ARMY RESEARCH LABORATORY

IChart: A Graphical Tool to View and Manipulate Force Management Structure Databases

by Frederick S. Brundick, George W. Hartwig, Jr., and Samuel C. Chamberlain

ARL-TR-4610 **September 2008**

Approved for public release; distribution is unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-4610 **September 2008**

IChart: A Graphical Tool to View and Manipulate Force Management Structure Databases

Frederick S. Brundick, George W. Hartwig, Jr., and Samuel C. Chamberlain
Computational and Information Sciences Directorate, ARL

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE					*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**					
1. REPORT DATE *(DD-MM-YYYY)* September 2008	**2. REPORT TYPE** Final				**3. DATES COVERED (From - To)** October 2003–October 2004
4. TITLE AND SUBTITLE IChart: A Graphical Tool to View and Manipulate Force Management Structure Databases					**5a. CONTRACT NUMBER**
					5b. GRANT NUMBER
					5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Frederick S. Brundick, George W. Hartwig, Jr., and Samuel C. Chamberlain					**5d. PROJECT NUMBER** 8TV0VC
					5e. TASK NUMBER
					5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: AMSRD-ARL-CI-CT Aberdeen Proving Ground, MD 21005-5067					**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-4610
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)					**10. SPONSOR/MONITOR'S ACRONYM(S)**
					11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Maintaining a high-resolution force structure that may be shared among various organizations is a difficult task. One approach is to design databases which store Global Force Management (GFM) data in the form of time-based tree graphs, using enterprise identifiers (EIDs) as unique surrogate keys. This report is the manual, users' guide, and general documentation for the IChart application, which is intended to be a guide and demonstration of the utility of the GFM Force Structure Construct, to include EIDs and time-based tree graphs. IChart is written in Java and communicates with a MySQL database server. A glossary is included to explain common object-oriented programming and database terms and concepts.					
15. SUBJECT TERMS TO&E, Java, SQL, database, GUI					
16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Frederick S. Brundick
a. REPORT UNCLASSIFIED	**b. ABSTRACT** UNCLASSIFIED	**c. THIS PAGE** UNCLASSIFIED	UL	66	**19b. TELEPHONE NUMBER** (*Include area code*) 410-278-8943

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures vi

List of Tables vii

Acknowledgments viii

1. Introduction 1

2. Database Schema 1

3. Java Classes 6

3.1 General Discussion 6
3.2 Classes and Elements Related to Database Tables 8
3.2.1 BasicEID and SimpleEID Classes 8
3.2.2 OrgType and Similar Classes 9
3.2.3 BasicAssoc Class and Subclasses 10
3.2.4 Other Classes 11
3.3 Database Methods 12
3.4 Display and Edit Methods and Classes 12
3.4.1 Panels and Dialogs 13
3.4.2 Inspectors 15
3.5 OopDatabase Class 16
3.5.1 JDBC and SQL Methods 17
3.5.2 Hash Table Methods 18
3.5.3 Tree Methods 19
3.5.4 Cache Access Methods 19
3.5.5 Other Methods 19
3.6 Tree Component Classes 20
3.6.1 Background 20

3.6.2 Node Classes 20
3.6.3 TreePanel Class 21
3.6.4 NodePanel Class 22
3.6.5 Displaying a Tree 22
3.6.6 Tree Positioning Algorithm 22
3.6.7 Node Drawing Details 24
3.6.8 Tree Rendering Details 25
3.7 Application Classes 26
3.7.1 Overview 26
3.7.2 IChart Class 26
3.7.3 DetailPanel and MultiItem Classes 28
3.7.4 AssocEditor Class 29

4. Users' Guide **30**
4.1 Introduction 30
4.2 Getting Started 32
4.2.1 Property Files 32
4.2.2 Initial Database 33
4.3 Menu Bar 33
4.3.1 Overview 33
4.3.2 File Menu 33
4.3.3 Find Menu 35
4.3.4 Show Menu 35
4.3.5 Help Menu 36
4.4 Organization Tree Panel 36
4.4.1 Overview 36
4.4.2 Display Popup Menu 36
4.4.3 Edit Popup Menu 37
4.5 Association Editor 40

5. Future Development **41**

6. Conclusion **42**

7. References **43**

Appendix A. Property Files **45**

Appendix B. IChart Installation **47**

Glossary **50**

Distribution List **53**

List of Figures

Figure 1. Table links . . . 6
Figure 2. Table class hierarchy . . . 7
Figure 3. Organization type display dialog . . . 13
Figure 4. Organization type editable dialog . . . 13
Figure 5. Person type display dialog . . . 14
Figure 6. Person type editable dialog . . . 14
Figure 7. Organization type/materiel type editable dialog . . . 15
Figure 8. Materiel type inspector . . . 16
Figure 9. JDBC properties dialog . . . 17
Figure 10. Representative tree diagram . . . 23
Figure 11. Order of position calculations . . . 24
Figure 12. Typical node with connecting lines . . . 25
Figure 13. Tree with first phase of connecting lines . . . 26
Figure 14. Tree with final connecting lines . . . 27
Figure 15. Application class hierarchy . . . 28
Figure 16. IChart application main window . . . 31
Figure 17. Association editor window . . . 32
Figure 18. Sample tool tip . . . 33
Figure 19. Find name selection dialog . . . 35
Figure 20. Partially collapsed tree . . . 37
Figure 21. Edit popup menu for an organization type node . . . 38
Figure 22. Subtree containing a multiplier . . . 38
Figure 23. Adding a node to the tree . . . 39
Figure 24. Subtree containing duplicate nodes . . . 40
Figure 25. Association editor type inspectors . . . 41

List of Tables

Table 1. Organization type 2
Table 2. Materiel type 2
Table 3. Skill type 3
Table 4. Person type 3
Table 5. Organization type associations 4
Table 6. Organization 4
Table 7. Organization/org type association 4
Table 8. Organization type relation 5
Table 9. Organization relation 5
Table 10. Database tables 6
Table A-1. Contents of file default.properties 46
Table A-2. Contents of file ichart.properties 46
Table B-1. Files in primary directory 48
Table B-2. Directory structure 48

Acknowledgments

The authors would like to thank Luke Johnson, a high school student in the Science and Engineering Apprentice Program, for beta testing the IChart program by constructing a database after reading the Users' Guide section of this report. He found logic errors in the program and made several suggestions for improving it, all of which were implemented.

1. Introduction

This report is the manual, users' guide, and general documentation for the IChart application. The program is intended to be a guide and demonstration of the utility of the Global Force Management (GFM) Force Structure Construct, to include enterprise identifiers (EIDs) and time-based tree graphs. The tool, while not necessarily a model of a fieldable application, allows the user to rapidly enter GFM data. The program was written with software reuse in mind, and the components were kept general and may be used to construct various EID-based applications.

IChart is written in Java 1.4 (*1*) and communicates with a MySQL 4.0 database server (*2*) using Java Database Connectivity (JDBC) for portability. The driver is MySQL Connector/J 3.0 (*3*). All three of these (free) packages* are available for all major architectures. During the development and testing phase, the MySQL server ran on a Linux laptop while the Java code was written and tested on SGI and Sun workstations and was also tested on a Windows 2000 desktop PC. New releases were put on a Windows 2000 laptop that had its own copy of MySQL and the sample databases.

The database portions of the code were kept isolated as much as possible to facilitate porting to other relation database management system (RDBMS) programs. We used no special MySQL extensions to structured query language (SQL) to further avoid any package-specific dependencies.

A glossary is included in this report to explain common object-oriented programming (*4*) and database terms and concepts. Appendix B contains IChart installation instructions.

2. Database Schema

The database we used is loosely based on the Command and Control Information Exchange Data Model (C2IEDM) (*5*). We ignored some of the intermediate abstraction layers because those values were constants in our context, although there is no reason why more fields could not be added to the tables. Below are tables describing each of our database tables, beginning with the four "basic" types in tables 1–4.

Notice that every table starts with a field named **EID** (*6*). This is an implementation of the fundamental concept of EIDs—every record of every table in every database on every system has a unique identifier in a common format. This enables applications to easily reference each other's data by exchanging the EIDs and does not require them to know the details of the foreign schema. It also provides a surrogate key for the mundane task of associating records in one table of a database with other tables in the same database (i.e., performing an SQL *join* operation).

*JDBC is part of the standard Java distribution.

An EID is a 64-bit value, which in MySQL corresponds to a bigint or int8. It is logically divided into two portions, which we may ignore for the purposes of this application. We thus consider an EID to be a single value which we display in hexadecimal.

The other fields that appear in every table are the start and terminal date/time groups (**s_dtg** and **t_dtg**). The database contains current, archival, and future data. By defining a date/time interval for every record, the user may select the information that pertains to a given date of interest (*7*). SQL has date and time fields, but it is very inconvenient to work with a date/time group that is broken into two parts. MySQL provides a datetime type but it is not a standard type. We chose to use timestamp for our date/time groups. It uses the standard UNIX epoch of 1 Jan 1970 through 31 Dec 2037. This is certainly adequate for the data we will be dealing with. The Java class **Timestamp** is supported by JDBC and allows us to easily manipulate SQL timestamps.

The initial version of the application always uses a time of 00:00:00. It is important to note that this is not midnight, but the beginning of the day. A time interval includes the start date/time but excludes the terminal date/time, or $s_dtg \leq date_time < t_dtg$. It is much clearer to state that an interval is from 1 Oct 2003 up to (but not including) 1 Jan 2004 and a second interval is from 1 Jan 2004 to 1 Apr 2004 than it is to say the first interval is from 1 Oct 2003 00:00:00 through 31 Dec 2003 23:59:59.

All text fields are the maximum allowable size of 255 characters. This is almost certainly too large, but for our small database we could afford to be extravagant. A few of the tables contain a multiplier field which is an integer or int (or int4).

Table 1. Organization type.

Field Name	Type
EID	bigint
Org_Type_Long_Name	char(255)
Org_Type_Short_Name	char(255)
Org_Type_Category	char(255)
s_dtg	timestamp
t_dtg	timestamp

Table 2. Materiel type.

Field Name	Type
EID	bigint
Mat_Type_LIN	char(255)
Mat_Type_Name	char(255)
s_dtg	timestamp
t_dtg	timestamp

The basic type tables are entirely self-contained except for the **person type** table which contains bigints (EIDs) that refer to records in the **skill type** table. This is explained in more detail later with sample data.

Table 3. Skill type.

Field Name	Type
EID	bigint
Skill_Type_Attribute_Name	char(255)
Skill_Type_Attribute_Code	char(255)
Skill_Type_Attribute_Text	char(255)
Skill_Type_Attribute_Remarks	char(255)
s_dtg	timestamp
t_dtg	timestamp

Table 4. Person type.

Field Name	Type
EID	bigint
Person_Type_Rank_EID	bigint
Person_Type_Grade_EID	bigint
Person_Type_Primary_Occupation_EID	bigint
Person_Type_Secondary_Occupation_EID	bigint
Person_Type_Skill_Level_EID	bigint
Person_Type_Remarks	char(255)
s_dtg	timestamp
t_dtg	timestamp

Each **organization type** may have various **materiel**, **skill**, and **person type**s associated with it. The restrictions imposed on such associations are beyond the scope of this report and are not enforced by the program at this time. The associations all follow the same basic form and contain a pair of EIDs, a multiplier to indicate how many of the associated type are required (e.g., a crew may need three camouflage nets), and a remarks field as shown in table 5.

The tables described so far have all been *type* definitions or associations. A type may be thought of as a template or generic description. For example, an organization type may be a platoon, a tank crew, or a company commander. A materiel type appears to be more specific since it contains a LIN (line item number), but it does not contain the serial number of an actual piece of materiel. Skill and person types function in a similar manner.

The database we constructed contains one table of concrete items, namely organizations. To continue with the earlier example, specific organizations could be `1st Platoon/A Co/1-67 AR BN`, `Tank 1/A Sec/1st Plt/A Co/1-67 AR BN`, and `CO/A Co/1-67 AR BN`. We did not implement tables to hold concrete materiel items (e.g., a specific vehicle with a certain bumper number) or individuals in billets (i.e., a real person with a social security number). Such details are both beyond the scope of this project and cause the database to become classified.

Every organization has a link to an organization type, which in turn provides links to the materiel, skill, and person types. It is possible that an organization's type remains constant over a given

Table 5. Organization type associations.

Field Name	Type
EID	bigint
Org_Type_EID	bigint
Org_T_*XXX*_T_Multiplier*	int
*XXX*_Type_EID*	bigint
Org_T_*XXX*_T_Remarks*	char(255)
s_dtg	timestamp
t_dtg	timestamp

**XXX* is one of Mat, Skill, or Person.

date/time interval, but the materiel that is aligned with the organization changes within that interval. Furthermore, another organization of that type may have a different implementation date. Our solution was to add a modification date/time group (**m_dtg**) to the organization to org type* link. While the m_dtg field is included in the database table, it is not implemented in the program.

The organizations and organization to org type associations are stored in the database tables described in tables 6 and 7.

Table 6. Organization.

Field Name	Type
EID	bigint
Org_Long_Name	char(255)
Org_Short_Name	char(255)
Org_Type_Category	char(255)
s_dtg	timestamp
t_dtg	timestamp

Table 7. Organization/org type association.

Field Name	Type
EID	bigint
Org_EID	bigint
Org_Type_EID	bigint
s_dtg	timestamp
t_dtg	timestamp
m_dtg	timestamp

*In this report, we always spell out the word "organization" when we refer to a concrete unit. The template "organization type" is often shortened to "org type."

The sample application maintains organization charts, which we implemented as trees that describe the organization hierarchy. We use the general term *link* for both *relations*, which refer to parent/child links, and *associations*, which are the horizontal links described earlier. The organization type and organization relation tables are identical except for one additional field in the org type relation table—a multiplier which indicates how many instances of the child type should be instantiated. In the sample database, the organization type **`Tank Company`** is the parent of one **`Company HQ`** and three **`Platoon`**s. The corresponding organization tree defines three distinct platoons, namely **`1st, 2nd,`** and **`3rd Platoon/A Co/1-67 AR BN`**. The organization type and organization relation tables are shown in tables 8 and 9.

Table 8. Organization type relation.

Field Name	Type
EID	bigint
Org_Type_Parent_EID	bigint
Org_Type_Relat_Multiplier	int
Org_Type_Child_EID	bigint
Org_Type_Relat_Role	char(255)
s_dtg	timestamp
t_dtg	timestamp

Table 9. Organization relation.

Field Name	Type
EID	bigint
Org_Parent_EID	bigint
Org_Child_EID	bigint
Org_Relat_Role	char(255)
s_dtg	timestamp
t_dtg	timestamp

The final table, **Seq_Table** (sequence table), contains only a single record, which is the highest EID that has been used. A more sophisticated application may use an EID server (*8*) to obtain EIDs when new records are created, but this approach is sufficient for our purposes. The construction of new EIDs is explained later. The complete list of tables is shown in table 10 along with the MySQL comment that describes each one.

The relationships between the tables are shown in figure 1. The EIDs that are the endpoints of each link are actually stored in the various association and relation tables except for the PersonType→SkillType links. With the exception of link 7, all links originate at the top or left and proceed down or to the right. Links 1 and 3 are the organization and organization type relations (stored in tables **OrgRelat** and **OrgTypeRelat**), while link 2 is the organization to org type association (**OrgOrgTAssoc**). The three organization type associations are represented by

Table 10. Database tables.

