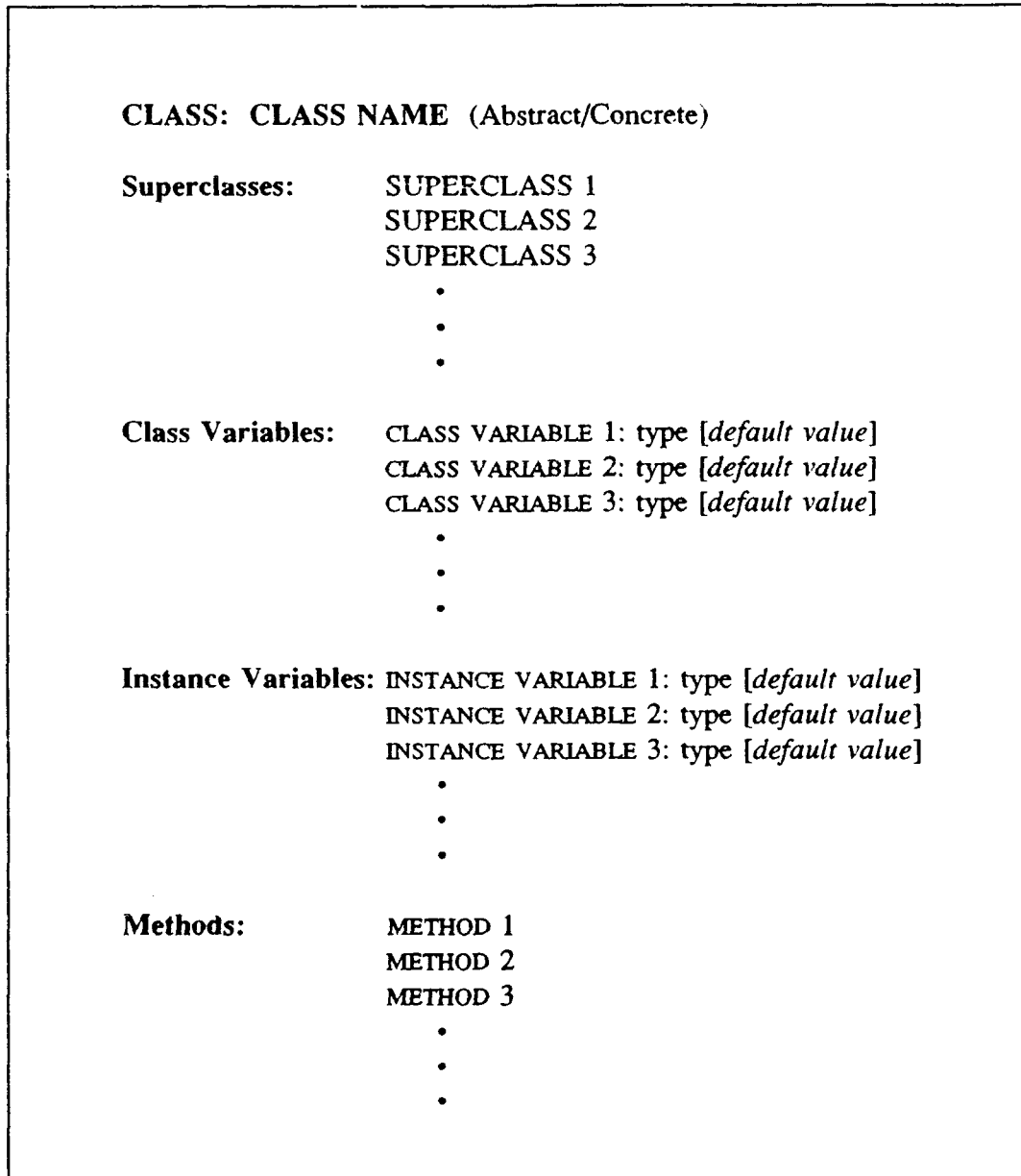


Figure 9: The Class Definition Format

Note that the addition of the term (abstract/concrete) after **CLASS NAME** indicates whether or not the class was designed to be instantiated. Even though this information is

not strictly necessary for the class definition, it helps the user understand the role played by the class in the class hierarchy.

