# PSYCHO: A GRAPHICAL LANGUAGE FOR SUPPORTING SCHEMA EVOLUTION IN OBJECT-ORIENTED DATABASES

Hyoung-Joo Kim and Henry F. Korth

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-43 November 1987

# PSYCHO: A GRAPHICAL LANGUAGE FOR SUPPORTING SCHEMA EVOLUTION IN OBJECT-ORIENTED DATABASES

*Hyoung-Joo Kim, Henry F. Korth*

**Department of Computer Sciences**
**The University of Texas at Austin**
**Austin, Texas 78712**

**ABSTRACT**

One of the important requirements of non-conventional applications such as CAD/CAM, AI, and OIS (office information systems) with multimedia documents is *schema evolution*, that is, the ability to make a wide variety of changes to the database schema dynamically. We provided a framework of schema evolution in [BKKK86,BKKK87] and the framework was realized in an object-oriented database system, ORION at MCC. Schema modifications using line-oriented interaction are difficult for the user to manage if class lattices are complicated. The difficulty is even greater because there are a large number of types of schema modifications. We have designed and implemented a powerful, yet user-friendly graphical interface PSYCHO (*Pictorial Schema-editor Yielding Class Hierarchies* and *Objects*) for supporting schema modification in object-oriented databases. In this paper, we give a brief overview of our schema evolution framework, and then discuss both the use of PSYCHO and its implementation.

**Key Words:** Graphical Language, Graphical Interface, Object-oriented Databases, Schema Evolution, Class Lattices, Objects

## 1. Introduction

The successful use of database management systems in data-processing applications has created a substantial amount of interest in applying database techniques to such areas as knowledge bases and artificial intelligence [STEF86], computer-aided design (CAD) [AFSA86], and office information systems [IEEE85, AHLS84, WOEL86]. In order to provide the additional semantics necessary to model these new applications, many researchers, including those referenced above, have adapted the object-oriented programming paradigm [GOLD81, GOLD83, BOBR83, CURR84, SYMB84] to serve a data model.

In order to use the object-oriented approach in a database system, it was necessary to add persistence and sharability to the object-oriented programming paradigm. Several database systems based on this approach are under implementation, including Gemstone [MOP85] and Iris [FISH87]. In this paper, we focus our attention on the ORION system [BAN87], developed within the database program at MCC. ORION includes several features to support CAD, artificial intelligence, and office information systems [KIM85, WOEL86]. Among its features is support for *schema evolution*, which allows users to modify the database schema dynamically.

We established a taxonomy of 34 useful schema change operations under the ORION data model. We defined the semantics of each schema change. Our framework of schema evolution has been reported in [BKKK86,BKKK87]. The entire set of schema change operations we defined in our taxonomy has been implemented in ORION.

† Research partially supported by NSF Grant DCR-8507724

Schema modifications using line-oriented interaction are difficult for the user to manage if class lattices are complicated. The difficulty is even greater because there are a large number of types of schema modifications. We have designed and implemented a graphical language PSYCHO for supporting user friendly and powerful schema modification in object-oriented databases. In this paper, we present a brief overview of our schema evolution framework and describe in detail the structure of PSYCHO. First we need to overview the ORION data model briefly.

## *1.1 The ORION Data Model*

The ORION data model supports the usual features of object-oriented languages, including the notions of classes, subclasses, and objects. Each entity in an ORION database is an *object.* Objects include *instance variables* that describe the state of the object. Instance variables may themselves be objects with their own internal state, or they may be *primitive objects* such as integers and strings which have no instance variables. Objects also include *instance methods* which contain code used to manipulate the object or return part of its state. These methods are invoked from outside the object by means of *messages.* Thus, the public interface of an object is a collection of messages to which the object responds by returning an object.