Database Name	Description
OrgType	organization type
MatType	materiel type
SkillType	skill type
PersonType	person type
OrgTMatTAssoc	org_type/mat_type association
OrgTSkillTAssoc	org_type/skill_type association
OrgTPersonTAssoc	org_type/person_type association
Org	organization
OrgOrgTAssoc	org/org_type association
OrgTypeRelat	org type relation (tree)
OrgRelat	organization relation (tree)
Seq_Table	EID sequence number

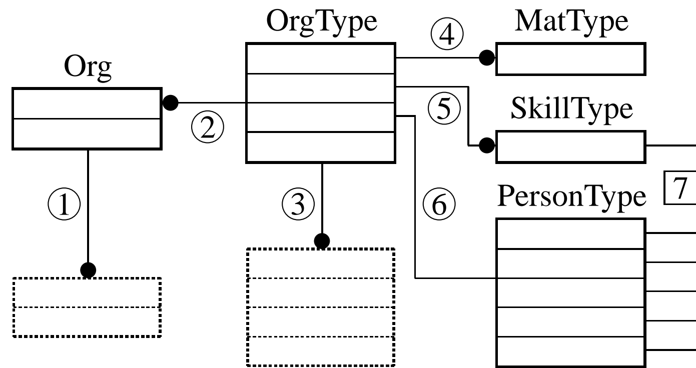


Figure 1. Table links.

links 4–6 (**OrgTMatTAssoc**, **OrgTSkillTAssoc**, and **OrgTPersonTAssoc**). The set of links denoted by 7 is a special case because a person type contains five skill types.

3. Java Classes

3.1 General Discussion

Every SQL table (except Seq_Table) has a corresponding Java class with the same name. Unlike the tables, which contain only data, the classes also contain methods to manipulate the data. Because of the fields and concepts common to the various tables, the classes are not simply copies of the tables. An *abstract* class named **BasicEID** was defined to hold an item's EID and date/time

group interval, act as the superclass of all the classes, and provide common functionality. The database tables described in table 5 are so similar that an abstract class, **BasicAssoc**, was defined to serve as the superclass of the corresponding classes. Likewise, the abstract class **BasicRelat** is the parent of the OrgTypeRelat and OrgRelat classes. The **OrgType** and **Org** classes have the abstract parent **BasicOrg** to make it easier to manipulate them in the organization type and organization trees. The special class **SimpleEID** does not correspond to a database table and is discussed in section 3.2.1. The class hierarchy is shown in figure 2 with the abstract classes in **bold**.

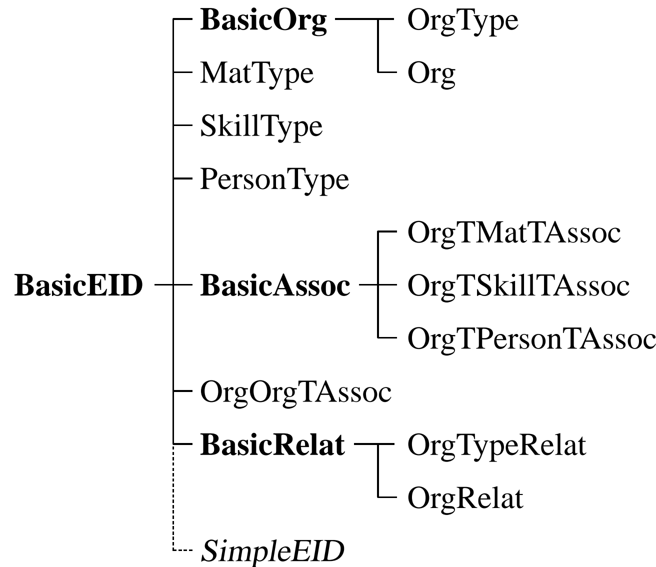


Figure 2. Table class hierarchy.

The mapping between MySQL data types and Java variables is very straightforward. Every time a bigint (EID) appears as a field in a table, it is declared to be a BasicEID in the matching class. A char(255) (text) field becomes a String, a timestamp (date/time group) field is a Timestamp, and an int (integer) is an int.

When a record is read from a table, an object of the appropriate type is instantiated. Storing EIDs as bigints in the database tables is fine for performing SQL join operations, but it is very inefficient when the Java objects are manipulated. Therefore, any EIDs that are found are initially stored as SimpleEIDs. After all of the data has been loaded into memory, the BasicEIDs (except for the first field) are replaced by references to the object that has that EID.*

Every class has the same general members:

- a protected instance variable for each table field,
- constructors,

*Object-oriented programming allows a variable to contain an instance of the declared class or any subclass of it. For example, a BasicEID variable may contain an OrgType, MatType, OrgTSkillTAssoc, or any of the classes in figure 2.

  - a no-argument constructor that uses default values
  - a "normal" constructor that takes a value for each field
  - a constructor that takes a vector of field values
  - a constructor that copies the values from an existing object of the same type
- an accessor and mutator method to get/set each instance variable's value,
- a **toBigString** method that returns a string with the EID, date interval, and at least one field value for debug purposes, and
- methods to:
  - load/store/update the database
  - display/edit an object
  - manipulate a table of these objects.

The basic type classes and organization class also define a **toDescString** method that returns a string with the value that best describes the object. It is used as a label with EID buttons to identify links.

Extensive use has been made of inheritance, overloading, overriding, and polymorphism. Inheritance is the simplest of the concepts; it means a subclass builds on the members (variables and methods) of its superclass. The subclass absorbs the attributes and behaviors of the superclass and adds new capabilities. Overloading is when a class has multiple methods with the same name but different argument lists or *signatures*. An example of this is the four different constructors that are defined in each class. In contrast to this, overriding occurs when two methods with identical signatures are defined in a superclass and a subclass. Suppose that instances of a superclass and subclass are both instantiated and the same method name is invoked. The superclass object of course uses the superclass's definition of the method, while the subclass object overrides that definition and uses its own definition of the method. The superclass method may be thought of as the default action to be performed unless the subclass overrides it. Polymorphism is the process by which Java allows a subclass object to be stored in a superclass variable and automatically overrides methods in a way that is transparent to the programmer. Our classes are highly modular and reuse existing methods—in both the current class and in its superclass—whenever possible.

3.2 Classes and Elements Related to Database Tables

Variables and methods in the various classes fall into three categories: members that mirror the table fields and manipulate the data, members that interface with the database and user, and members that were defined for the needs of the application. We will explain the former members first as defined in the classes listed in figure 2.

3.2.1 BasicEID and SimpleEID Classes

The **BasicEID** abstract class is the general template for EID objects in our application. It has three instance variables that map to table fields—**EID** is a long or 64 bit integer while **s_dtg** and

t_dtg are Timestamps or date/time groups. The no-argument constructor sets the EID to zero, another constructor uses the EID value passed as an argument, and the third accepts the EID, start date/time group, and terminal date/time group. The methods **getEID**, **getSDtg**, and **getTDtg** fetch the appropriate value and **setEID**, **setSDtg**, and **setTDtg** assign a new value.

Every Java class should define a **toString** method that returns a string that describes the object. BasicEID's toString returns the EID as a string of hexadecimal characters and it is used extensively in our application. Every time an object is displayed, its EID is shown using this method. We also wanted a method that would return a descriptive string for debug purposes. Since toString was already being used, we chose to declare an abstract **toBigString** method. The method **getIntervalString** returns the date portion of the object's interval as a string for debug purposes.

We required the ability to compare two EIDs and see if they are equal. The method **equals** accepts an object and compares it to the current object. If the argument is an instance of BasicEID (or one of its subclasses), then the EIDs of the two objects are compared. A method named **isInInterval** determines if the supplied date/time group falls within the interval defined for this object. Another utility method is **hashCode**. It uses the same algorithm as **Long.hashCode** to return an integer that represents the current object. The value is not unique (integers contain only 32 bits compared to the 64 bits in a long variable) but is adequate for use in our application's hash table.

Because BasicEID declares several abstract methods, it is an abstract class and therefore may not be instantiated. We needed to instantiate a minimal EID object for the primary key of every record and as a temporary placeholder in other EID fields. The class **SimpleEID** is our solution to the problem. It defines two constructors that invoke the corresponding BasicEID constructors. We must provide definitions for all of the abstract methods, but at the same time we wanted to make sure that they are never invoked (which would indicate a logic error). The definition for each method therefore throws an exception with a description to indicate which method was invoked.[*]

3.2.2 OrgType and Similar Classes

The **OrgType** class is described as being representative of the **MatType**, **SkillType**, **PersonType**, and **Org** classes. We adopted a naming convention for these five tables and classes:

- field names have the table name as a prefix,
- words in field names are separated by underscores,
- the corresponding Java variable has the same name, except variables use "camel casing,"
- **get**/**set** methods drop the table name prefix, and
- the argument to a **set** method may be abbreviated.

For example, the table **OrgType** is mirrored by the class **OrgType**, its field **Org_Type_Long_Name** becomes the variable **orgTypeLongName**, and the class methods are **getLongName()** and **setLongName(longName)**. Likewise, table **SkillType** becomes class

[*]We use this technique in other classes where an abstract method must be given a definition but the method should never be invoked.

SkillType, field **Skill_Type_Attribute_Name** is the variable **skillTypeAttributeName**, and the methods are **getAttributeName()** and **setAttributeName(attrName)**.

The OrgType class has three string variables that mirror the fields in the OrgType table. Each has a **get*XXX*** and **set*XXX*** method to retrieve or store the value in the current object, respectively. The methods are declared as abstract in the parent class **BasicOrg** so that our application may access the variables in a general manner.

The no-argument constructor sets the EID to zero by indirectly invoking BasicEID's no-argument constructor via BasicOrg's no-argument constructor and stores empty strings in the remaining variables. A constructor which accepts values in the argument list also invokes BasicEID's constructor (via BasicOrg) to store the EID and date/time groups, then stores the rest of the arguments locally. The third constructor takes a vector of values that we assume are in the correct order, while the last constructor extracts the values from a supplied OrgType object. Both of these constructors pass the extracted values to the second constructor to minimize redundant code.

The **toBigString** method returns the EID in hexadecimal (by invoking BasicEID's toString method) concatenated with the organization type's long name and the date/time group interval, while **toDescString** returns the short name.

All linking EID fields, such as the ones in PersonType, are handled in a special way as explained in the next section.

3.2.3 BasicAssoc Class and Subclasses

The abstract **BasicAssoc** class is both representative of the classes that contain EIDs that are used as links and is the superclass of the three organization type association classes. The integer **multiplier** is like the variables described earlier and has **getMultiplier** and **setMultiplier** methods. Likewise, the **remarks** string variable is accessed via **getRemarks** and **setRemarks**.

EIDs are handled differently because they are of type `BasicEID`. The no-argument constructor sets the EID of the new object to zero by explicitly invoking BasicEID's no-argument constructor, sets the multiplier to one, and stores an empty string in the remarks variable. The default values for **orgTypeEID** and **assocTypeEID** are assigned by instantiating new SimpleEID objects with that class's no-argument constructor.

The next two constructors act the same way except the new values are obtained as separate arguments or in a vector of values. The two EIDs are passed in as `long`s and new SimpleEIDs are instantiated by invoking SimpleEID's one-argument constructor.

The last two constructors are similar to the others except they access `BasicEID` objects and not `long` variables. One receives BasicEID references and the other extracts them from the given BasicAssoc object. Thus, no new SimpleEID objects are instantiated. The references to the two existing objects are stored in the new object.

To get an EID with the method **getOrgTypeEID**, a BasicEID is returned because the application probably wants the reference to the actual OrgType object. If only the numerical EID is required,

the object's **getEID** method may be invoked.* The **setOrgTypeEID** method is overloaded with two different versions. The first accepts a long and instantiates a SimpleEID with it, while the other one takes an existing BasicEID. Their respective arguments are named **orgTypeEID** and **orgTypeEIDRef** to emphasize their different types.

As in other classes, the **toBigString** method returns a string that starts with the object's EID in hexadecimal. If the multiplier is greater than one, it is appended between square brackets. Next is the OrgType EID in hexadecimal, an arrow, the associated EID, and the date interval. The string ends with the association's remarks.

The three subclasses of BasicAssoc—**OrgTMatTAssoc**, **OrgTSkillTAssoc**, and **OrgTPersonTAssoc**—are almost identical. Each has five one-line constructors that invoke BasicAssoc's constructors with the proper arguments. In order to make it clearer what sort of association they contain, one-line access methods were written to invoke the generic get/setAssocEID methods in BasicAssoc. For example, OrgTMatTAssoc has methods **getMatTypeEID** and two versions of **setMatTypeEID**.

3.2.4 Other Classes

The **OrgOrgTAssoc** class is a simplified version of BasicAssoc. It does not have a multiplier instance variable because there is a one to one relationship between each organization and its organization type.

The class requires three constructors (no-argument, multiple arguments, and vector). Logically, it could have been a subclass of BasicAssoc, but we chose to create a separate class because the organization type is the "child" end of the link, not the "parent." In fact, a general "link" superclass could have been defined for all associations and relations. However, the remaining class methods do not have much in common, and most of each class would have ended up being unique. We felt it was sufficient to use BasicEID as the parent class.

BasicRelat is another variation of BasicAssoc, having a BasicEID for the parent, another for the child, a multiplier, and a role. It does not follow the naming convention of converting the database table field names to Java variables because of its generic nature. Its subclasses are **OrgTypeRelat** and **OrgRelat**. OrgRelat has additional constructors because it does not use a multiplier and therefore always passes a multiplier value of 1 to BasicRelat's constructors.

The class **SQLUtility** has one static method and nothing else. This method, **makeAllTables**, contains the SQL code to create all of the tables in a new database. The current definition has details that are specific to MySQL, for example, the keyword used to denote a 64-bit integer. By isolating the code in this class, it should be easy to modify it to support other RDBMSes.

*Experienced Java programmers are familiar with the concept of chaining method calls together. The operation may be performed with *someAssoc*`.getOrgTypeEID().getEID()`.

3.3 Database Methods

Following the tenets of object-oriented programming, we defined the same set of methods for each of our nine classes that has a database table as a counterpart. Whenever possible, the application invokes the desired method by name and polymorphism determines which version of the method should be run. In order to implement this, we had to declare the methods in BasicEID. We made them abstract to further emphasize the fact that each subclass must define its own version of the methods. We also tried to keep SQL code out of the data classes as much as possible.

The method **buildCommaDelimString** returns a comma delimited string containing the values of all the fields that make up one row of a table. Values that are text strings or timestamps are surrounded by single quotes. The class that invokes this method is required to add the SQL-specific code to turn it into a valid SQL *insert* command.* The method is also used to save the data in normal text files for off-line manipulation or archiving.

A MySQL *update* command requires that the field names be supplied along with the values.† We patterned the **buildUpdateString** method after the buildCommaDelimString method. In both cases the table name is supplied by the **OopDatabase** methods that perform the actual database updates.

The final database method is **loadRecords**. Unlike all of the methods mentioned thus far, it is a *static* method. It loops through every record in a ResultSet, extracts the field values, and invokes the class's constructor to instantiate a new instance of the class with the given values. It returns a vector containing all the new objects that have been instantiated. As before, some of the details have been hidden from the class. A method in OopDatabase extracts the desired records from a table (in our initial version, we fetch all of the records in each table) and passes the result set to loadRecords for processing.

3.4 Display and Edit Methods and Classes

The user must be able to view and edit objects in the database, both one at a time and in a scrollable table. We implemented a layered approach to provide the former abilities, and defined a new class **Inspector** to construct and manipulate tables.

A note about color: we use color in our application to highlight and identify different items such as materiel types versus skill types. Because this report is not in color, we disabled the graphical user interface (GUI) parameter colors as defined in table A-1 of appendix A. The same colors would normally be used in the title of an inspector, the button bar of a dialog, and tabs. We left the colors intact for nodes and some buttons.

*The command for MySQL is `INSERT` *tableName* `VALUES` (*value*, *value*, . . . , *value*).

†We did not need the field names in the *insert* command because we were supplying a value for every field.