Note also the convention that class names are written in **CAPITALS**, variable and method names are written in **SMALL CAPITALS**, variable types are written in normal font, and variable values are written in *italics*. This form of representation will be used in this thesis.

C. CLASSES DEFINED FOR THE TACTICAL DATABASE

CLASS: ANGLE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: DEGREE: (0..359) [0]
MINUTE: (0..59) [0]
SECOND: (0..59) [0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET DEGREE
SET DEGREE
GET MINUTE
SET MINUTE
GET SECOND
SET SECOND

CLASS: LATITUDE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: ANGLE: ANGLE [0:0:0]
HEMISPHERE: (N, S) [N]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET ANGLE
SET ANGLE
GET HEMISPHERE
SET HEMISPHERE

CLASS: LONGITUDE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: ANGLE: ANGLE [0:0:0]
HEMISPHERE: (E, W) [W]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET ANGLE
SET ANGLE
GET HEMISPHERE
SET HEMISPHERE

CLASS: GEOGRAPHICAL POSITION (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: LATITUDE: LATITUDE [0:0:0:N]
LONGITUDE: LONGITUDE [0:0:0:W]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET LATITUDE
SET LATITUDE
GET LONGITUDE
SET LONGITUDE

CLASS: DIRECTION (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: ANGLE: ANGLE [0:0:0]
REFERENCE NORTH: {T, M} [T]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET ANGLE
SET ANGLE
GET REFERENCE NORTH
SET REFERENCE NORTH

CLASS: RELATIVE POSITION (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: BEARING: DIRECTION [0:0:0:T]
RANGE: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET BEARING
SET BEARING
GET RANGE
SET RANGE

CLASS: WIND (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: DIRECTION: DIRECTION [0:0:0:T]
SPEED: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET DIRECTION
SET DIRECTION
GET SPEED
SET SPEED

CLASS: MAGNETIC VARIATION (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: ANGLE: ANGLE [0:0:0]
HEMISPHERE: (E, W) [W]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET ANGLE
SET ANGLE
GET HEMISPHERE
SET HEMISPHERE

CLASS: HMS (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: HOUR: (00..23) [00]
MINUTE: (00..59) [00]
SECOND: (00..59) [00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET HOUR
SET HOUR
GET MINUTE
SET MINUTE
GET SECOND
SET SECOND

CLASS: TIME (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: LOCAL TIME: HMS [00:00:00]
TIME ZONE: (A..Z) [Z]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET LOCAL TIME
SET LOCAL TIME
GET TIME ZONE
SET TIME ZONE

CLASS: DISTANCE RANGE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: MINIMUM DISTANCE: real [0.0]
MAXIMUM DISTANCE: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET MINIMUM DISTANCE
SET MINIMUM DISTANCE
GET MAXIMUM DISTANCE
SET MAXIMUM DISTANCE

CLASS: TIME RANGE (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: MINIMUM TIME: HMS [00:00:00]
MAXIMUM TIME: HMS [00:00:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET MINIMUM TIME
SET MINIMUM TIME
GET MAXIMUM TIME
SET MAXIMUM TIME

CLASS: CLOSE CPA (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: DISTANCE RANGE: DISTANCE RANGE [200:9999]
TIME RANGE: TIME RANGE [00:05:00:01:39:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET DISTANCE RANGE
SET DISTANCE RANGE
GET TIME RANGE
SET TIME RANGE

CLASS: COLLISION CPA (Concrete)

Superclasses: None

Class Variables: None

Instance Variables: DISTANCE RANGE: DISTANCE RANGE [1:199]
TIME RANGE: TIME RANGE [00:05:00:01:39:00]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET DISTANCE RANGE
SET DISTANCE RANGE
GET TIME RANGE
SET TIME RANGE

CLASS: TDB OBJECT (Abstract)