Although each object has its own set of instance variables and methods, several objects may have the same *types* of instance variables and the same methods. Such objects are grouped into a *class* and are said to be *instances* of the class. Instance variables and instance methods shared by all members of a class may be referred to as *class variables* and *class methods* respectively.

Usually each instance of a class has its own instance variables. If, however, all instances must have the same value for some instance variable, that variable is called a *shared-value variable.* A default value can be defined for a variable. This value is assigned to all instances for which a value is not specified. Such variables are called *default-valued variables.*

The *domain* of an instance variable is a class. The ORION data model allow the domain of an instance variable to be bound to a specific class and all subclasses of the class.

Each of instance variables and instance methods has a *document* that is a comment on itself. So do class variables and class methods.

Similar classes are grouped together into *superclasses.* The result is a directed acyclic graph (DAG) containing an edge $(C_1, C_2)$ if class $C_1$ is a superclass of $C_2$. A class inherits properties (instance variables and methods) from its immediate superclasses, and thus, inductively, from every class $C$ for which a path exits to it from $C$. The class-superclass relationship $(C_1, C_2)$ is an "ISA" relationship in the sense that every instance of a class is also in instance of the superclass. Using the terminology of the entity-relationship model (see, e.g., [KS86]), we say that $C_1$ is a *generalization* of $C_2$ and $C_2$ is a *specialization* of $C_1$.

Because we allow use of a DAG to represent the ISA relationship among classes, it is possible for a class to inherit properties from several superclasses. This is called *multiple inheritance* in [GOLD83, STEF86]. This leads to possible naming conflicts between properties inherited from superclasses. Another source of conflict is the possibility that a locally-defined class variable or method has the same name as an inherited property. These conflicts are resolved by giving the local definition precedence. Other conflicts are resolved based upon a user-supplied total ordering of the superclasses. This ordering can be changed at any time by the user. Furthermore, the user may override the default conflict resolution scheme by either renaming or by explicit choice of the property to be inherited.

In the remainder of this paper, we refer to the class DAG as a *lattice*, so as to conform with accepted practice in the literature. These "lattices" do not necessarily have any of the properties associated with the mathematical concept of a lattice.

## 2. An Overview of Schema Evolution

Our framework for schema evolution under the ORION data model consists of a set of properties of the schema called *invariants*, and a set of *rules* that guide the selection of the most meaningful way of preserving the invariants. The invariants must hold at every quiescent state of the schema, that is, before and after a schema change operation. They guide the definition of the semantics of every meaningful schema change, by ensuring that the change does not leave the schema in an inconsistent state (one that violates an invariant). However, for some schema changes, the schema invariants can be preserved in more than one way. The set of rules that we have guides the selection of one most meaningful way.

In our previous papers [BKKK86,BKKK87], we showed that how schema evolution rules are applied to maintain the schema evolution invariants to all types of schema changes in ORION. A more formal treatment of schema evolution is in [KKBK86]. As such, we omit the details of invariants and the schema transformation rules here.

We emphasize that, although our framework has been developed for the particular data model of ORION, we believe that our methodology for the development of the framework is applicable to most object-oriented systems. This is because the ORION model has incorporated all the basic object concepts for which there is a wide acceptance, and has enhanced the basic object-oriented model with the notion of composite objects.

### *2.1 Taxonomy of ORION Schema Change Operations*

In this subsection, we classify all schema changes that we support in ORION, and define the semantics of schema changes, using our schema evolution invariants and rules. Changes to the class lattice can be broadly categorized as (1) changes to the contents of a node, (2) changes to an edge, and (3) changes to a node. Our schema change taxonomy is as follows:

```
(1) Changes to the contents of a node (a class)

 (1.1) Changes to an instance variable

  (1.1.1) Add a new instance variable to a class
  (1.1.2) Drop an existing instance  variable from a class
  (1.1.3) Change the name of an instance variable of a class
  (1.1.4) Change the domain of an instance variable of a class
  (1.1.5) Change the inheritance (parent) of an instance variable
          (inherit another instance variable with the same name)
  (1.1.6) Change the document of an instance variable
  (1.1.7) Change the default value of an instance variable
  (1.1.8) Manipulate the shared value of an instance variable
     (1.1.8.1) Add a shared value
     (1.1.8.2) Change the shared value
```