3.4.1 Panels and Dialogs

The method **makePanel** constructs a Java JPanel with a grid of abbreviated field names and their values. It accepts a boolean (true/false) argument to specify whether or not the object's fields should be editable. By having a method that returns a JPanel, we may embed panels in various GUI components. The simplest of these is a dialog as shown in figure 3 and its editable version in figure 4. The **makeDialog** method in BasicEID creates a JDialog, inserts a panel into it, and creates buttons to close the dialog.

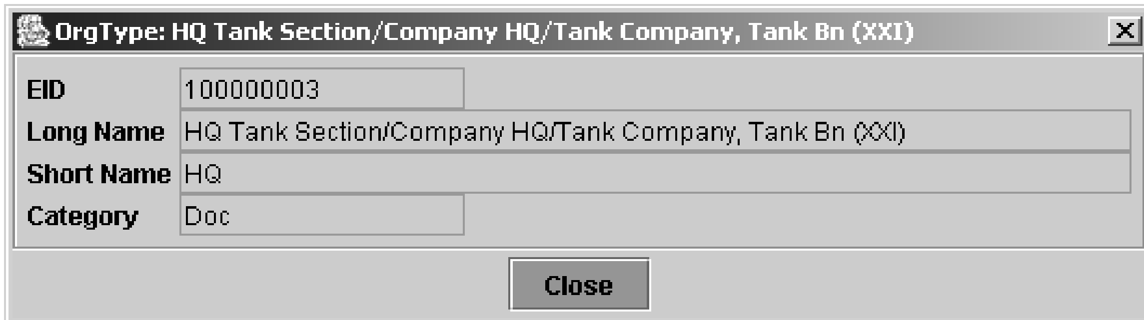


Figure 3. Organization type display dialog.

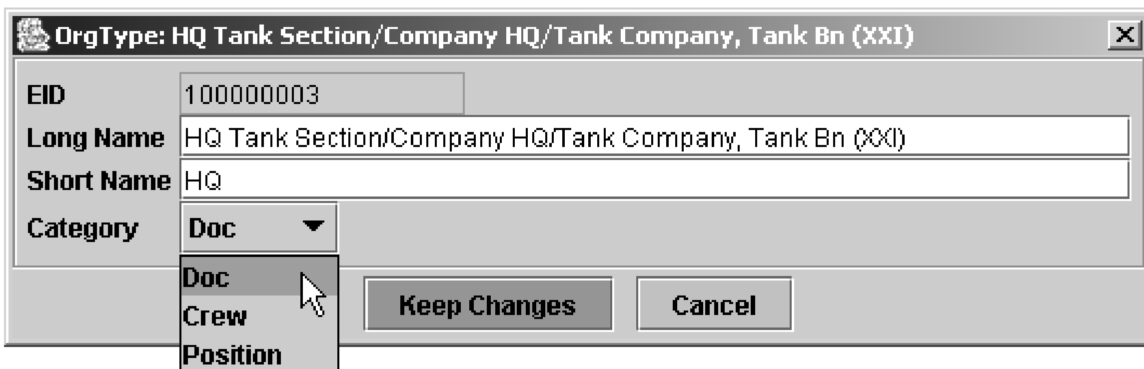


Figure 4. Organization type editable dialog.

The standard Java conventions are used in the panels. The field labels are written in a bold font. A box with a grey outline may not be changed, while a white box contains text that may be edited. The text scrolls within the box and the user is not limited to the area shown on the screen. There are a limited number of categories that may be assigned to an organization type, so a drop-down list (or *combo box* in Java terms) is constructed instead of a simple text field. The editable organization type is shown with the user selecting a value from the drop-down list.

One of our criteria was that the user should never deal with EIDs directly; in fact, he should never see them. This application displays EIDs for debug and explanation purposes. However, for future applications and to aid in clarity, a description is also displayed next to all EIDs (except the

primary EID in each object, which would be redundant). Figures 5 and 6 show how the skill type EIDs are identified in noneditable and editable person type panels.

EID links are always displayed as buttons (which is why the text is emboldened). Clicking on the button will cause a display dialog to be opened for the linked object; in the case of a person type, all of the links are to skill type objects. The descriptive text next to each button is provided by the linked object's **toDescString** method. The editable version of the panel is similar, except the EID and description pairs are shown in drop-down lists.

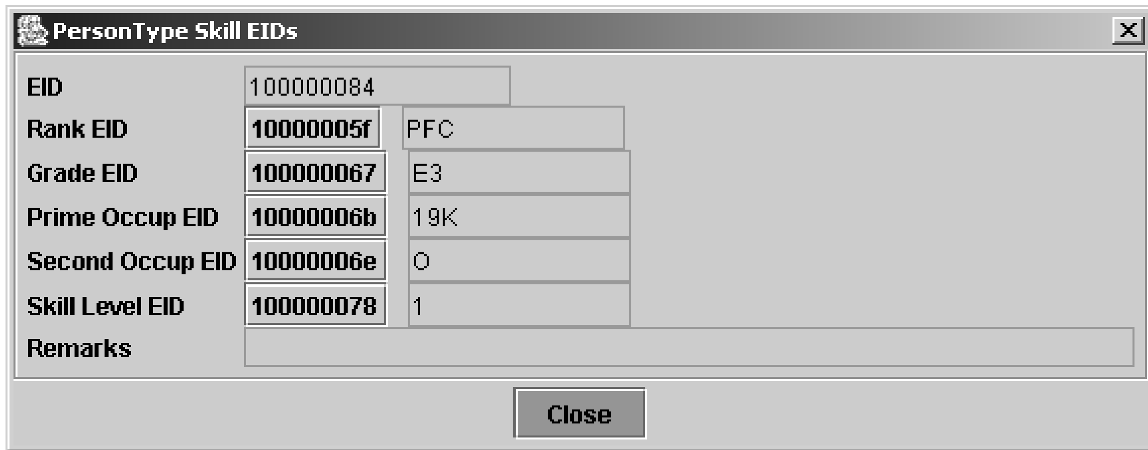


Figure 5. Person type display dialog.

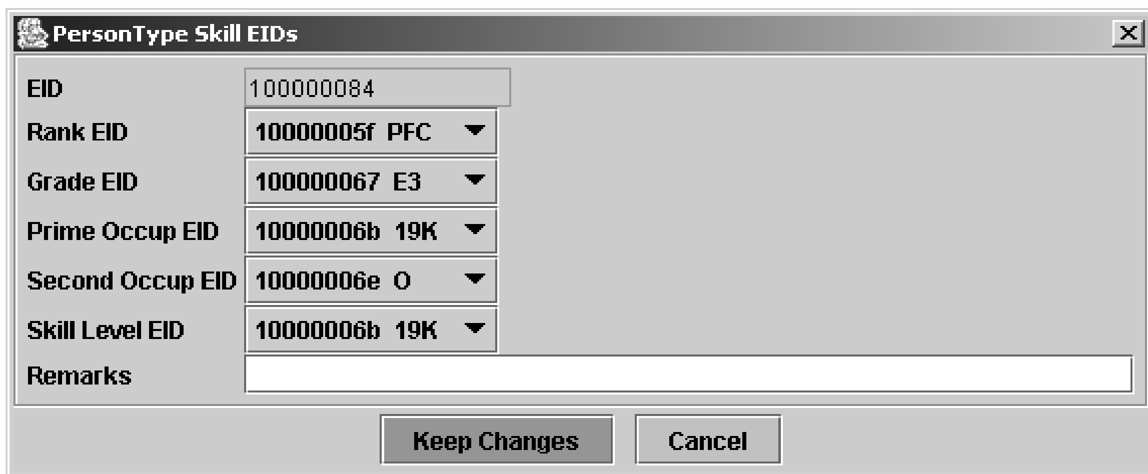


Figure 6. Person type editable dialog.

In keeping with the graphical nature of the relationships between objects, the application provides ways for the user to define links. These are discussed in more detail later. The user also needs to be able to enter descriptive text about the links, so we use editable dialogs such as the one shown in figure 7. We chose to initially allow the user to supply the multiplier by simply typing it in. Those are the only two editable fields.

Figure 7. Organization type/materiel type editable dialog.

The remaining classes—OrgTypeRelat, OrgRelat, and OrgOrgTAssoc—use variations of the OrgTMatTAssoc panel and are not shown in this report. The first two have a field for the relation's role, and only the first has a multiplier.*

When the user accepts the changes he has made to an object in a showDialog editing panel, the object's **fireEditingStopped** method is invoked with the panel as its argument. This method extracts the values from all of the editable fields in the panel. If a value has been changed, the old value in the object is replaced and a flag is set to indicate that the object has been modified. The BasicEID method **wasEdited** returns `true` if the flag is set.

3.4.2 Inspectors

The class **Inspector** was given a generic name to emphasize the general nature of the class. It is a subclass of **AbstractTableModel** and combines a custom table model with a **JTable** and the **Vector** of data rows that are displayed in the table. It is used to display the items associated with a given organization type and to display rollups of the items in an org type subtree.

Inspector's constructor takes an array of column names and uses the private method **addButtonSymbol** to add an unlabeled, fixed width column at the beginning. As shown in figure 8, the contents of this column are "o", "N", or "m" to indicate an unchanged, new, or modified record, respectively. We chose to make the cells of the table noneditable and for all editing to be performed with a panel as described in the previous section. If the user clicks on the first cell of any row, the record is displayed in a noneditable dialog. This first column is the only part of an Inspector that references the BasicEID class.

We incorporated the **TableMap** and **TableSorter** classes from Sun Microsystems (*9*) and modified the related classes **SortHeaderRenderer** and **SortArrowIcon** published in *Java Pro* magazine (*10*). These classes allow the user to sort a table by clicking on any column's header cell, while a second click reverses the direction of the sort. A triangle is drawn to indicate the sort order, with the triangle pointing from high to low values. The materiel type inspector shown has been sorted on the LIN column.

*OrgRelat inherits a multiplier which is ignored.

Materiel Type

	EID	Qty	LIN △	Name
o	100000025	1	B49272	BAYONET-KNIFE: W/SCABBARD F...
N	100000026	1	B67766	BINOCULAR: MODULAR CONSTR...
o	100000021	1	M18526	MASK CHEMICAL BIOLOGICAL: CO...
m	100000024	1	P98152	PISTOL 9MM AUTOMATIC: M9

Figure 8. Materiel type inspector.

The Inspector class has several methods for manipulating the data that is stored in the Inspector. Method **addRow** adds a row of data, **updateRow** finds the row that has the same primary EID as the supplied row and replaces it with the new values, and **deleteAllRows** clears the Inspector.[*] Every class that corresponds to a database table defines a **makeRow** method to construct a row of values to be inserted (or updated) in an Inspector's JTable. The application may get the JTable associated with an Inspector by invoking **getTable**. In addition, the standard methods as required by AbstractTableModel are defined. The inner class **EIDCellEditor** extends **DefaultCellEditor** and provides the mechanism to display a dialog when the user clicks on the first cell of any row.

Each table class also contains two or three versions of a static method named **makeInspector**. The primary version may accept a boolean argument which denotes whether or not the Inspector should have a "Qty" (quantity) column.[†] It instantiates and returns a new Inspector using the column names that are appropriate for this class. Associations and relations have a multiplier field that states how many instances of an object are required (e.g., a company may have three platoons and a crew may need two sets of night vision goggles), and the "Qty" column is not necessary. However, when a basic type is displayed in an Inspector, the quantity must be obtained from the link or computed by rolling up the child objects.

The other makeInspector instantiates an Inspector by invoking the single argument version, then loads the table with a vector of data. The three OrgT*XXX*TAssoc classes use the static **loadInspector** method in BasicAssoc, while the other classes must each duplicate the same enumeration loop to load the data. Other arguments denote the table's width in pixels and the maximum number of rows to display. We assume that the table will be displayed in a scrollable pane.

3.5 OopDatabase Class

The class **OopDatabase** serves two primary roles: it contains all of the methods that connect to a database server and it holds the cache of data that has been loaded from the database into memory. The only methods that involve SQL in the classes we have described are buildCommaDelimString and buildUpdateString, which return strings that conform to the standard SQL *insert* and *update* commands. The static method loadRecords must know the name of every field and processes a ResultSet of records. However, none of those methods communicates directly with a database.

[*] BasicEID invokes the first two methods with its convenience methods **addRowTo** and **updateRowIn**.
[†] If no-argument version exists, it invokes the other method with the value `false`.

3.5.1 JDBC and SQL Methods

The constructor for OopDatabase processes the application's property list with **processProperties** to determine the name of the JDBC driver, the URL and host name for the MySQL server,* and the user name and password needed to connect to the database. If any JDBC property is not given, a JDBC dialog is displayed (with **showJDBCDialog**) as shown in figure 9. If the user name is not given, the current login name is used, while a missing password causes the user to be presented with a login dialog (with **showLoginDialog**). The JDBC driver is loaded and a login timeout is set so the application terminates cleanly if the server is not functioning.

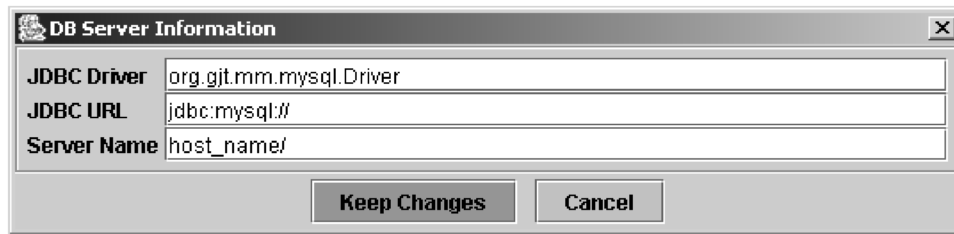


Figure 9. JDBC properties dialog.

The method **connect** is given the name of the desired database and attempts to open a connection to it with the stored user name and password. If the database name is `null`, a simple connection is made to the server, but no database is opened. If the connection attempt is successful, the value `true` is returned and the connection is stored in an instance variable. The **close** method closes the connection, while **reconnect** may be used to open a new connection to the current database.

Invoking **loadTable** with the name of a table causes an SQL query statement† to be executed, returning a ResultSet of all of the records in the table. It is invoked several times by **loadAllTables**, each time passing the result set to the appropriate loadRecords class method. This isolates the instantiation of the new objects from the query that is used to fetch their records from the database. The method also gets the current EID from the Seq_Table table and indirectly builds the internal links as described in the next section.

To store data in the database, method **saveVector** is given a vector of BasicEIDs that are all of the same type (OrgType, OrgTMatTAssoc, etc.). The class name of the first object is determined‡ and used as the table name. Each object in the vector is inserted into the table by invoking the **insertTable** method which constructs an SQL *insert* command with the object's buildCommaDelimString. The object's status flag is then cleared to indicate that the object is identical to the record in the database.

Modified objects are updated in a similar manner. Method **updateVector** is given a vector of BasicEIDs, where each object may be in one of three states. Unchanged objects are ignored, while newly created objects are passed to insertTable. An SQL *update* command is constructed

*Switching to a different RDBMS may be as simple as changing the value of the driver and URL.

†`SELECT * FROM` *tableName*.

‡*someVector*`.get(0).getClass().getName()`, then remove the package prefix.

by giving a modified object to **updateTable** which in turn invokes buildUpdateString. The object's status flag is then cleared.

The methods saveVector and updateVector are invoked from the **saveAll** and **saveChanges** methods, respectively, for each of the database class vectors. The current EID is also stored in the Seq_Table table, and a cache status flag is cleared because the cache and database are now identical.

The related methods **saveAllToFiles** and **writeVectorToFile** should be self-explanatory. The former calls the latter to store data in text files so they may be archived or *carefully* edited and reloaded with **loadFiles**. (The private method **parseLine** extracts the elements from a **StreamTokenizer** and returns a vector of String and Long objects.) This technique could also be used to copy the data to another system. The data strings are created with buildCommaDelimString and written to files with the same names as the classes (or tables).