Superclasses: None

Class Variables: None

Instance Variables: TDB OBJECT NUMBER: integer [0]
ORIGIN: {Local Manual, Local Auto, Remote} [*Local Manual*]
ADDITIONAL INFORMATION: string(30) [*None*]

Methods: GET MENU
CONVERT STRING TO NUMERIC
CONVERT NUMERIC TO STRING
GET TDB OBJECT NUMBER
SET TDB OBJECT NUMBER
GET ORIGIN
SET ORIGIN
GET ADDITIONAL INFORMATION
SET ADDITIONAL INFORMATION

CLASS: USERDEFINED OBJECT (Concrete)

Superclasses: TDB OBJECT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
CREATE NEW CLASS
DEFINE NEW VARIABLE
DEFINE NEW METHOD

CLASS: PREDEFINED OBJECT (Abstract)

Superclasses: TDB OBJECT

Class Variables: None

Instance Variables: CLOSE CPA: CLOSE CPA [200:9999:00:05:00:01:39:00]
COLLISION CPA: COLLISION CPA [1:199:00:05:00:01:39:00]

Methods: GET MENU
CPA PROCESSING
GET CLOSE CPA
SET CLOSE CPA
GET COLLISION CPA
SET COLLISION CPA

CLASS: TDB AREA (Concrete)

Superclasses: PREDEFINED OBJECT

Class Variables: None

Instance Variables: CENTER GEO POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
TIME OF POSITION: TIME [00:00:00:Z]
NORTH LIMIT: GEOGRAPHICAL POSITION [0:0:0:N]
SOUTH LIMIT: GEOGRAPHICAL POSITION [0:0:0:S]
EAST LIMIT: GEOGRAPHICAL POSITION [0:0:0:E]
WEST LIMIT: GEOGRAPHICAL POSITION [0:0:0:W]
IDENTITY: {Hot, Protected, Unprotected} [Unprotected]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
DEAD RECKONING
DRAW TDB AREA
GET CENTER GEO POSITION
SET CENTER GEO POSITION
GET NORTH LIMIT
SET NORTH LIMIT
GET SOUTH LIMIT
SET SOUTH LIMIT
GET EAST LIMIT
SET EAST LIMIT
GET WEST LIMIT
SET WEST LIMIT
GET IDENTITY
SET IDENTITY
GET COURSE
SET COURSE
GET SPEED
SET SPEED

CLASS: TDB LINE (Abstract)

Superclasses: PREDEFINED OBJECT

Class Variables: None

Instance Variables: None

Methods: GET MENU
DRAW TDB LINE

CLASS: ROUTE (Concrete)

Superclasses: TDB LINE

Class Variables: None

Instance Variables: ROUTE NUMBER: (1..6) [1]
WAYPOINTS: array 1..50 of WAYPOINT [0:0:0:N:00:00:00:Z:{1}]
IDENTITY: {Hot, Protected, Unprotected} [Unprotected]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
ROUTE GEOMETRY
GET ROUTE NUMBER
SET ROUTE NUMBER
GET WAYPOINTS
SET WAYPOINTS
GET IDENTITY
SET IDENTITY

CLASS: TDB POINT (Abstract)

Superclasses: TDB OBJECT

Class Variables: None

Instance Variables: TIME OF POSITION: TIME [00:00:00:Z]

Methods:
GET MENU
PLOT TDB POINT
GET TIME OF POSITION
SET TIME OF POSITION

CLASS: TRACK (Abstract)

Superclasses: TDB POINT

Class Variables: None

Instance Variables: RELATIVE POSITION: RELATIVE POSITION [0:0:0:T:0.0]
COURSE: ANGLE [0:0:0]
SPEED: real [0.0]
IDENTITY: {Unknown, Friendly, Hostile} [Unknown]

Methods:
GET MENU
CALCULATE GEOGRAPHICAL POSITION
TRACK COURSE AND SPEED DETERMINATION
DEAD RECKONING
GET RELATIVE POSITION
SET RELATIVE POSITION
GET COURSE
SET COURSE
GET SPEED
SET SPEED
GET IDENTITY
SET IDENTITY