(1.1.8.3) Drop the shared value

(1.1.9) Drop the composite link property of an instance variable

(1.2) Changes to an instance method

(1.2.1) Add a new instance method to a class

(1.2.2) Drop an existing instance method from a class

(1.2.3) Change the name of an instance method of a class

(1.2.4) Change the code of an instance method in a class

(1.2.5) Change the inheritance (parent) of an instance method (inherit another method with the same name)

(1.2.6) Change the document of an instance method

(1.3) Changes to a class variable

(1.3.1) Add a new class variable to a class

(1.3.2) Drop an existing class variable from a class

(1.3.3) Change the name of a class variable of a class

(1.3.4) Change the domain of a class variable of a class

(1.3.5) Change the inheritance (parent) of a class variable (inherit another class variable with the same name)

(1.3.6) Change the document of an class variable

(1.3.7) Drop the composite link property of an class variable

(1.4) Changes to a class method

(1.4.1) Add a new class method to a class

(1.4.2) Drop an existing class method from a class

(1.4.3) Change the name of a class method of a class

(1.4.4) Change the code of a class method in a class

(1.4.5) Change the inheritance (parent) of a class method (inherit another class method with the same name)

(1.4.6) Change the document of a class method

(2) Changes to an edge

(2.1) Make a class S a superclass of a class C

(2.2) Remove a class S from the superclass list of a class C

(2.3) Change the order of superclasses of a class C

(3) Changes to a node

(3.1) Add a new class

```
(3.2) Drop an existing class
(3.3) Change the name of a class
```

## *2.2 Semantics of ORION Schema Change Operations*

Below we present the semantics of schema change operations informally. See [BKKK86, BKKK87] for further details.

- **(1.1.1) Add a new instance variable to a class C:** The new instance variable, in case of a conflict with an already inherited instance variable, will override the inherited instance variable. In that case, the inherited variable must be dropped from C, and replaced with the new instance variable; and existing instances of C will take on the value nil or the user-specified default value for the new instance variable. If C has subclasses, they will inherit the new instance variable of C. If there is a conflict with an inherited variable they have already defined or inherited, the new instance variable is ignored. If there is no conflict, the subclasses will inherit the new variable, together with a default value, if any.
- **(1.1.2) Drop an instance variable V from a class C:** The instance variable V is dropped from the definition (and from the instances) of the class C. C may inherit V from another superclass, if there had been a name conflict involving V. All subclasses of C will also be affected if they had inherited V. In case V must be dropped from C or any of its subclasses without a replacement, existing instances of these classes lose their values for V.
- **(1.1.3) Change the name of an instance variable V of a class C:** We take the view that name changes are made primarily to resolve conflicts, and as such they should not introduce new conflicts. Therefore, if a name change causes any conflict within the class C, the change is rejected. If the name change is accepted, it is propagated to subclasses of C that have inherited from V of C. The name change is required to be propagated only if it does not give rise to new conflicts in the subclasses. Further, name change propagation is inhibited in the subclasses that have explicitly changed the name of their inherited instance variable V.
- **(1.1.4) Change the domain of an instance variable V of a class C:** The domain of an instance variable is itself a class. The domain, class D, of an instance variable V of a class C may be changed only to a superclass of D. The values of existing instances of the class C are not affected in any way. If the domain of an instance variable V must be changed in any other way, V must be dropped, and a new instance variable must be added in its place.
- **(1.1.5) Change the inheritance (parent) of an instance variable:** (Inherit another instance variable with the same name) As discussed earlier, if two or more superclasses of a class C have an identically named variable (either through inheritance or local definition), the system selects only one of them for inheritance by C, based on the order in which the superclasses have become associated with C. The user can override this default explicitly.