The general database methods are **makeNewDB**, which creates a new database and indirectly creates its tables, and **emptyDBTables**, which deletes all data from the tables of an existing database. The application allows the user to load data from an existing database. The list of database names ending with "chart" is generated with **getDatabaseNames**. The method **getErrorCode** returns the integer error code of the last SQL statement that was executed and is primarily used to detect when a new database could not be created because it already exists.

3.5.2 Hash Table Methods

One component of the cache is a hash table that maps BasicEIDs to database table objects. Given an EID, we need to be able to quickly and easily find the object with that value. Mappings are created with **addToHash**, which takes any child of BasicEID and inserts it into the hash table. EID values are supposed to be unique, but to be safe we check for duplicates here and throw an **EIDException** if we find one. Method **updateHash** takes a vector of objects and adds them all with addToHash. We also need to perform the reverse operation, i.e., find an object in the hash table given its EID. The overloaded method **getBasicEIDFromHash** does this with either a long EID or a BasicEID value.

The private method **makeLinks** performs a very important operation. It replaces the simple BasicEIDs that are links with references to the objects that have those EIDs as mentioned in section 3.1. It steps through every vector in the cache, gets the EID from a link field, uses it to get the appropriate object from the hash table, and stores the object reference in the link field.* This method is invoked after data has been loaded from a database or set of data files. It also builds a vector of all of the BasicEID vectors for easy manipulation.

*Given the OrgTMatTAssoc object `otmt`, the organization type link is updated with `otmt.setOrgTypeEID(getBasicEIDFromHash(otmt.getOrgTypeEID()))`.

3.5.3 Tree Methods

We assume that a database contains exactly one organization type tree and one organization tree for a given date of interest.* The root (top-most node) of any tree by definition must appear exactly once. The method **getRootNode** searches the list of OrgTypeRelat links and finds the OrgType which is a parent but never a child. The method **buildOrgTypeTree** starts building the org type tree with the root object, then recursively invokes the private method **addOrgTypeSubTree** to construct subtrees from the OrgTypeRelat data and add them to the tree. Since we also want to know how many times an org type appears in the tree when the user edits it, the method **countOrgTypeNodes** traverses the org type tree and invokes the **addToNodeCount** method of each OrgNode object. (As a side effect, countOrgTypeNodes returns the total number of nodes in the tree.)

The organization tree is constructed in a similar manner. The root organization is located with getRootNode, then recursion is used by **addOrgSubTree** to construct the tree from the OrgRelat links. Every organization is unique, so there is no reason to compute node counts.

3.5.4 Cache Access Methods

A large number of methods provide access to the values stored in the cache. Simple methods like **getOrgTypes** and **getOrgTSkillTAssocs** return the vector of desired objects, while similar methods (such as **addOrgType** and **addOrgTSkillTAssoc**) add a new object to a vector and also insert it in the hash table.

Objects containing links require more complicated methods. For example, the method **getOrgTMatTFromOrgT** searches the vector of OrgTMatTAssoc objects for objects that have the desired OrgType object as one end of a link (by invoking getOrgTypeEID) and returns a vector of all of the links found.† The English equivalent is, "Give me all of the alignments of materiel type objects with this organization type object." Association creation methods like **makeOrgTypeAssocs** use the getOrgT*XXX*TFromOrgT methods to get vectors of associations, then instantiate new OrgT*XXX*TAssoc objects and perform the necessary bookkeeping to properly update the cache. Relations are manipulated in a similar fashion. Method **getOrgTypeLink** finds the OrgType object with the desired parent and child EIDs, and **makeOrgTypeLink** performs the reverse operation by creating a new OrgTypeRelat object with the given parent, child, and (optional) role.

3.5.5 Other Methods

In order to properly instantiate new objects for eventual insertion into the database, the application must obtain new EIDs. The method **getNextEID** increments the EID counter in the cache, then returns the new value. A nonincrementing version named **getLastEID** simply returns

*The ability to store incomplete subtrees may be added in the future.

†Because the reference to the actual materiel type object has been stored in the OrgTMatTAssoc object, the hash table is not required.

the current value, although the application does not invoke it at this time because only the OopDatabase class needs the value. Eventually an EID server will be required to ensure that the EIDs are unique throughout the enterprise.

The debug method **printVectors** prints the first 10 elements of every vector to verify that the data is being loaded properly. It also prints the number of objects of each type and the current EID value.

3.6 Tree Component Classes

3.6.1 Background

The application allows the user to build or modify MTOEs, or Modification Tables of Organization and Equipment, in the form of a tree. We chose to use the standard Java class **DefaultMutableTreeNode** which defines general-purpose nodes in a tree data structure. However, we could not use the **JTree** class because it displays hierarchical nodes in an outline form and we wanted to use a horizontal display layout.*

To display the organization data in a conventional, top to bottom, tree representation, we adapted an algorithm from the OrgChart program (*11*). The algorithm was converted from the original C code into Java methods that accept a tree constructed from DefaultMutableTreeNodes as input.

3.6.2 Node Classes

Before we may build a tree, we need to define our own node classes. The class **OrgNode** is a small "wrapper" class that contains a BasicOrg and a BasicRelat as instance variables. The BasicOrg is a reference to an OrgType or Org, while the BasicRelat, an OrgTypeRelat or OrgRelat, is the link to the parent node. By storing the link in this node, we may obtain the value of the multiplier and also determine which links are new.

In addition to a constructor that accepts a BasicOrg and a BasicRelat, along with access methods to get and set those variables, there is a constructor which instantiates a new OrgNode from an existing one. A set of generic convenience methods allow the programmer to access the BasicOrg's components (EID, long name, short name, category, and node count) directly by relaying the request to the appropriate BasicOrg method.†

Two OrgNodes may be compared with the **equals** method which returns `true` if the nodes contain the same BasicOrg. Method **findInName** invokes the BasicOrg's method of the same name to determine if the given string is part of the object's long or short name.

Class **DisplayNode** is a child of OrgNode and adds the variables and methods needed to draw a node. It has many instance variables to keep track of the geometric aspects of the node, such as the coordinates of the upper left and lower right corners of the node, the symbol's center, and the

*We did use JTree during the early development phase of the project.

†The programmer may write *node*`.getLongName()` instead of *node*`.getBasicOrg().getLongName()`.

attachment points of the parent and child nodes. The method **processProperties** invokes the **getColorFromProperty** method of class **MyProperties** to get the colors to be used when drawing the nodes.

The variable **nodeIsVisible** indicates whether or not the node should be drawn, **nodeIsExpanded** does the same for its children, and **nodeIsSelected** is `true` if this node has been selected by the user. Most of the variables are private, but the ones that are heavily used by the tree-drawing class are public. There are also public constants that define sizes and other parameters. The method **drawNode** draws and labels the node. It uses the inner class **ParseSymbol** which will eventually generate the labels from a MIL-STD-2525B (*12*) descriptive string.

3.6.3 TreePanel Class

The class **TreePanel** is a subclass of **JPanel**. It draws an organization or organization type tree, displays popup menus when the user clicks on a node, processes the menu choice, and provides other tree-related methods. The constructor accepts a reference to the parent application class and the root of a DefaultMutableTreeNode tree. It determines if the tree contains organization or org type nodes because different operations may be performed on the trees. It creates menus for both the left and right buttons, assigns context help or "tool tips" to them, registers listeners for mouse events, instantiates a **NodePanel**, and indirectly builds the tree of DisplayNodes. Other operations are application-specific and are explained later.

Method **makeTree** starts by getting the reference to the cache (OopDatabase object) from the main application object. We did not do this in the constructor because the cache and tree used by a TreePanel may change while the application is running. The tree is built by the private method **buildDisplayTree**, which copies the tree (with **copyTree**) into a new structure and replaces all of the OrgNodes with DisplayNodes. The **cloneTree** method is similar to copyTree, except it instantiates new DisplayNode objects instead of copying references to existing objects.

The standard Swing method **paintComponent** is overridden to draw the tree. It traverses the tree, draws all nodes that are visible, and records the bounding box (area enclosing the tree) of the visible tree. It then traverses a second time to draw the horizontal connecting lines. If the bounding box has changed size since the last time the tree was drawn, the panel is resized and the tree is redrawn within it. One version of the overloaded method **findInTree** searches the entire tree for a node with the desired EID. The other version builds a vector of nodes whose long or short name contains a given substring. When the user clicks the mouse in a TreePanel, **mouseClicked** marks the node that the mouse cursor is in and opens a popup menu.

There are several private utility methods. The methods **positionTreeNodes** and **position** compute the bounding box of a subtree and the attachment points to the parent and child nodes, respectively. Another method is **hideTree**, which recursively collapses a subtree so that all of its nodes are invisible. The method **findNode** determines if the mouse was clicked within a node, and if so it returns the DefaultMutableTreeNode.

3.6.4 NodePanel Class

Class **NodePanel** is similar to TreePanel, but it displays a vector of nodes in a row rather than in a tree. It is used as a working area for new nodes that are to be added to a tree. Each TreePanel instantiates its own NodePanel object. The constructor initializes the instance variables and registers a mouse listener. Method **addNode** adds a node to the vector and **removeNode** removes one. Each resizes the panel to match the necessary display area. The **getNode** method returns the ith node in the vector, while **getSelectedNode** returns the currently selected node.

Like TreePanel, **paintComponent** draws all of the nodes. No connecting lines are drawn because the nodes have no relation to each other. The **mouseClicked** method uses a version of **findNode** to see if the user clicked on a node. If so, the node is marked as selected.

3.6.5 Displaying a Tree

The tree-drawing algorithm uses a simplistic approach in that no effort is made to optimally pack the positions of children under their parent. Instead, a simple nonoverlapping technique is used. A representative tree diagram is shown in figure 10. In this example the root node is A, B–D are internal nodes, and E–G are leaf nodes. Since tree data structures are generally displayed with the root, or starting point, at the top of the page and the widest part of the structure somewhere lower, the display algorithm is necessarily built on a depth first, recursive algorithm. This algorithm calculates the position of each node, starting at the bottom left and finishing at the root node. Once these positions are known, the rendering algorithm calls the **drawNode** method of each DisplayNode to draw the nodes and the necessary lines to interconnect all of the displayed nodes. The following sections describe node positioning, node drawing, and node connecting in detail.

3.6.6 Tree Positioning Algorithm

In this implementation of the rendering algorithm, the calculation of node positions is divided between two methods, both of which are found in the TreePanel class. The process begins with the method **positionTreeNodes** being invoked with the the root node of the tree to be drawn and initial values of 0 for **maxX** and -3 for **y**. The -3 value is required to make spacing work correctly as the recursion occurs. In positionTreeNodes the vertical spacing is set and for each child positionTreeNodes is recursively called. It should be noted that the coordinate system we are using has (0,0) in the upper left corner with **x** increasing to the right and **y** increasing as we go down on the screen. The following is a pseudocode description of positionTreeNodes:

```
1: initial arguments are maxX = 0, y = -3, treeNode = root
2: y += 2 * YSPACE
3: firstX = maxX
4: for each of the child nodes:
5:    maxX = positionTreeNodes(maxX, y, childNode)
6: maxX = position(treeNode, maxX, firstX, y)
7: return maxX
```

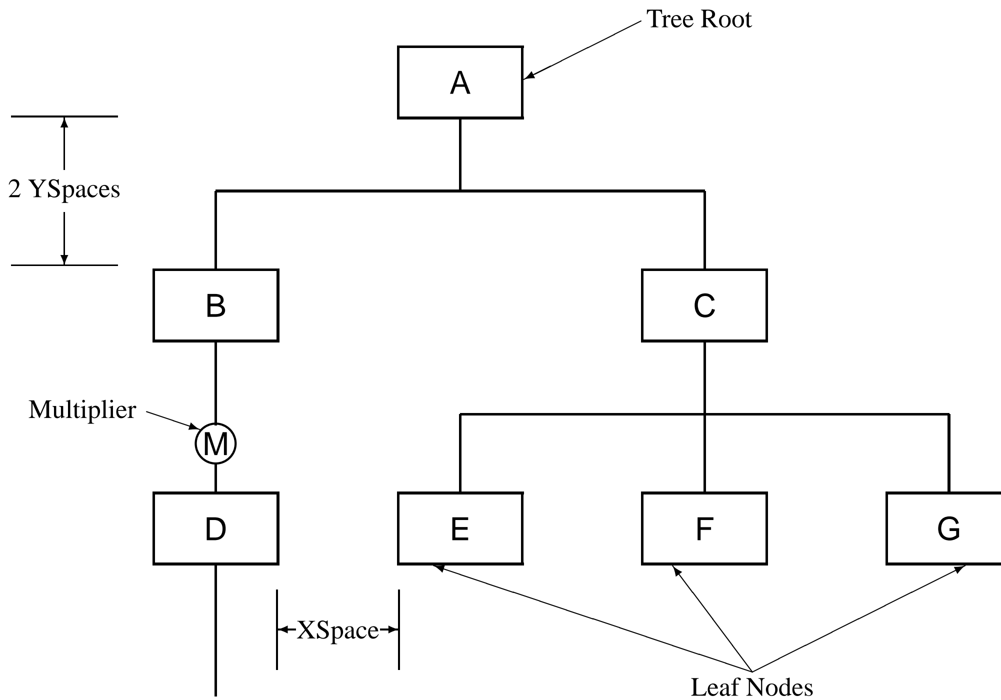


Figure 10. Representative tree diagram.

The method **position** computes the **x** positions for each tree node. The pseudocode for this process is shown. Method positionTreeNodes recurses until a leaf node is reached. Once a leaf is reached, steps 26–29 are executed, with the variable maxX being used to keep track of the **x** spacing of the nodes and the **y** position having been set in positionTreeNodes.

After all the children of a particular node have been positioned, we back up the recursion tree and position the parent node over the average of the leftmost and rightmost child nodes. This is done in steps 5–24.

```
 1: arguments are treeNode, maxX, firstX, y
 2: subUnitsFlag = true
 3: rightX = leftX = -XSPACE
 4: if treeNode has children
 5:    leftflag = true
 6:    for each child node:
 7:       subUnitsFlag = false
 8:       if leftSubFlag
 9:          if parentConnect.x == -1
10:             leftX = firstX
11:          else
12:             leftX = parentConnect.x
13:          leftSubFlag = false
14:       if parentConnect.x == -1
```

```
15:          rightX = leftX
16:       else
17:          rightX = parentConnect.x
18:    if subUnitsFlag
19:       maxX = 3 * XSPACE
20:       rightX = firstX + 3 * XSPACE
21:       leftX = rightX
22:    treeNode.setParentConnect(leftX+rightX)/2, y)
23:    treeNode.rightChildX = rightX
24:    treeNode.leftChildX  = leftX
25: else (no children)
26:    treeNode.setParentConnect(firstX + 3 * XSPACE, y)
27:    treeNode.rightChildX = 0
28:    treeNode.leftChildX  = 0
29:    maxX += 3 * XSPACE
30: return maxX
```

Figure 11 shows the order in which the node positions are calculated. Node D is positioned first; and since it is the only child of node B and its children are not being displayed, node B is drawn next. Nodes E, F, and G are next to have their positions calculated, then node C is positioned. Finally, node A can be centered over B and C.