CLASS: TENTATIVE TRACK (Concrete)

Superclasses: TRACK

Class Variables: None

Instance Variables: CATEGORY: {Tent-Air, Tent-Surface, Tent-Subsurface} [*Tent-Air*]

Methods:

- GET MENU
- CREATE INSTANCE
- DELETE INSTANCE
- UPDATE INSTANCE
- DISPLAY INSTANCE
- NEW TRACK ESTABLISHMENT
- INITIAL CATEGORY ASSIGNMENT
- CLASS CHANGE PROCESSING
- GET CATEGORY
- SET CATEGORY

CLASS: FIRM TRACK (Abstract)

Superclasses: TRACK

Class Variables: None

Instance Variables: POSITION NUMBER: integer [0]

Methods:

- GET MENU
- IDENTIFICATION FUNCTION
- TRACK HISTORY PROCESSING
- GET POSITION NUMBER
- SET POSITION NUMBER

CLASS: AIR TRACK (Concrete)

Superclasses: FIRM TRACK

Class Variables: None

Instance Variables: HEIGHT: real [0.0]

Methods:
GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET CATEGORY
GET HEIGHT
SET HEIGHT

CLASS: SURFACE TRACK (Concrete)

Superclasses: FIRM TRACK

Class Variables: None

Instance Variables: None

Methods:
GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET CATEGORY

CLASS: SUBSURFACE TRACK (Concrete)

Superclasses: FIRM TRACK

Class Variables: None

Instance Variables: DEPTH: real [0.0]

Methods:
GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET CATEGORY
GET DEPTH
SET DEPTH

CLASS: OWNSHIP (Concrete)

Superclasses: SURFACE TRACK

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
GREENWICH MEAN TIME: TIME [00:00:00:Z]
JULIAN DATE: (0001..9366) [0001]
MARKTIME: HMS [00:00:00]
MAGNETIC VARIATION: MAGNETIC VARIATION [0:0:0:W]
OWNSHIP SET: DIRECTION [0:0:0:T]
OWNSHIP DRIFT: real [0.0]
WIND INFO: WIND [0:0:0:T:0.0]
NAVIGATION SOURCE: {Inertial, SATNAV, Omega, DR} [DR]

Methods:
GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
MONITOR OWNSHIP POSITION
BEARING AND RANGE FROM POSITION TO POSITION

GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET GREENWICH MEAN TIME
SET GREENWICH MEAN TIME
GET JULIAN DATE
SET JULIAN DATE
GET MARKTIME
SET MARKTIME
GET MAGNETIC VARIATION
SET MAGNETIC VARIATION
GET OWNERSHIP SET
SET OWNERSHIP SET
GET OWNERSHIP DRIFT
SET OWNERSHIP DRIFT
GET WIND INFO
SET WIND INFO
GET NAVIGATION SOURCE
SET NAVIGATION SOURCE

CLASS: SPECIAL POINT (Abstract)

Superclasses: TDB POINT

Class Variables: None

Instance Variables: GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [0:0:0:N]
ORIGIN: {Local Manual, Remote} [*Local Manual*]

Methods: GET MENU
CALCULATE RELATIVE POSITION
GET CATEGORY
GET GEOGRAPHICAL POSITION
SET GEOGRAPHICAL POSITION
GET ORIGIN
SET ORIGIN

CLASS: FIXED SPECIAL POINT (Abstract)

Superclasses: SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU

CLASS: REFERENCE POINT (Concrete)

Superclasses: FIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

CLASS: DATA LINK REFERENCE POINT (Concrete)

Superclasses: FIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE

UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

CLASS: WAYPOINT (Concrete)

Superclasses: FIXED SPECIAL POINT

Class Variables: None

Instance Variables: STEAMING ROUTE: subset of {1,2,3,4,5,6} [{ }]
ORIGIN: {Local Manual} [*Local Manual*]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
OWNSHIP DISTANCE TO POSITION
ESTIMATED TIME OF ARRIVAL AT POSITION
ESTIMATED TIME ENROUTE TO POSITION
GET IDENTITY
GET STEAMING ROUTE
SET STEAMING ROUTE
GET ORIGIN
SET ORIGIN