  If C has instances, the present values of the conflicting instance variable V must be dropped, and replaced by any default value under the new definition. If C has subclasses which had inherited V, they will now inherit the new definition. Consequently, their instances will be subjected to the same changes as those for the instances of C.
- **(1.1.6) Change the document of an instance variable V of a class C:** Document change is propagated to subclasses of C that have inherited from V from C. However, document change propaga-

tion is inhibited in the subclasses that have explicitly changed the document of their inherited instance variable V.

- **(1.1.7) Change the default value of an instance variable V of a class C:** All instances of C, for which no value has been supplied for the variable V, already have a default value or nil. They will now get the new default value. If there exists any subclass of C which had inherited V from C, it must also inherit this new default value, unless that subclass has redefined the default value of V.
- **(1.1.8.1) Add the shared value of a variable V of a class C:** This operation converts a non-shared-value instance variable V to a shared-value instance variable. If V already had a shared value, then all instances of the class C receive the new value. If V was not previously a shared-value variable, it now becomes one, and all instances of C will take on this new value, dropping any existing values for V in existing instances of C.

  If C has subclasses which had inherited V, they will now inherit the new shared value of V, unless they have redefined the value.
- **(1.1.8.2) Change the shared value of a variable V of a class C:** This operation replaces the shared value of V with a new one. If C has subclasses which had inherited V, they will now inherit the new shared value of V, unless they have redefined the value.
- **(1.1.8.3) Drop the shared value of a variable V of a class C:** This operation changes a shared-value instance variable V to a non-shared one. V will now have a default value of nil. If C has subclasses which had inherited V, they will now drop the shared value of V, unless they have redefined the shared value.
- **(1.2) Change an instance method** The semantics for operations 1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5 and 1.2.6 are easily inferred from 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, and 1.1.6 respectively.
- **(1.3) Change a class variable** The semantics for operations 1.3.1, 1.3.2, 1.3.3, 1.3.4, 1.3.5 and 1.3.6 are easily inferred from 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, and 1.1.6 respectively.
- **(1.4) Change a class method** The semantics for operations 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5 and 1.4.6 are easily inferred from 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, and 1.1.6 respectively.
- **(2.1) Make a class S a superclass of a class C:** S is made the last superclass in the list of superclasses of C. C will now inherit the variables and methods from S. If this causes any name conflict, the system will ignore the instance variable (or method) of S in conflict, because C must already have inherited the conflicting variable (or method) from some of its current superclasses. The user may explicitly specify alternate conflict resolution.

  If C has subclasses, immediate or indirect, they also inherit instance variables and methods from S. Such inheritance may cause new name conflicts, but they will also be ignored. (Once again, it is up to the user to explicitly specify conflict resolutions that will override the default.)

  C and its subclasses may have existing instances. Since the instance variables they inherit from S are new to these instances, they appear in the instances with the value nil or any default value specified.
- **(2.2) Remove a class S as a superclass of the class C:** The variables (or methods) inherited from S are dropped from the definition of C. C may newly inherit the dropped instance variables (or methods) from other superclasses, if there had been name conflicts involving them. The instances of C are also modified as discussed earlier for the dropping of a class. All subclasses of C will also be affected similarly if they had inherited variables (or methods) from S via C.
- **(2.3) Change the order of the superclasses of a class C:** This alters the default conflict resolution with respect to the class C. Conflicting variables and methods will now be inherited according to the

new permutation of superclasses. If the definition of a variable changes because of this new permutation, existing instances of class C will be affected as well. The value that each variable takes on is the default value under the new definition. Subclasses of C are affected similarly.