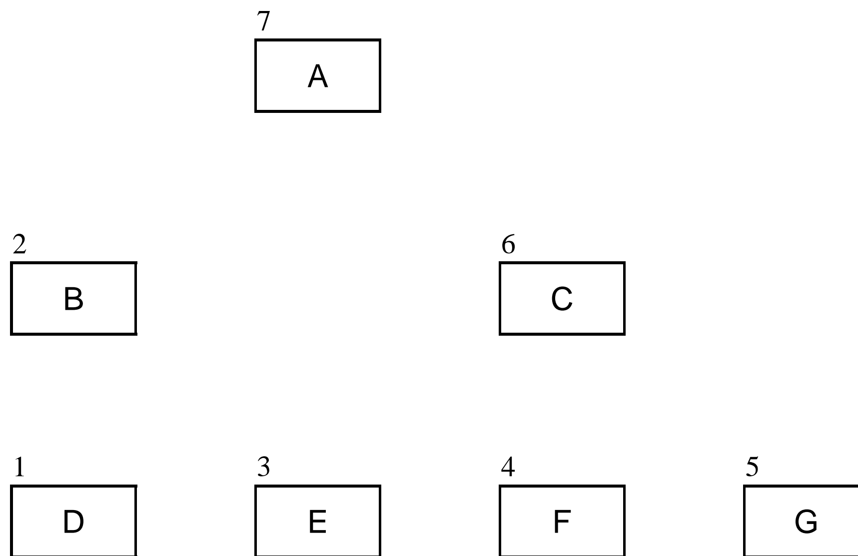


Figure 11. Order of position calculations.

3.6.7 Node Drawing Details

Figure 12 shows a typical node as used in this application. What is drawn for the node is entirely a function of the drawNode method of the DisplayNode class. Each node consists of three parts: an

echelon symbol area, a unit type area, and a unit name area. The dotted lines do not appear on the display, but represent areas reserved for the appropriate information. Of these display areas, name is filled out appropriately. The unit type field is also filled in for an organization type tree, but not an organization tree. It is expected that eventually all the information needed to properly fill in a node image could be obtained from parsing MIL-STD-2525B data strings, assuming that such data is stored in the database. In the future, it is also hoped that unit names could be derived from the path used to access the node. There are currently three types of OrgType nodes. These are doctrinal, crew, and position. Position type nodes are associated with individuals, whereas a crew node would consist of one or more position nodes and be associated with a particular piece of hardware. Doctrinal nodes are those nodes internal to the tree and are composed of other nodes.

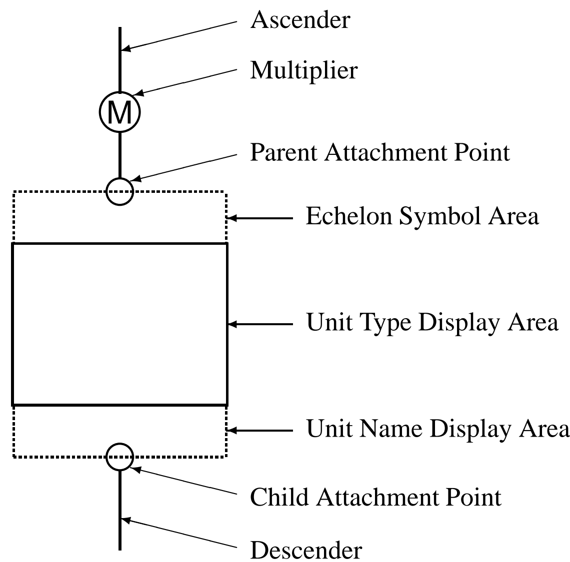


Figure 12. Typical node with connecting lines.

3.6.8 Tree Rendering Details

Were it not for the fact that we allow multipliers for certain nodes in the organization type trees, the ascenders and descenders for each node could be drawn by the node drawing routines. However, we do allow multipliers; and since multipliers are a function of the tree structure, not a particular node, the ascenders and descenders are drawn by the rendering algorithm. As we traverse the tree data structure and call the drawNode method to draw each node in the appropriate place, we also draw an ascender for each node that has a parent node and a descender for each node that has children. If the link is new, the ascender is drawn in the color specified by the **node.new.color** property (the default is red). Even if the children are not displayed, the descender serves as a reminder that there are more nodes down this branch. Figure 12 shows the ascenders and descenders and includes a multiplier construct. If the multiplier is one, then the construct is not shown and a straight line is drawn in place of the circle and number. Figure 13 shows the example tree after this phase of the rendering algorithm has completed.

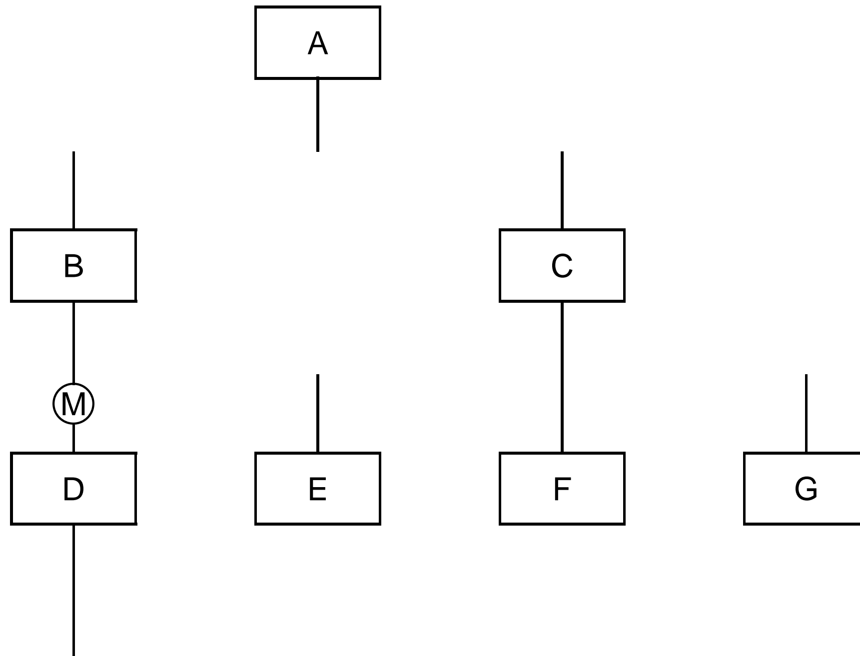


Figure 13. Tree with first phase of connecting lines.

Finally, the horizontal lines that connect the displayed child nodes of a single parent are drawn. This is shown in figure 14, which represents the tree as it would be displayed, except in a real case the multiplier would be a specific number and not M.

3.7 Application Classes

3.7.1 Overview

Many of the classes used by the application have already been presented. The complete hierarchy of application classes is shown in figure 15. Standard Java classes are shown in sans serif and the standard abstract class is in **bold sans serif**. Intermediate classes, such as between Object and JPanel, are not named and are indicated with a dotted line. The remaining application classes are discussed in this section.

3.7.2 IChart Class

The **IChart** class is the primary control for the application. It provides the GUI consisting of a menu bar, tree panels, node panels, and inspectors to display detail and rollup information. This section discusses the general structure of the class, while the operational details are in the users' guide section.

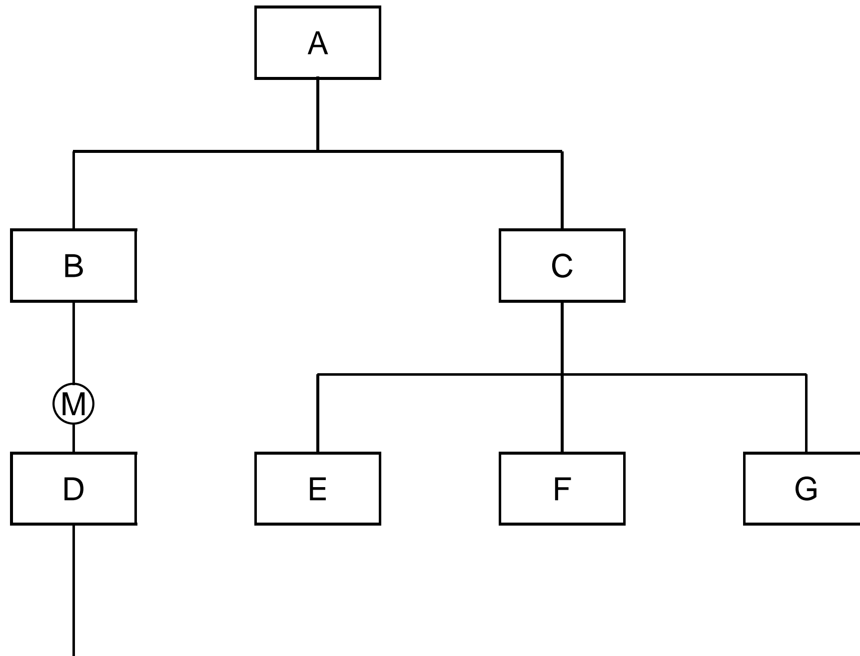


Figure 14. Tree with final connecting lines.

The constructor begins by reading a property file and processing the properties (with **processProperties**) that are required by this class. The application's argument list is scanned, and values that are found override the defaults from the property list.* The method **loadDatabase** is given the name of the database. It in turn instantiates an OopDatabase object, passing its constructor the property list so that it may extract server information such as the name of the MySQL server. OopDatabase methods are invoked to open a connection, load all of the data into the cache, and close the connection. For test purposes, the debug method printVectors is invoked to verify that the data has been successfully read into the cache.

After the **main** method has instantiated an IChart object, it invokes IChart's **createGUI** method. This method creates the menu bar with all of its items and the panel to display detail information in inspectors (an object of type **DetailPanel**). The two trees are built with OopDatabase's buildOrgTypeTree and buildOrgTree methods, then they are passed to TreePanel's constructor when the two tree panels are instantiated. The node panels are instantiated by TreePanel and retrieved by createGUI with TreePanel's **getSandbox**† method. The last component is a status area to display messages to the user.

Now that the user interface has been built, the program becomes asynchronous. It enters an implicit loop and waits for the user to access a menu item or interact with the trees. The **actionPerformed** method determines which menu item was chosen and either processes the request directly or invokes another method to do the work.

---

*Currently only the name of the database and the date of interest may be supplied in this way.
†We use the term "sandbox" because it is a holding area for new objects.

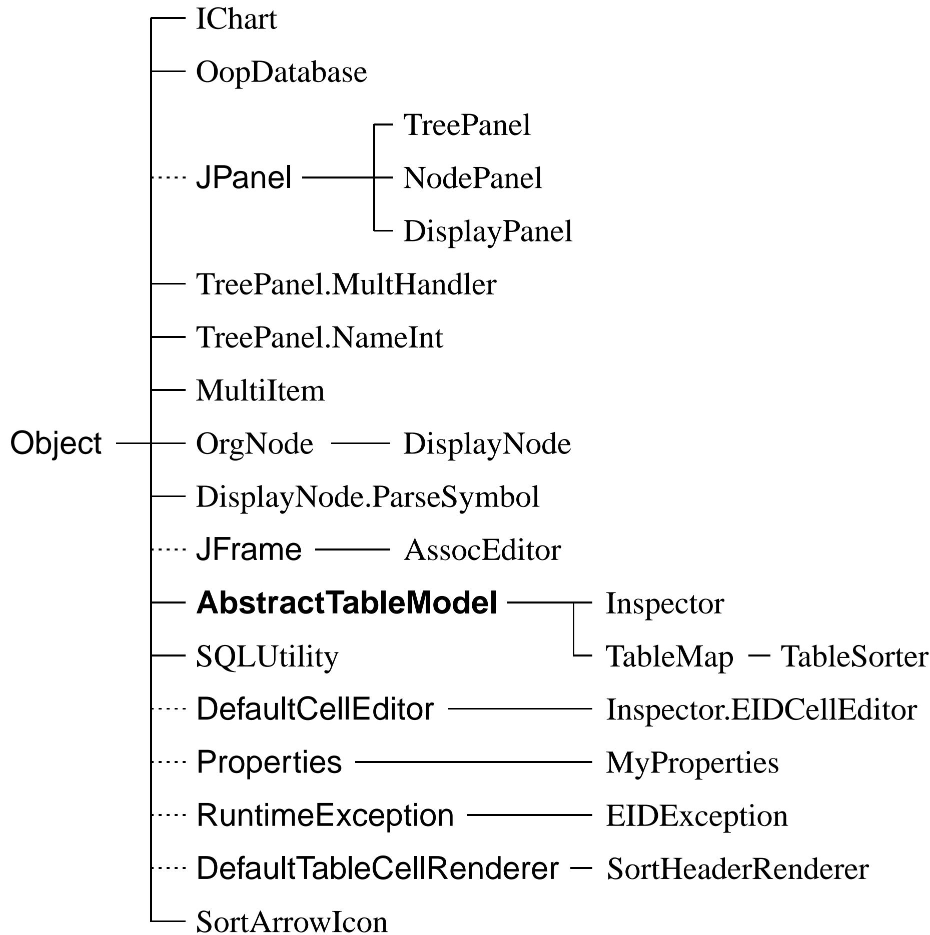


Figure 15. Application class hierarchy.

Access methods are provided to allow other objects to get the private instance variables that must be shared. The objects returned by the methods should be apparent from their names: **getOrgTree**, **getDetailPanel**, **getOopDatabase**, and **getProperties**. Other simple methods are **showStatus**, which displays a line of text in the status area, and **makeTitle**, which changes the window's title.

3.7.3 DetailPanel and MultiItem Classes

DetailPanel is another subclass of **JPanel**. It contains scrollable inspectors for BasicOrg (OrgType or Org), MatType, SkillType, and PersonType objects.* The inspectors are stored in nested **JSplitPane**s to allow the user to independently resize them as needed.

The constructor is given the reference to the OopDatabase object, the width of the entire panel in pixels, and the number of rows to display in each inspector. If the number of rows of the BasicOrg inspector is zero, the inspector is not instantiated. A color-coded label is placed above each

*The materiel type inspector shown in figure 8 is taken from a DetailPanel.

inspector to serve as its title. A component listener is registered to detect when the DetailPanel is resized so that the inspectors may be resized, otherwise they would be centered horizontally with blank space on the sides.

Access methods exist to get each inspector. Their names are of the form **get*XXX*Inspector** where *XXX* is **Org**, **Mat**, **Skill**, or **Person**. The **setOopDB** method reinitializes everything by storing a new OopDatabase reference and emptying all of the inspector tables.

The remaining methods populate the inspector tables. Before explaining them, we must discuss the **MultiItem** "wrapper" class. This class combines a BasicEID with an integer quantity. It is very similar to the OrgNode class, but was kept separate for clarity. The no-argument constructor is never used and was included for completeness. The other constructor accepts a reference to a BasicEID object and the initial quantity. The access methods are self-explanatory: **getItem**, **setItem**, **getQuantity**, **setQuantity**, and **addToQuantity**. The **toString** method returns a string consisting of the quantity in square brackets followed by the EID in hexadecimal. The method **equals** returns `true` if the BasicEID instance variable is equal to the given BasicEID reference.

The DetailPanel method **showDetails** extracts the organization type or organization from an OrgNode. If the node contains an organization, its associated org type is obtained with OopDatabase's getOrgTFromOrg method. The objects aligned with the org type are collected with the various get*XXX*TFromOrgT methods. The results are displayed in the appropriate inspectors. The **showRollup** method is very similar, except it invokes the recursive private method **doOrgRollup** to combine the details for an organization or org type subtree. The latter method does essentially the same thing as showDetails. Instead of immediately adding the materiel type, skill type, or person type and its multiplier to the appropriate inspector, it instantiates a MultiItem object or adds the quantity to an existing object which has been found with the private **getMultiItem** method. After the entire subtree has been processed, the MultiItems are displayed in the inspectors.

3.7.4 AssocEditor Class

The **AssocEditor** class provides the user with a way to create and edit materiel, skill, and person types and to align them with a given organization type. It extends **JFrame** so that it may be displayed as an independent window. It consists of several components:

1. the org type displayed in a non-editable panel,
2. a DetailPanel containing inspectors for the aligned types,
3. an inspector for each type showing all items in the cache, and
4. two sets of buttons to perform the desired operations.