CLASS: NONFIXED SPECIAL POINT (Abstract)

Superclasses: SPECIAL POINT

Class Variables: None

Instance Variables: COURSE: ANGLE [0:0:0]
SPEED: real [0.0]

Methods: GET MENU
DEAD RECKONING
GET COURSE
SET COURSE
GET SPEED
SET SPEED

CLASS: FORMATION CENTER (Concrete)

Superclasses: NONFIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

CLASS: NAVIGATION HAZARD (Concrete)

Superclasses: NONFIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

CLASS: MAN IN WATER (Concrete)

Superclasses: NONFIXED SPECIAL POINT

Class Variables: None

Instance Variables: None

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
GET IDENTITY

CLASS: POSITION AND INTENDED MOVEMENT (Concrete)

Superclasses: NONFIXED SPECIAL POINT

Class Variables: None

Instance Variables: ORIGIN: {Local Manual} [*Local Manual*]

Methods: GET MENU
CREATE INSTANCE
DELETE INSTANCE
UPDATE INSTANCE
DISPLAY INSTANCE
OWNSHIP DISTANCE TO POSITION
ESTIMATED TIME OF ARRIVAL AT POSITION
ESTIMATED TIME ENROUTE TO POSITION
GET IDENTITY
GET ORIGIN
SET ORIGIN

D. METHODS DEFINED FOR THE TACTICAL DATABASE

The following are the methods that were defined for the Tactical Database:

1. Get Menu

Classes: All classes, either concrete or abstract.

Description: This method presents a list of options for user selection. For a class that has subclasses, the list of options includes that class' subclasses plus the methods defined locally for that class. For a class that does not have subclasses, the list of options includes only the methods defined locally for that class. Since each class has its own structure, this method should be redefined (i.e, defined locally) for each class.

2. Create Instance

Classes: All concrete classes.

Description: This method asks the user to enter the values of the variables defined for the class to which this instance is to belong. After these values are entered, a new instance of the class is created. It is recommended that this method be implemented by means of the OODBMS predefined operations for instance creation. Since each concrete class has its own structure, this method should be redefined (i.e., defined locally) for each concrete class.

3. Delete Instance

Classes: All concrete classes.

Description: This method asks the user to enter the name or identification number of the instance (object) to be deleted. It is recommended that this method be implemented by means of the OODBMS predefined operations for instance deletion. It is also recommended that user confirmation be requested before the deletion is executed. Since each concrete class has its own structure, this method should be redefined (i.e., defined locally) for each concrete class.

4. Update Instance

Classes: All concrete classes.

Description: This method displays a list of the variables (and their respective current values) defined for the instance. The user is allowed to modify any of these values. It is recommended that this method be implemented by means of the OODBMS predefined operations for instance update. Since each concrete class has its own structure, this method should be redefined (i.e., defined locally) for each concrete class.

5. Display Instance

Classes: All concrete classes.

Description: This method displays a list of the variables, and their respective current values, defined for the instance. The user is not allowed to modify these values. It is also recommended that this method be implemented by means of the OODBMS predefined operations for instance display. Since each concrete class has its own structure, this method should be redefined (i.e., defined locally) for each concrete class.

6. Get "Variable"

Classes: All classes for which at least one variable was defined, and all classes for which at least one method was used to store a constant value (rather than defining a class variable).

Description: The term "variable" represents the name of the variable that is to be accessed. This method returns the value of the variable requested.

7. Set "Variable"

Classes: All classes for which at least one variable was defined, and all classes for which at least one method was used to store a constant value (rather than defining a class variable).

Description: The term "variable" represents the name of the variable that is to be updated. This method is generally used in conjunction with the method GET "VARIABLE". The user is allowed to modify the value of the requested variable.