- **(3.1) Define a new class C:** The new class C may be created as a specialization of an existing class or classes. These latter classes can be specified as the superclasses of the new class. The variables (or methods) specified for C will override any conflicting instance variables (or methods) inherited from the superclasses. If there is a name conflict involving the variables (or methods) that C inherits from its superclasses, default conflict resolution is used, unless the user explicitly overrides it.
  The class C may also be defined without any superclasses. In this case, C is made a subclass of Object which is a system defined class. Conceptually the Object class is a root node of every class hierarchy. The user may, at a later time, add superclasses for C, in which case Object will no longer be an immediate superclass of C.
- **(3.2) Drop a class C:** Whenever a class definition is dropped, all its instances are deleted automatically, since instances cannot exist outside of a class. However, subclasses of C, if any, are not dropped. Subclasses of C will lose C as their superclass; however, they will gain C's superclasses as their immediate superclasses. Further, when a class C is dropped, its subclasses will lose the instance variables and methods they had previously inherited from C. If, in the process, a subclass of C loses a variable V (or a method) which was selected over a conflicting variable in another superclass of that subclass, it will now inherit the alternative definition of V. Consequently, the instances of any such subclass will lose their present values for V, and inherit the default value (or nil) under the new definition of V.
  When an instance of the class C is deleted, all objects that reference it will now be referencing a non-existent object. The user will need to modify those references when they are encountered. ODBS will not automatically identify references to non-existent objects, because of the performance overhead.
  If the class C being dropped is presently the domain of an instance variable V1 of some other class, V1's domain becomes the first superclass of the class C. Of course, the user has the choice of specifying a new domain for V1.
- **(3.3) Change the name of a class:** If the new name is unique among all class names in the class lattice, the name change is allowed. This name change is not propagated.

## 3. System Structure of PSYCHO/ORION Environment

PSYCHO is a graphical schema manipulation language which is being used in an object-oriented system ORION. PSYCHO and ORION have been implemented within the ODBS project at MCC. ORION is written in CommonLisp on a Symbolics 3640. PSYCHO is implemented (roughly, 3000 lines) with Flavors and ZetaLisp on a Symbolics 3640. Figure 2 shows a diagram of the PSYCHO/ORION environment.

The user can use ORION system directly or indirectly through the PSYCHO system. PSYCHO communicates with the schema manager and the transaction manager of ORION.

In the PSYCHO/ORION environment, class lattices are represented as DAGs (directed acyclic graphs) on the screen, and the user can manipulate DAGs directly using a mouse and pop-up menus. A schema manipulation session of PSYCHO/ORION environment goes roughly as follows. First, the user chooses "schema-load" option in the command menu, and the system draws a DAG representing the database schema. The user enters schema-change commands by using mouse and pop-up menus to manipulate the DAG. *Transaction Mode* is used to group several schema change operations into a single atomic action. If the user reaches to his desired database schema, he terminates the session by clicking "Quit" in the command menu.

We show several examples of PSYCHO sessions in following sections. These examples sessions of PSYCHO demonstrate the power of graphics to provide a simple method stating schema changes that would be complicated to express in the text-oriented command language of the ORION schema manager.

## 4. Overall Structure of PSYCHO

PSYCHO provides various facilities to help users in posing schema-change commands. In this section, we briefly introduce the structure of PSYCHO system. Figure 3 shows a PSYCHO window which consists of 5 subwindows: a directed acyclic graph window, a PSYCHO lisp window, a command menu window, a class index window and a mouse documentation line window. Besides those subwindows, a method code editor window is provided for editing or modifying code of a method during the schema change session.

### *4.1 Directed Acyclic Graph Window*

The directed acyclic graph window is for the graphical manipulation of the DAG representation of a database schema. The schema of the "University Person" database is shown in Figure 3: the class UNIVERSITY PERSON has two subclasses, UNIV-EMPLOYED and STUDENT, and in turn UNIV-EMPLOYED has two superclasses GRADUATE-STUDENT and UNDERGRADUATE-STUDENT, and so on. We shall use the "University Person" database for illustrating facilities of PSYCHO through sample sessions in section 5.