The materiel, skill, and person type inspectors in item 3 contain many records, so they are tabs in a **JTabbedPane**. The user may display a table by clicking on the desired tab.

The constructor creates the GUI components and loads all of the data. It receives references to the application's OopDatabase object and the OrgType object that the user is interested in. The organization type creates its panel by invoking makePanel with the edit flag set to **`false`**. After the DetailPanel has been instantiated, data is put into the inspectors by simply invoking its showDetails method with the organization type. The materiel type inspector is made and filled with data by passing the vector of all materiel types to MatType.makeInspector, converted into a scrollable pane, and added to the JTabbedPane as a new tab. The process is repeated for the SkillType and PersonType inspectors. The buttons are instantiated in two simple **Box**es with a horizontal layout.

The standard method **actionPerformed** receives all button clicks. It processes the commands from the right hand box directly and invokes the private method **editLink** for the left hand box. The private method **scrollToLastRow** redisplays an inspector so that the last row of data is visible. This is necessary because new records are always appended to the end of the table, and we want to show the user the new materiel (or skill or person) type object that he just created.

---

4. Users' Guide

4.1 Introduction

Before we explain how to use the IChart application, we must present our implementation philosophy, assumptions, and limitations. Figure 1 shows the relationships between the tables. The emphasis is on *types* or templates, which may be thought of as typical versions of an organization, materiel, or other object. In order to build an organization chart, the user first creates one or more organization types and the materiel, skill, and person types to be aligned with those org types. The main IChart window, shown in figure 16, is a GUI that is used to assign parent/child relationships to the org type nodes and to construct the force structure.

The main window is divided into several major areas. The frame's title contains the name of the current database and the date of interest. The menu bar is located directly under the title bar and is described in section 4.3. A status line to provide feedback to the user appears along the bottom of the window. The central area is divided into two sections. On the left is the organization type or organization tree panel (section 4.4), with the working area below it. The right portion has several inspectors to display the details about a unit or the rollup of the units in a subtree.

The organization tree, which consists of notional units, is built using the organization type tree as a template. Our application does not contain actual materiel or person items, although they would be needed in an application that tracks information about fielded units. The database tables (and corresponding Java classes) would have more fields added to augment the minimal sample data that we defined.

The association editor is work in progress and is explained in detail in section 4.5. The association editor, shown in figure 17, is the component that enables the user to create, edit, and manipulate the materiel, skill, and person types and align them with the org types. It is a separate

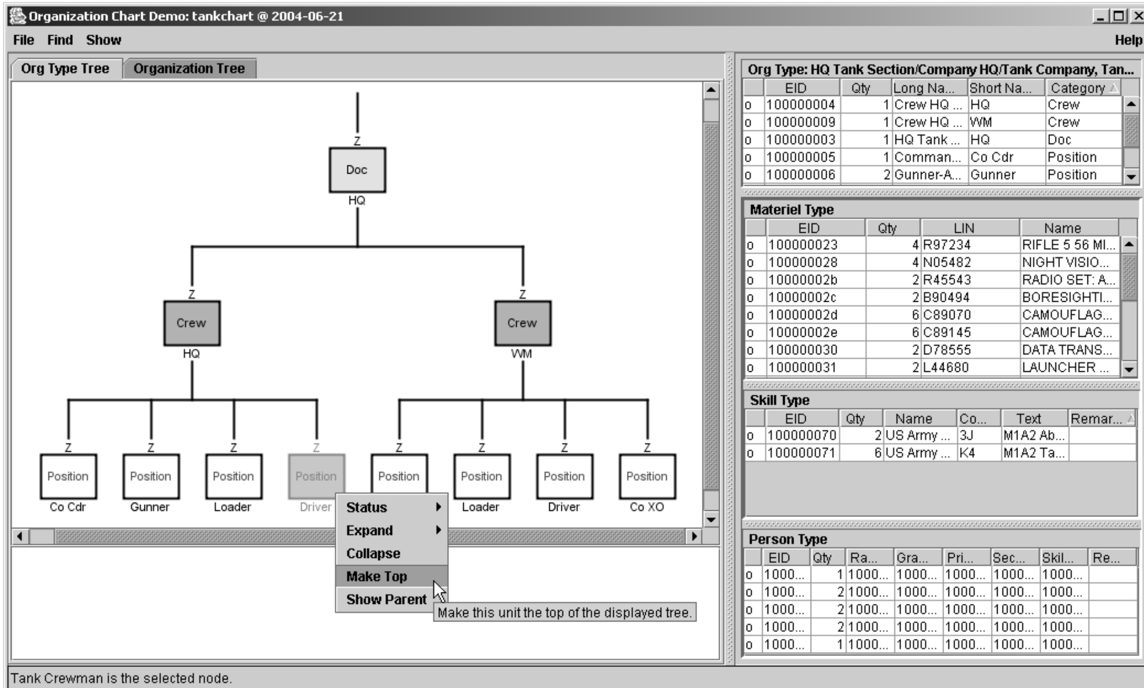


Figure 16. IChart application main window.

window that currently allows the user to create new materiel, skill, and person types, align them with a specific organization type, and edit existing alignments to change the multiplier and remarks. The left side contains an org type in a noneditable panel with its details in inspectors below it. A row of edit buttons is placed at the bottom. The right side has a tabbed panel with an inspector in a separate tab for each of the three alignable types. All instances of a type are displayed in an inspector, which could be unwieldy for a large database. Below the tabbed panel is a set of buttons which are similar in use to the edit popup menu in the tree panel.

When the application is used to view or modify the data in a database, the first thing that occurs is all of the data is read into a cache. This copy of the data is used until the user chooses to store the changes into the database. We made the assumption that the database is not huge, and it is practical to load the entire contents into memory.* Caching the data minimizes database accesses and avoids network lag times. By having the user edit a copy of the data, changes may be easily stored in the database or discarded.

Date/time groups have been partially implemented. The values are ignored when the data is loaded into the cache, to prevent records from being lost when the cache is saved in a new database or in text files. The tree construction algorithm examines all of the objects and uses only those whose date/time group interval includes the date of interest. The user may not edit date/time groups because of the complexity of side effects. Such a task is outside the scope of this project, and we manually created objects with different date/time groups to test their use.

*A future version could implement a paging algorithm or constrain the user by using a subset of the data.

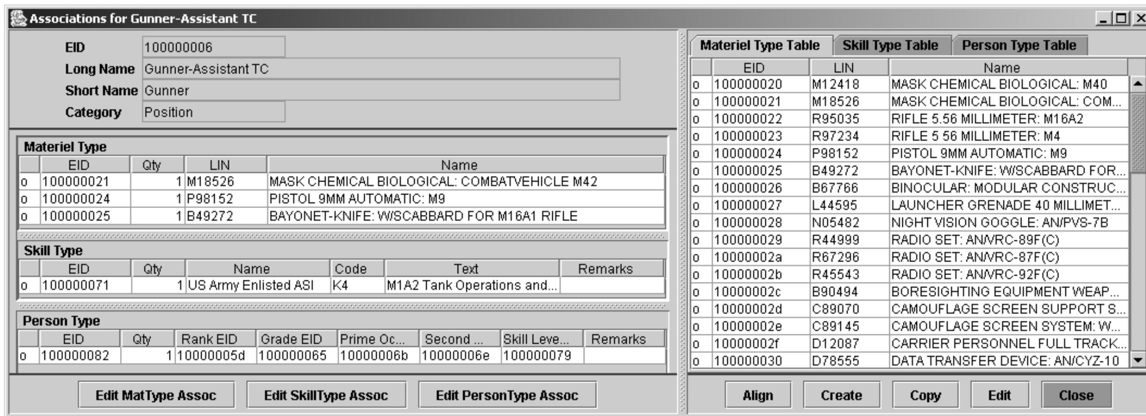


Figure 17. Association editor window.

The user interface is adequate, but not optimal. This is a workbench tool and not a prototype of a production system. We assume that all data is valid and do not impose any military constraints on the trees that the user creates. Minimum error recovery procedures have been implemented. In the unlikely event that an SQL error occurs, what should the program do? We catch the exception that is thrown, print a stack trace, and continue with the program. Every time this occurred during testing, there was a logic error in the program or we had changed the database schema.

4.2 Getting Started

4.2.1 Property Files

We employ property files to facilitate customization. When IChart is run, it begins by attempting to load the file `default.properties`. If the application is run from a Java ARchive (JAR) file, the property file may be stored in the JAR file. If it is not run from a JAR, or the JAR does not contain the default file, then the program attempts to load a local default.properties file. In either case, it then loads the local file named `ichart.properties`.

The default file should define parameters that are the same throughout an organization. The important properties are the JDBC **driver** class and the **URL** to the SQL server.* The property file in table A-1 in appendix A also contains default colors. The ichart property file may contain user-specific settings, such as the name of the host running the SQL **server**, the **database** to use, and the SQL **user** name. The user's SQL **password** may also be stored here, but for security reasons we recommend that this not be done. Table A-2 in appendix A lists the remaining property names.

Any properties that are defined in the ichart file override the corresponding values in the default file. These in turn may be overridden by arguments on the program command line. Any property may appear in either property file or not at all with a couple of exceptions—the driver and URL properties must be defined, and the database name must either be defined in a property file or

*These would probably be hardwired in a production application.

supplied on the command line. The user will be prompted for the remaining JDBC property values except for **datetime**, which defaults to the current date.

4.2.2 Initial Database

At this time the user must manually create a new database. Our initial database was created from an existing Microsoft Access database, and our emphasis was on writing a tool that would allow us to modify and extend the data. We are exploring ways to create a minimal database, either by starting the application in a special mode or by running a second program. The minimum consists of an organization root, org type root, organization/org type association linking the roots, and the highest EID used.

4.3 Menu Bar

4.3.1 Overview

The IChart application contains a menu bar along the top of the main window. The menu bar has four drop-down menus named File, Find, Show, and Help. Every menu item displays help in the form of a Java "tool tip" when the user pauses with the mouse cursor over the menu item. The tip text for the Quit item is shown in figure 18.

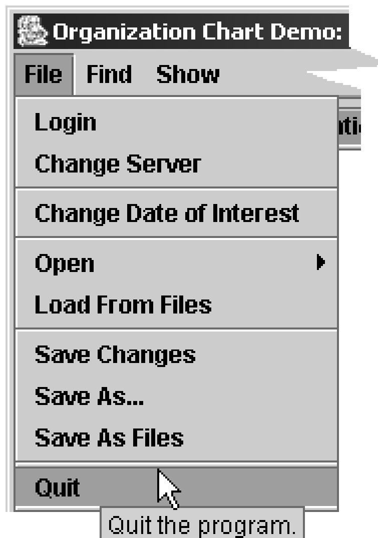


Figure 18. Sample tool tip.

4.3.2 File Menu

The File menu allows the user to change program parameters, load and save databases, and exit from the program. The Login command displays a dialog to let the user change the user SQL

name and password. This dialog is displayed at startup if the user name and/or password are not set in a property file. The JDBC driver and URL and the SQL server name may be supplied with the dialog displayed by the Change Server command, as shown in figure 9. This happens automatically if any of the three values are not set in a property file.

The date of interest is used to select the active records in the database. It may be changed with the Change Date of Interest dialog. After the date is changed, the trees are rebuilt and the scratch pad "sandboxes" are emptied.

The Open command displays a list of databases on the server. To avoid confusion with other databases that may be stored there, only the ones whose names end with **`chart`** are shown. The current cache is emptied and the chosen database is loaded in its place. A similar operation is performed by Load From Files, except the data is loaded from text files. A standard Java file chooser is displayed with the names of all the subdirectories in the current directory.* The user may move around in the file system and select the desired directory name. Because IChart is database-oriented, the user is prompted to save the newly loaded data to a new database. Both Open and Load remove any objects that are in the sandboxes to prevent them from being used in the second database.

Like many other applications, the data changes made by the user are lost when the program exits, unless the user chooses to save them. IChart keeps track of which records have been modified and which are new. The Save Changes command performs SQL *update* and *insert* operations on the current database, ignoring all records that are unchanged. In contrast to this, Save As. . . prompts the user for the name of a new database, then saves the entire cache into it. The suffix **`chart`** is appended to the name if the user did not supply it. The program attempts to create a new database and all of the tables that it requires. If a database with the desired name already exists, we assume that it is an IChart database and simply empty the contents of the tables before inserting all of the records. Both commands reset the cache status flag, because the cache and database are now identical. The two commands that load data check for unsaved changes to prevent them from being accidentally lost; the user is prompted to save them to the current database, discard them, or cancel the load command.

The user may wish to save all of the data as text files for archive purposes or for installation on another system. The Save As Files command prompts the user for the name of a directory, which is analogous to asking for a database name. If the directory does not exist, it is created. Instead of updating database tables, a file is created in the directory for each of the tables, replacing any existing files with the same names. The file names are the same as the corresponding table names (e.g., OrgTypes are stored in the file **`OrgType`**), and the fields are separated by commas. Strings and timestamps are surrounded by single quotes. For this reason, single quotes may not be used in the database.†

The final command in the file menu is Quit. It checks whether the cache has been modified before terminating the program. If any unsaved changes exist, the user is prompted to save them, discard them, or continue to run the program. The cache will also be checked if the user attempts to stop the program by closing the main window, but not if the process is killed.

---

*No restrictions are placed on the directory names, although a special naming convention could be used.

†A work-around exists, but was not implemented.

4.3.3 Find Menu

The Find menu permits the user to perform limited searches on the data. The Find Name command searches the appropriate tree (organization or org type, depending on which is being displayed) for a unit whose long or short name contains the desired string.* If a single unit is found, the tree is redrawn with that unit's node at the top. Figure 19 shows the dialog that is displayed when multiple units contain the string. Notice that the entries directly above the highlighted item have the same name. This is because they are the same organization type (have the same EID) and appear twice in the tree. We display both of them, so that the user may choose which node to display in the tree. Name searches are more useful with the unique names found in organizations. As more fields are added to the tables, it is likely that more Find commands will be created.

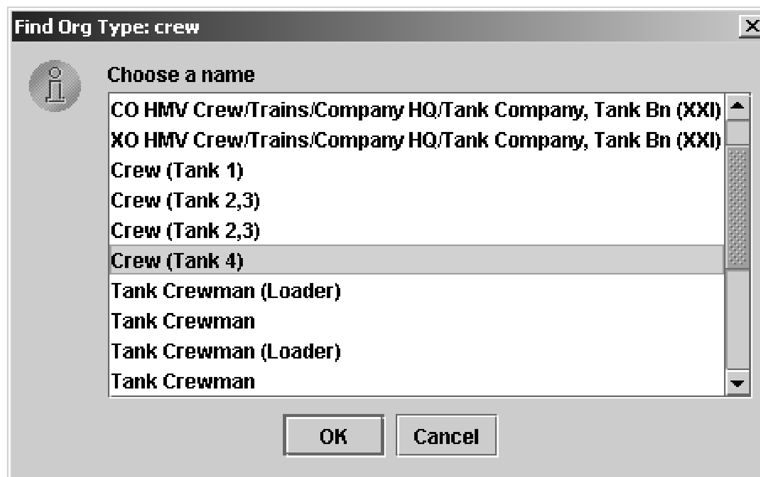


Figure 19. Find name selection dialog.

Sometimes we needed to quickly locate a particular object during our testing. The Find EID command uses the EID hash table (see section 3.5.2) to see if the user-supplied value is an EID. The object is shown in a display dialog (like the one in figure 3). If it is an organization or org type, the appropriate tree is also drawn with the unit at the top.

4.3.4 Show Menu

The Show menu was used during the development of IChart to display information about the cache that was not available through other means. The only command that is left is Show Link Tables, which opens a window with six untitled Inspectors. They contain the objects from the four association and two relation tables. IChart displays information about objects and nodes, not links, and this command was used to verify that new links were being created correctly.