8. CPA Processing

Classes: PREDEFINED OBJECT

Description: This method should access the geographical position, course, and speed of the two instances (objects) and, by means of extrapolation, calculate the new geographical positions in which the instances (objects) will be closest to each other. Information about range, bearing, and time between the objects should also be provided.

This method should use the methods GET CLOSE CPA and GET COLLISION CPA in order to issue notifications when an object meets one of the cpa criteria.

9. Convert String to Numeric

Classes: TDB OBJECT

Description: The method converts a string to numeric format. It is recommended that this method be implemented by means of the operations for type conversion provided by the OODBMS used.

10. Convert Numeric to String

Classes: TDB OBJECT

Description: The method converts a number to string format. It is recommended that this method be implemented by means of the operations for type conversion provided by the OODBMS used.

11. Create New Class

Classes: USERDEFINED OBJECT

Description: The method should allow the user to list the variables and methods that will compose the new class. This method should be used in conjunction with the methods CREATE NEW VARIABLE and CREATE NEW METHOD. It is recommended that this method be implemented by means of the operations for class definition (creation) provided by the OODBMS used.

12. Create New Variable

Classes: USERDEFINED OBJECT

Description: The method should allow the user to specify the variable's name, type, and default value. It is recommended that this method be implemented by means of the operations for variable definition provided by the OODBMS used.

13. Create New Method

Classes: USERDEFINED OBJECT

Description: The method should allow the user to define a method containing any of the operations provided by the OODBMS used. It is recommended that this method be implemented by means of the operations for method definition provided by the OODBMS used.

14. Dead Reckoning

Classes: TDB AREA, TRACK, and NONFIXED SPECIAL POINT.

Description: Based on the input parameters and assuming constant course and speed, the method calculates by extrapolation the geographical position of the object (instance) at the desired time (or after a desired period of time). The user should be allowed to enter either the time of the desired position, or the period of time after which the position is to be calculated.

15. Plot TDB Point

Classes: TDB POINT

Description: The method retrieves the geographical position of the instance requested. This method should interact directly with the Man-Machine Interface, which is responsible for plotting the desired point on the screen.

16. Draw TDB Line

Classes: TDB LINE

Description: The method retrieves the geographical positions of the points that defined the desired line. This method should interact directly with the Man-Machine Interface, which is responsible for drawing the desired line on the screen.

17. Draw TDB Area

Classes: TDB AREA

Description: This method retrieves the values of the variables NORTH LIMIT, SOUTH LIMIT, EAST LIMIT, and WEST LIMIT which are the limits of a rectangular area. Since the contour of an area can be drawn by means of lines, this method can be implemented using the method DRAW LINE. This method should interact directly with the Man-Machine Interface, which is responsible for drawing the desired area on the screen.

18. Calculate Geographical Position

Classes: TRACK

Description: Based on Ownship's geographical position and the relative position of the object (instance) with respect to Ownship, the method calculates the desired object's geographical position.

19. Track Course And Speed Determination

Classes: TRACK

Description: This method should retrieve at least two previous relative positions of the track with respect to Ownship and their respective time of position and, based on these data, calculate the track's speed and course.

20. New Track Establishment

Classes: TENTATIVE TRACK

Description: The method should allow the user to manually enter the new track's relative position, course, and speed. The newly entered track should be initiated as an instance of the class TENTATIVE TRACK, and the value *Unknown* should be assigned to the new track's identity. Another function of this method is to call (activate) the method INITIAL CATEGORY ASSIGNMENT.

21. Initial Category Assignment

Classes: TENTATIVE TRACK

Description: This method should access the value of the variable CATEGORY SPEED and assign the category to the new track according to the following criterion:

- track speed less than category speed: *Surface*
- track speed greater than category speed: *Air*

The preset value of the category speed shall be 50.0 knots.

22. Class Change Processing

Classes: TENTATIVE TRACK

Description: This method should delete an instance of the class TENTATIVE TRACK and create an instance of one of the classes AIR TRACK, SURFACE TRACK, or SUBSURFACE TRACK, as a result of one of the following conditions:

- After three manual updates by the user.
- When the category is entered manually before three position updates.
- When the track is manually declared firm.