*The search is not case sensitive.

4.3.5 Help Menu

Eventually, this menu will be expanded to display a detailed help system or at the very least a link to the text of this report. The About command pops up a standard dialog with a summary of the IChart application, its version number, and contact information.

4.4 Organization Tree Panel

4.4.1 Overview

When the IChart application is started, the root node of the organization type tree is displayed in the tree panel. Two tabs just under the menu bar allow the user to view either the org type or organization tree. Below each tree panel is a "sandbox" which functions as a scratch pad. The component parts of a node are shown in figure 12. The colors used by the nodes are defined in the property files. We chose to define different colors for doctrinal, crew, and position (billet) nodes. A fourth pair of colors is given to highlight a selected node. Other colors are used to denote links and nodes that have been newly created and not stored in the database.

The user may click the left mouse button on any node of a tree, causing the node to be highlighted and the display menu to appear as shown in figure 16. Clicking the right mouse button on a node opens the edit menu. The next two sections explain the operations performed by the popup menu items.

4.4.2 Display Popup Menu

The Display menu allows the user to control which portions of a tree are displayed. The first menu item, Status, contains two items in a submenu. Selecting Show Details causes the org type object and all of the materiel, skill, and person types that are aligned with it to be displayed in the tables on the right side of the window. The second item, Show Rollup, examines the subtree beginning with the selected node and constructs summary information of the data.[*] Figure 16 contains a rollup of the **`HQ`** node and its subordinates. The **Category** column has been sorted to show that the subtree contains two crew and one doctrinal org type. The **Skill Type** table lists two objects of one skill type (officer) and six of another skill type (enlisted); these were obtained from the eight positions at the bottom of the tree.

The organization tree handles details and rollups in a slightly different manner. The top table in the detail pane contains the current organization node or the list of the nodes in the subtree.[†] The link is followed from each organization to its org type, and the objects aligned with it are used as before.

The Expand item also has a submenu with two items. Expand Node causes all of the immediate children of the selected node to be drawn, while Expand Subtree causes the entire subtree to be expanded and drawn.

[*]The entire subtree is examined, not just the nodes that are being displayed.
[†]Every node in an organization tree is unique.

The opposite result may be obtained with the Collapse item. It causes the entire subtree of a node to be visually erased. Figure 20 shows the same tree as the one in figure 16 after the **WM** node has been collapsed. Notice that a stub is drawn to show that the node has one or more children.

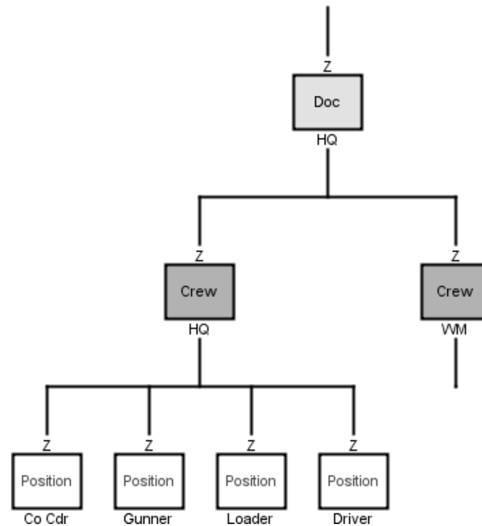


Figure 20. Partially collapsed tree.

After the user has located a node or subtree of interest, he may use Make Top to redraw the tree with the desired node at the top. The tree in figure 16 was drawn by using Expand Subtree on the root node, then selecting the **HQ** Doc node and making it the top.

The stub drawn on top of the **HQ** Doc node indicates that it is not the root of the tree and more nodes are above it. The user may move up the tree by selecting Show Parent. It redraws the tree with the expanded parent node at the top.

4.4.3 Edit Popup Menu

There are two versions of the edit menu because certain operations are not applicable to organizations. The node edit menu for an organization type is shown in figure 21. The user is changing the node's multiplier to "3."

The Create item appears only on the org type menu. The subitems are Create Org Type and Create Organization. Both commands instantiate a new object with default values for all of the fields, then get the next available EID and assign it to the object. An organization is automatically given the same category as the org type node that was selected.* The new object is displayed in an edit dialog (such as figure 4) to allow the user to fill in the remaining data fields. If the user clicks on the dialog's Cancel button, the operation is aborted, while accepting the new object causes it to be added to the cache. The object is placed in the sandbox as explained later. If a new organization is created, an OrgOrgTAssoc object is instantiated to link it to the selected org type.

*The org type category **Position** is changed to the corresponding organization category **Billet**.

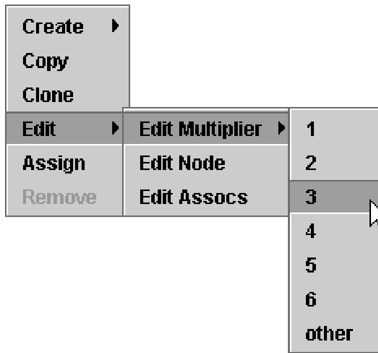


Figure 21. Edit popup menu for an organization type node.

Both menus contain a Copy item. It is basically the same as the Create command with two additions: the new object's data fields are given the values of the selected object and, after a new org type is accepted, links are created for all of the objects that are aligned with the selected org type. If the user does not make any editing changes, the new object will be identical to the old one except for having a different EID.

The Clone command simply places a copy of the current org type node into the sandbox. This allows the user to assign the same org type subtree to multiple parent nodes. Since organizations are unique, this item is not available in the organization edit menu.

The Edit item appears with both trees, but the submenu is different. The org type menu in figure 21 should be self-explanatory. Edit Multiplier allows the user to state that a node appears a certain number of times under a given parent. This is demonstrated in figure 22, where the tank company has one company headquarters and three platoons. The Edit Node item appears in both menus and opens an edit dialog on the selected org type or organization. The user may edit the org type's associations with Edit Assocs as explained in section 4.5.

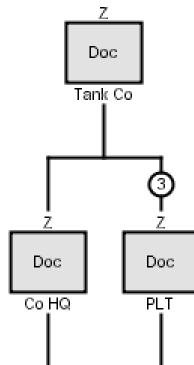
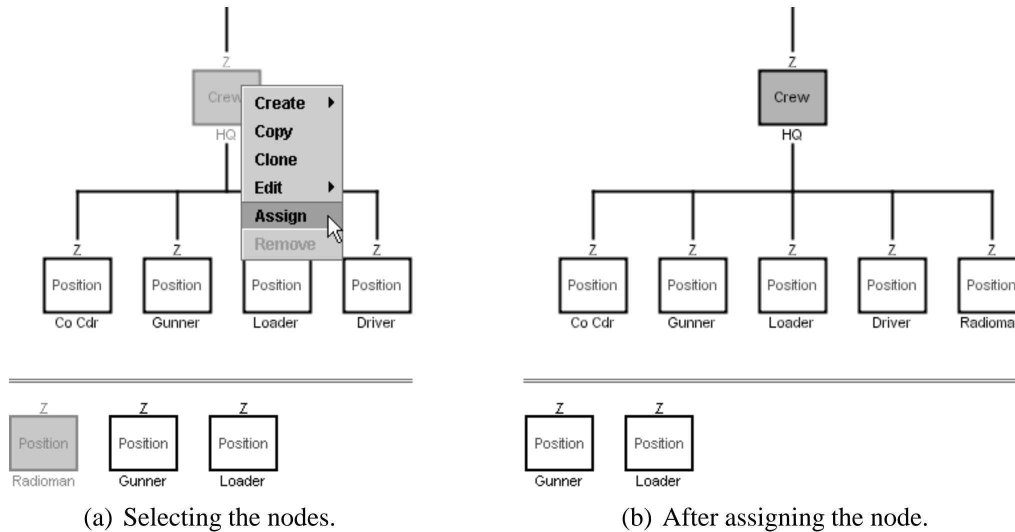


Figure 22. Subtree containing a multiplier.

The final item is Assign,* and it permits the user to construct a tree. The user places one or more objects into the sandbox with create, copy, or clone, then clicks on a node in the sandbox to select it. He then right clicks on the desired parent node in the tree and chooses Assign. A Radioman is about to be assigned to the HQ Crew in figure 23a; his node has been moved from the sandbox to the tree in figure 23b.



(a) Selecting the nodes. (b) After assigning the node.

Figure 23. Adding a node to the tree.

Some special aspects of working with organization types must be kept in mind when constructing a tree. The subtree in figure 24 contains duplicate nodes. The HQ Crew and the WM Crew are identical except one is led by the company commander and the other by the company executive officer. These four nodes are therefore different org types and have four different EIDs. On the other hand, the remaining crew members are the same in each case. The Gunner assigned to HQ is the same EID as the Gunner assigned to WM, the Loader is the same type in both cases, and the Drivers are identical. It should be apparent that editing any data fields in one Loader will be reflected in the other Loader, because they are in fact the same Loader.

What is less obvious is what happens if a child is added to a node that is repeated—each instance gets the child. Suppose that an assistant is assigned to the Loader.† A single OrgTypeRelat object is instantiated linking the Loader org type with the Assistant Loader org type. The user is stating that a typical loader of that type always has an assistant. The result is an assistant loader node will appear below every (visible) loader node. Using our sample database, the status field in IChart will state, "Tank Crewman (Asst Loader) added to 6 copies of parent Tank Crewman (Loader)." The same situation applies when a node is cloned and assigned. Not only is the node linked to all copies of its new parent, but its children go with it.

If some loaders should get assistants and some should not, then a new loader org type must be instantiated with a new EID, and once again we are beyond the scope of this project.

*An inactive Remove item appears as a future enhancement.

†He would actually be assigned to the Crew node, but, since we do not enforce policy, assume that this assignment is allowed.

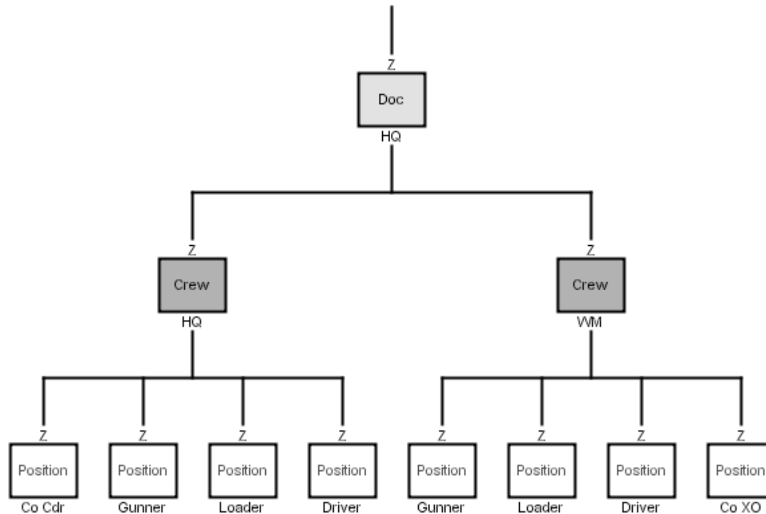


Figure 24. Subtree containing duplicate nodes.

4.5 Association Editor

The user opens the association editor window by right-clicking on a node in the organization type tree and selecting Edit Assocs. The complete window is shown in figure 17, but we will begin our discussion with the type inspectors which are reproduced in figure 25.

To align an object with the organization type displayed in the left side of the association editor window, the user selects an object on the right by clicking on the desired row. Clicking on the Align button opens an edit dialog on an association link such as the one shown in figure 7. If the user keeps the link, it is added to both the cache and the small inspector table on the left side of the window.

The next three buttons function in a manner similar to the same menu items in the edit popup menu in the org type tree. The context of each button depends on which type is visible at the time. Normally the selected tab is the color that is used for that particular type* and the other two tabs are dark gray. We chose to use the default Java colors for this report.

The Create button instantiates a new object of the desired type and displays a blank edit dialog for that type. If the user cancels the dialog, the object is disposed of; otherwise, it is added to the cache, the inspector is updated, and the table is scrolled to the bottom to show the new object. The Copy button works the same way, except first the user must click on an object in the inspector to select it, and its fields are used as the values for the new object. The Edit button opens an edit dialog on an existing object which has been selected.

The last button is Close, and it disposes of the association editor window.† The program prevents the user from opening more than one window to avoid confusion when the various types are being created or edited.

*The type headers on the left side of the window use the same colors.
†The user may safely close the window manually instead of using the button.

Materiel Type Table | Skill Type Table | Person Type Table

	EID	LIN	Name
o	100000020	M12418	MASK CHEMICAL BIOLOGICAL: M40
o	100000021	M18526	MASK CHEMICAL BIOLOGICAL: COM...
o	100000022	R95035	RIFLE 5.56 MILLIMETER: M16A2
o	100000023	R97234	RIFLE 5 56 MILLIMETER: M4
o	100000024	P98152	PISTOL 9MM AUTOMATIC: M9
o	100000025	B49272	BAYONET-KNIFE: W/SCABBARD FOR...
o	100000026	B67766	BINOCULAR: MODULAR CONSTRUC...
o	100000027	L44595	LAUNCHER GRENADE 40 MILLIMET...
o	100000028	N05482	NIGHT VISION GOGGLE: AN/PVS-7B
o	100000029	R44999	RADIO SET: AN/VRC-89F(C)
o	10000002a	R67296	RADIO SET: AN/VRC-87F(C)
o	10000002b	R45543	RADIO SET: AN/VRC-92F(C)
o	10000002c	B90494	BORESIGHTING EQUIPMENT WEAP...
o	10000002d	C89070	CAMOUFLAGE SCREEN SUPPORT S...
o	10000002e	C89145	CAMOUFLAGE SCREEN SYSTEM: W...
o	10000002f	D12087	CARRIER PERSONNEL FULL TRACK...
o	100000030	D78555	DATA TRANSFER DEVICE: AN/CYZ-10

Align | Create | Copy | Edit | Close

Figure 25. Association editor type inspectors.

After the user has aligned an object with an organization type, he must be able to edit the link. We provided three edit buttons on the left for this purpose. The user selects the desired row in the proper detail table, then clicks on the corresponding edit button, causing the edit dialog to appear. If he changes the multiplier, the table is updated to show the new value. The user may define the color that is used for each association as shown in table A-1 in appendix A. The color is used for each edit button and its corresponding dialog. We chose to use darker shades of the same colors that we used for the type objects to remind the user of the relationship (e.g., **MatType.color** is bright cyan while **OrgTMatTAssoc.color** is dark cyan).

5. Future Development

This is the initial version of a program to manipulate force management trees that are stored in a relational database. The application must be made more robust before it may be deployed. We did not implement policy enforcement because the various services have different policies. While we used U.S. Army data, the tool is intended for use by all of the services. It may be desirable to have modes that implement different policies.

The user interface will be refined as more people use the application. Two prototype changes to TreePanel are a way to scale the tree to fit in the current window and code to draw MIL-STD-2525B symbols for each node. Both will be implemented in the next major release.

The next phase of the project will include changing the database schema to support the Global Force Management Information Exchange Data Model (*13*), which is an augmented subset of the C2IEDM. Many of the tables in this model have an hierarchical relationship versus the flat structure of our current database schema. Many more data fields will be added beyond the minimal fields required to test the program.

New features will include the ability to attach one organization or org type to another to create task forces and other temporary force structures. The user will specify the duration of the attachment by defining date/time intervals. Another feature will be a way to generate a new database instead of the current requirement that the user create one manually.

During our testing, we discovered that users want the ability to delete objects. This goes against one of our basic tenets, which is that all data is kept for archival purposes. However, it is easy to make mistakes when building a tree, and the proposed compromise is to allow the user to undo his actions until the data is stored in the database. The first step toward implementing this has been to use a special color to denote new nodes and links in the trees.

6. Conclusion

The IChart application has proven to be an effective tool for building and maintaining force management structures. It has been successfully tested both within the U.S. Army Research Laboratory and by other organizations, and it is being used to construct actual GFM data for another project. IChart will serve as a model for future EID applications and databases.

7. References

1. Sun. Java 1.4.2. http://java.sun.com/j2se/1.4.2/download.html (accessed February 2003).
2. MySQL AB. MySQL Version 4.0.18. http://www.mysql.com/downloads/mysql-4.0.html (accessed February 2004).
3. MySQL AB. MySQL Connector/J Version 3.0.9. http://www.mysql.com/downloads/api-jdbc-stable.html (accessed October 2003).
4. Arnow, D.; Weiss, G. *Introduction to Programming Using Java*; Addison Wesley Longman, Inc.: New York, NY, 2000.
5. Multilateral Interoperability Programme Data Modelling Working Group. *Overview of the C2 Information Exchange Data Model (C2IEDM)*, 6.1 ed.; MIP Data Modelling Working Group: Greding, Germany, November 2003.
6. Chamberlain, S. Enterprise Identifiers for Global Naming Across the C4I-Simulation Boundary. *Proceedings of the 2001 Spring Simulation Interoperabiltiy Workshop*, Orlando, FL, 25–30 March 2001.
7. Chamberlain, S.; Leeds, C. Time-Based Tree Graphs for Stabilized Force Structure Representations. *Proceedings of the* 8th *International Command & Control Research & Technology Symposium*, Washington, DC, 17–19 June 2003.
8. Brundick, F.; Hartwig, G., Jr. *A Primary Server for Organizational Identifiers*; ARL-TR-2530; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, 2001.
9. Walrath, K.; Campione, M. *The JFC Swing Tutorial*. Addison-Wesley: Reading, MA, 1999.
10. Duguay, C. An Easy Way to Sort Things Out. *Java Pro*, August 2002.
11. Smith, K. *The Information Distribution System: Org Chart—An Organization Display Application Program*; BRL-MR-3903; Ballistic Research Laboratory: Aberdeen Proving Ground, MD, 1991.
12. MIL-STD-2525B. *Common Warfighting Symbology* **1999**.
13. Joint Data Support. Global Force Management Initiative. http://jds.pae.osd.mil/GFM/GFMTOC.htm (accessed August 2004).

INTENTIONALLY LEFT BLANK.

Appendix A. Property Files

When IChart is run, it attempts to load program parameters, or properties, from the files **`default.properties`** and **`ichart.properties`**. If it is run from a Java ARchive (JAR) file and the JAR contains the default property file, the defaults are read from that source. If a JAR file is not used, or it does not contain the default property file, the application looks for a local file named default.properties. The program then looks for parameters in the local file named ichart.properties. All values in this file override the corresponding settings in the default file.

Since it is much more convenient to run the application from a single JAR file, an organization may extract the default.properties file, edit it, and replace it in the JAR file. Settings that are unique to specific users or groups could be defined in a local ichart.property file for each user.

A property file consists of multiple lines, each of which is composed of a keyword, colon or equal sign separator, and a value. Whitespace is allowed around the separator for readability and blank lines are ignored. Lines starting with a sharp sign (`#`) are comment lines to document the property file. Table A-1 shows the contents of a typical default.properties file, and table A-2 shows the contents of ichart.properties.

The properties for the Java Database Connectivity (JDBC) **driver** and **URL** should not be changed unless an RDBMS other than MySQL is used. If a different system is used, not only will these lines need to be edited, but the class **SQLUtility** will also need to be modified.

There are restrictions on the values of some of the properties. The colors are given as 6 hexadecimal digits denoting a 24-bit RGB (red, green, blue) value. The JDBC **server** should be the host name or its Internet IP address followed by a slash. If the application is being run on the same host as the MySQL server (like our laptop-based demo), the name **`localhost/`** may be used. The **database** is the name of the database to be loaded initially, although a different name may be given on the program command line. Likewise, the date of interest is given via the **datetime** property or on the command line. It is of the form shown, which is the 4-digit year and 2-digit month and day, all separated by dashes. The time has the fields separated by colons. IChart currently has no way of allowing the user to manipulate the time, so the file should contain `00:00:00`.*

The **user** property is a user name recognized by the MySQL server. We recommend not storing the **password** in the property file for security reasons. If either the user name or password is not given, the application will display a login dialog.

*The time of zero is appended to a date supplied on the command line.

Table A-1. Contents of file default.properties.

```
# IChart default properties

# JDBC (MySQL) parameters
jdbc.driver = org.gjt.mm.mysql.Driver
jdbc.URL = jdbc:mysql://

# GUI parameters
OrgType.color = FFC800
MatType.color = 00FFFF
SkillType.color = FFFF00
PersonType.color = FFBFBF
OrgTMatTAssoc.color = 00CCCC
OrgTSkillTAssoc.color = DDDD00
OrgTPersonTAssoc.color = DD8D8D
Org.color = 00FF00
#OrgTypeRelat.color =
#OrgOrgTAssoc.color =
#OrgRelat.color =

# Node and link colors
node.selected.foreground.color = 888888
node.selected.background.color = FFAFAF
node.doc.foreground.color = 000000
node.doc.background.color = FFFF00
node.crew.foreground.color = 000000
node.crew.background.color = 00FFFF
node.position.foreground.color = FF0000
node.position.background.color = FFFFFF
#node.new.color =
#link.new.color =
```

Table A-2. Contents of file ichart.properties.

```
# IChart local properties

# JDBC (MySQL) parameters
jdbc.server = some_host/
jdbc.database = some_database
#jdbc.datetime = yyyy-mm-dd hh:mm:ss
jdbc.user = some_user
jdbc.password = some_password
```

Appendix B. IChart Installation

B.1 System Requirements

We assume that the person performing the IChart installation has a working knowledge of Java and MySQL. The current version of the IChart application requires Java 1.4 or newer, MySQL 4.0, and the Java Database Connectivity (JDBC)/MySQL driver. The Java 2 Platform, Standard Edition (J2SE) Java Runtime Environment (JRE) is sufficient if the precompiled IChart Java ARchive (JAR) file is used, while the J2SE Software Development Kit (SDK) is needed to build the application from its source files. These may be obtained from Sun's Java web site.[1] We did not use any Java extensions.

The application needs access to a MySQL 4.0 server running on the same machine as IChart or on a remote host. It may be downloaded from the MySQL web site[2] and must be installed and configured by a system administrator.

We created our own web site to distribute the IChart application, the MySQL driver, and a sample MySQL database. Please contact the authors if you need access to our site.

B.2 Building the Application from Source Files

The IChart application may be obtained as either source files or as precompiled Java class files in a single JAR file. To install the source version, get the file **`ichartsrc.zip`** and extract all of the component files into the directory of your choice. This may be done with the command "**`jar xvf ichartsrc.zip`**" or with your favorite ZIP program.[*] The primary directory will receive the files listed in table B-1, and the subdirectory tree starting with **`src`** will be created and the source files stored in it.

The application may be built with either **`make`**[3] or **`ant`**.[4] We originally started with **`make`** and later switched to **`ant`**; we recommend that you use **`ant`** if you have it. Entering either command without any arguments will create the **`build`** subdirectory and compile the files into it.

The installer should edit the file **`default.properties`** and define organization-wide properties as explained in appendix A. Invoking **`make`** or **`ant`** with the target **`dist`** creates subdirectory **`dist`** and constructs **`ichart.jar`** there. This new JAR file may be copied to other computers and installed as described in the next section.

[1] Sun. Java 1.4.2. http://java.sun.com/j2se/1.4.2/download.html (accessed February 2003).

[2] MySQL AB. MySQL Connector/J Version 3.0.9. http://www.mysql.com/downloads/api-jdbc-stable.html (accessed October 2003).

[*] Java JAR files use the ZIP format.

[3] Free Software Foundation, Inc. make v3.8.0. http://www.gnu.org/software/make/make.html (accessed October 2003).

[4] The Apache Software Foundation. Apache Ant 1.6.1. http://ant.apache.org/bindownload.cgi (accessed June 2004).

Table B-1. Files in primary directory.

File Name	Contents
Makefile	Makefile for IChart application
build.xml	Ant version of Makefile
Readme	usage, limitations, to do list
default.properties	default property file for IChart
ichart.properties	property file for IChart application
runchart	shell script to run app
runjar	shell script to run app from a JAR file
runchart.bat	DOS batch file to run app
runjar.bat	DOS batch file to run app from a JAR file

The target `docs` generates the HTML API files in the new subdirectory `api`. To display a complete list of targets and what they do, run the commands "`make help`" or "`ant -p`." There are several development and testing targets in both the `Makefile` and `build.xml` files in addition to those just mentioned, and they may be ignored.

B.3 Configuring the Application

Simple scripts and batch files were written to run the application. The installer may either build the environment which matches the one that we used or edit the scripts to match the current environment. We will explain the former approach, because the latter should be apparent to anyone who is familiar with writing scripts.

The first step is to create the remaining directories as shown in table B-2. If the application was built as explained in the previous section, the file `ichart.jar` is already in the `dist` directory. If the installer obtained a prebuilt JAR file from us, or built the application himself and copied it to another machine, he must manually create the `dist` directory and copy the JAR file into it. Next, the JDBC driver must be put in the proper location. Create the `lib` directory and copy the MySQL driver JAR file* into it.

Table B-2. Directory structure.

Directory	Contents
api	HTML API javadoc files (optional)
build	root of binary (compiled) subtree
dist	distribution directory for JAR and ZIP files
lib	JAR files needed by this application
src	Java source files for the application

The `ichart.properties` file must define any properties that were not specified in `default.properties`. If IChart is being installed on multiple computers, this file is probably slightly different on each one.

*We used `mysql-connector-java-3.0.9-stable-bin.jar`.

B.4 Running the Application

If the application was compiled from the source files, the UNIX or Windows scripts named **`runchart`** may be used, while **`runjar`** is the script to use with the **`ichart.jar`** file. The Windows scripts expect the environment variable **JAVA_HOME** to be defined; it is also used by **`ant`** in both operating systems.

Do not double-click on the **`runchart`** or **`runjar`** icons in Windows. If an error occurs, the diagnostic messages will be lost. Open a *Command Prompt* window and **`cd`** to the folder where you installed IChart.* Manually type **`runchart`** or **`runjar`** with optional command line arguments to run the application.

B.5 Common Errors

If you see an error message that says, "**`Exception in thread "main"`** ... **`getTimeInMillis()J from class`** ...," the problem is you are attempting to use the Java 1.3 runtime system. Make sure that JAVA_HOME points to the Java 1.4 (or newer) tree. IChart will not compile or run under Java 1.3, although we tested it with both 1.4 and 1.5 beta.

The IChart message, "**`Error: database name must be defined in a property file or on the command line.`**" means that IChart was not given the name of the database to use. The quick fix is to include the name of the database on the command line (e.g., **`runjar tankchart`**), while the better solution is to edit ichart.properties and define **jdbc.database**.

*Or open the window with the folder as its target.

Glossary

Object Oriented Programming Terms

abstract an abstract class contains one or more abstract methods, which is a method declaration without a definition.

camel casing names are constructed by capitalizing the first letter of multiple words and concatenating them together, e.g., “org type long name” becomes “orgTypeLongName.”

constructor a method that is invoked when an object is created. It initializes the object’s instance variables.

hash table a data structure which maps key objects to value objects. The Java class that implements this is HashTable.

hexadecimal a number written in base-16. The values 10–15 are represented by the letters A–F.

inheritance a technique in which a subclass assumes the attributes and behaviors of its superclass and adds new capabilities. Java uses single inheritance where a subclass may have only one direct superclass.

instance variable a variable that is declared within a class but outside of its methods. Each object that is created has its own unique instance variables.

instantiate to create an instance of an object.

JAR file a Java ARchive file is a collection of class and data files stored with the ZIP format.

member general term for a variable or method contained in a class.

overloading a class has multiple methods with the same name that perform similar operations but are distinguished by having different argument lists.

overriding the act of reimplementing a method in a subclass with the exact same signature of a method in its superclass.

polymorphism the exact method to be invoked is determined at run time by the class of the object.

private a member of a class that may not be directly accessed from another class.

property list a hash table of keyword/value pairs of strings stored in an instance of the Java class Properties.

public the opposite of private, a member that may be accessed from any class.

result set the set of records generated by executing a query statement. The Java class that implements this is ResultSet.

signature a method's name along with the number, order, and type of its arguments.

static method a method that is not associated with any particular object, but with the class itself. Therefore, it may be invoked without a reference to an object. Also known as a **class method**.

subclass the direct or indirect child of a superclass.

superclass any class which acts as the direct or indirect parent of another class.

Terms Used in IChart

align associate a materiel, skill, or person type with an org type.

assign define an organization or org type as the organic (habitual) parent of another organization or org type.

association the "horizontal" link between an org type and its materiel, skill, and person types; also the link between an organization and its org type. The **OrgTXXXTAssoc** and **OrgOrgTAssoc** tables contain associations.

attach assign an organization or org type to another on a temporary basis, e.g., to create a task force.

clone put an org type in the sandbox so it may be assigned or attached to another org type in the tree. A new EID is *not* created. This allows the user to reuse an org type, e.g., assign the same gunner type to several crews.

copy make a new organization or org type with a new EID and the same field values and associations as an existing organization or org type. The user may also copy a materiel, skill, or person type.

create make a new org type with a new EID, default values, and no associations, or make a new organization with default values and an association with an existing org type. The user may also create a materiel, skill, or person type.

date/time group a specific date and time in the standard UNIX epoch of 1 Jan 1970 through 31 Dec 2037. It is stored as a timestamp in the database and a Timestamp in Java.

EID an enterprise identifier is a surrogate key that is guaranteed to be unique.

instance a specific or concrete item, e.g., a particular M1 tank with a certain bumper number.

org type abbreviation for "organization type." This report never abbreviates organizations, only organization types.

relation the “vertical” link between an organization or org type and each of its children. The **OrgRelat** and **OrgTypeRelat** tables contain relations.

sandbox the scratch pad area where organizations and org types are placed before being assigned or attached to a node in a tree.

tree a data structure that contains an organization or org type hierarchy.

type a template or generic description, e.g., an org type could be a mechanized platoon which defines the attributes that are common to all mechanized platoons.

NO. OF COPIES	ORGANIZATION
1 (PDF only)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA 8725 JOHN J KINGMAN RD STE 0944 FORT BELVOIR VA 22060-6218
1	US ARMY RSRCH DEV & ENGRG CMD SYSTEMS OF SYSTEMS INTEGRATION AMSRD SS T 6000 6TH ST STE 100 FORT BELVOIR VA 22060-5608
1	DIRECTOR US ARMY RESEARCH LAB IMNE ALC IMS 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRD ARL CI OK TL 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRD ARL CI OK T 2800 POWDER MILL RD ADELPHI MD 20783-1197

ABERDEEN PROVING GROUND

NO. OF COPIES	ORGANIZATION
1	DIR USARL AMSRD ARL CI OK TP (BLDG 4600)

NO. OF COPIES	ORGANIZATION
1 (CD ONLY)	JOINT STAFF J-8 MASO G SPRUNG RM 2C646 JOINT STAFF PENTAGON WASHINGTON DC 20318-8000
1	DIR USARL AMSRD ARL CI J GOWENS 2800 POWDER MILL RD ADELPHI MD 20783-1197

ABERDEEN PROVING GROUND

NO. OF COPIES	ORGANIZATION
6	DIR USARL AMSRD ARL CI C B BROOME AMSRD ARL CI CT F BRUNDICK S CHAMBERLAIN H INGHAM M MITTRICK M THOMAS