23. Track Identification

Classes: FIRM TRACK

Description: This method uses the methods GET IDENTITY and SET IDENTITY in order to access and update the identity of the requested track. The new identity of the track is entered manually by the user.

24. Track History Processing

Classes: FIRM TRACK

Description: This method retrieves the previous positions and respective times of position of the requested track, starting at a time or geographical position specified by the user. This method should interact with the Man-Machine Interface, which is responsible for displaying the retrieved points. It is recommended that this method be implemented by means of the operations for query processing provided by the OODBMS used.

25. Monitor Ownship Position

Classes: OWNSHIP

Description: This method should allow the user to manually enter the Ownship's geographical position, course, speed, time of position, GMT, Julian Date, and marktime.

26. Bearing and Range from Position to Position

Classes: OWNSHIP

Description: Based on the two given geographical positions, this method calculates the range and bearing from the first to the second selected point.

27. Calculate Relative Position

Classes: TDB AREA and SPECIAL POINT

Description: Based on Ownship's geographical position, this method calculates the relative position of the object (instance) with respect to Ownship.

28. Ownship Distance to Position

Classes: WAYPOINT and POSITION AND INTENDED MOVEMENT

Description: Based on the geographical positions of Ownship and the instance, the distance between the two points is calculated.

29. Estimated Time Enroute to Position

Classes: WAYPOINT and POSITION AND INTENDED MOVEMENT

Description: This method can be implemented by means of a call to the method OWNSHIP DISTANCE TO POSITION, whose output is the distance from Ownship to the desired instance. Therefore, based on this distance and on Ownship's speed, the time enroute is calculated.

30. Estimated Time of Arrival at Position

Classes: WAYPOINT and POSITION AND INTENDED MOVEMENT

Description: This method can be implemented by means of a call to the method ESTIMATED TIME ENROUTE TO POSITION, whose output is the time enroute to the desired instance. Based on this time and on the current time, the time of arrival at the instance is calculated.

31. Route Geometry

Classes: ROUTE

Description: For each waypoint that forms the selected route, this method calculates the course at the desired speed that Ownship or a selected track must take in order to intercept the waypoint. Estimated time of arrival at the waypoint and estimated time enroute to the waypoint are also calculated. The route geometry calculation should be updated every four seconds. The estimated times of arrival and estimated times enroute can be calculated by means of the methods ESTIMATED TIME OF ARRIVAL AT POSITION and ESTIMATED TIME ENROUTE TO POSITION, respectively.

LIST OF REFERENCES

1. Seveney, J., and Steinberg, G., *Requirements Analysis for a Low Cost Combat Direction System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990.
2. Department of the Navy, (NAVSEA) 0967-LP-027-8602, *Systems Engineering Handbook Vol.I, Combat Direction System Model 5*, February 1985.
3. Elmasri R., and Navathe, S. B., *Fundamentals of Database Systems*, pp. 442-444, The Benjamin/Cummings Publishing Company, Inc., 1989.
4. Rumbaugh, J., "Relations as Semantic Constructs in an Object-Oriented Language", *Proceedings of OOPSLA '87*, Orlando, FL, October 4-8, 1987.
5. Smith, K. E., and Zdonik, S. B., "Intermedia: A Case Study of the Differences between Relational and Object-Oriented Database Systems", *Proceedings of OOSPLA '87*, Orlando, FL, October 4-8, 1987.
6. Wirfs-Brock, R., Wilkerson, B., and Wiener, L., *Designing Object-Oriented Software*, Prentice-Hall, Inc., 1990.
7. Nelson, M. L., *An Introduction to Object-Oriented Programming*, Naval Postgraduate School, Monterey, CA, Technical Report No NPS52-90-024, Apr 1990.
8. Booch, G., *Software Engineering with Ada*, 2nd ed., The Benjamin/Cummings Publishing Company, Inc., 1986.
9. Coad, P., and Yourdon, E., *Object-Oriented Analysis*, Prentice-Hall, Inc., 1990.
10. Winblad, A. L., Edwards, S. D., and King, D. R., *Object-Oriented Software*, Addison-Wesley Publishing Company, Inc., 1990.
11. Booch, G., *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., 1991.
12. Naval Postgraduate School, Monterey, CA, Technical Report No NPSCS-91-007, *An Object-Oriented Design Methodology*, by De Paula, E. G., and Nelson, M. L., December 1990.

13. Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, Inc., 1988.
14. Shlaer, S., and Mellor, S., *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall, Inc., 1988.
15. Ross, R., *Entity Modeling: Techniques and Application*, Database Research Group, Boston, MA, 1987.
16. Bretl, R., and others, "The GemStone Data Management System", in *Object-Oriented Concepts, Databases, and Applications*, Kim W., and Lochovsky, F. H., pp. 283-308, ACM Press/Addison-Wesley Publishing Company, Reading, MA, 1989.
17. Borland International, Inc., *Turbo Pascal 5.5: Object-Oriented Programming Guide*, 1988.
18. Banerjee, J., and others, "Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Information Systems*, v. 5, No. 1, pp. 3-26, January 1987.
19. Lippman, S., B., *C++ Primer*, Addison-Wesley Publishing Company, 1989.
20. Ontologic, Inc., *Vbase Integrated Object Database: System Documentation, Releases 0.8-1.0*, 1988.
21. Ontologic, Inc., *ONTOS Object Database Documentation, Release 1.5*, 1990.
22. Snyder, A., "Object-Oriented Programming for Common Lisp", Report ATC-85-1, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA, 1985.
23. Borland International, Inc., *Object-Oriented Programming in the Real World*, Scotts Valley, CA, 1990.
24. Ross, D. L., *Object-Oriented Database Manager for the Low Cost Combat Direction System*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1989.
25. Servio Logic Development Corporation, *Programming in OPAL, Version 1.5*, 1989.
26. Fishman, D. H., and others, "Overview of the Iris DBMS", in *Object-Oriented Concepts, Database, and Applications*, Kim, W., and Lochovsky, F. H., pp. 219-250, ACM Press/Addison-Wesley Publishing Company, Reading, MA, 1989.

27. Carey, M. J., and others, "Storage Management for Objects in EXODUS", in *Object-Oriented Concepts, Database, and Applications*, Kim, W., and Lochovsky, F. H., pp. 341-369, ACM Press/Addison-Wesley Publishing Company, Reading, MA, 1989.
28. Kim, W., and others, "Features of the ORION Object-Oriented Database System", in *Object-Oriented Concepts, Database, and Applications*, Kim, W., and Lochovsky, F. H., pp. 251-282, ACM Press/Addison-Wesley Publishing Company, Reading, MA, 1989.
29. Bjornerstedt, A., and Hulten, C., "Version Control in an Object-Oriented Architecture", in *Object-Oriented Concepts, Database, and Applications*, Kim, W., and Lochovsky, F. H., pp. 451-485, ACM Press/Addison-Wesley Publishing Company, Reading, MA, 1989.
30. Korth, H. F., and Silberschatz, A., *Database System Concepts*, McGraw-Hill, Inc., 1986.
31. Wirfs-Brock, R., and Wilkerson, B., "Object-Oriented Design: A Responsibility-Driven Approach", Proceedings of OOPSLA '89, New Orleans, LA, October 1-6, 1989.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, VA, 22304-6145
2. Library, Code 52 2
Naval Postgraduate School
Monterey, CA 93943-5002
3. Chief of Naval Research 1
800 Quincy Street
Arlington, VA 22217
4. Center for Naval Analysis 1
4401 Ford Avenue
Alexandria, VA 22302-0268
5. Naval Sea Systems Command 1
Attn: LCDR Scott Kelly
Code 06D3131
Washington, D.C. 20362-5101
6. Naval Sea Systems Command 1
Attn: William Wilder
PMS - 412
Washington, D.C. 20362-5101
7. Chairman, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100
8. Dr. Luqi 3
Code CS/LQ
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943