### *4.2 PSYCHO Lisp Window*

During a schema-change session, the user may have to pose ORION commands or queries directly on the lisp top level. The PSYCHO lisp window is a symbolics lisp window whose size is altered to fit inside PSYCHO system. The user can enter any lisp expressions and any ORION commands in this window.

### *4.3 Command Menu Window*

The command menu consists of 15 commands, which govern schema navigation and mode changes. The 15 commands are solely for PSYCHO, and are not transmitted to ORION.

- *Schema-Load:* PSYCHO reads the database schema definition from ORION into internal data structures, and PSYCHO draws a DAG in accordance with the schema definition.
- *Tutorials:* PSYCHO provides the user with a brief introduction to PSYCHO, some example sessions of schema change, and explains the functions of mouse buttons, pop-up menus and commands.
- *Notification-On:* Various types of system error/warning messages are provided from ORION or PSYCHO. By choosing this command, the user can receive error and warning messages with a beeping sound.
- *Notification-Off:* The user is protected from the error and warning messages.

Including the schema-change mode (default mode), PSYCHO supports 5 different modes. Three of them are implemented and the remaining two are planned to be implemented. A number of pop-up menus are provided in accordance with the modes. The following 4 commands are for changing the modes of PSYCHO.

- *Query-Mode:* This mode is not implemented currently. We are planning to support a graphical query language facility within PSYCHO. We shall discuss this mode in section 6.
- *Transaction-Mode:* In this mode, the user can pose a series of schema change operations as an atomic action (i.e., schema-change transaction). PSYCHO requests transaction service to PSYCHO by sending a message "Begin Transaction" to ORION. In the middle of a transaction, the user can abort. Then ORION undoes the operations performed between abort point and begin-transaction point. If the user commits a transaction, PSYCHO sends a message "End Transaction". We shall discuss this mode in section 5.5.
- *Composite-Object-Mode:* This mode is not implemented. We shall discuss this mode in section 6.
- *Sketch-Mode:* In this mode, PSYCHO does not communicate with ORION. The user can manipulate the database schema experimentally. We shall discuss this mode in section 5.5.
- *Move-Screen-Up/Move-Screen-Down/Move-Screen-Left/Move-Screen-Right:* The DAG window is scrolled up, down, left, or right, respectively. These commands are used for a DAG that is too large to fit on the screen.
- *Reset-Top:* The window is adjusted to show the top of the DAG (i.e. the root appears at the top of the window).
- *Screen-Dump:* Print a hard copy of the screen at a laser printer.
- *Quit:* Terminates a session.

### *4.4 Class Index Window*

The small window located in the left bottom of the screen is the class index window. The class index window shows a list of class names in alphabetical order. The user can scroll the class index window up and down by clicking the *top* and *bottom* labels in the screen respectively. If the user clicks on a particular class name, the system draws the sub-DAG whose root is the chosen class. We shall discuss this window in section 5.2.

### *4.5 Mouse Documentation Line Window*

The mouse documentation line window contains information about what different mouse clicks mean. As the user moves the mouse across different mouse-sensitive items or areas of the screen, the mouse documentation line shows the corresponding documentation to reflect the changing commands available.

### *4.6 Method Code Editor*

In ORION, a method is a CommonLisp form. As such, the method definition process is similar to a lisp function programming session. Therefore the user needs an zmacs[3]-like editor window. In the method code editor, the user can test a new method, modify the method and save the definition of the method into ORION.

## 5. PSYCHO Facilities

In this section we will illustrates PSYCHO facilities by showing example schema-change sessions on the "University Person" database. Before we proceed, we have to clarify the mouse function in PSYCHO. The

[3] Zmacs is Symbolics version of emacs

Symbolics mouse has three buttons. We use only the left and right button, not the middle button. The left button is used for clicking mouse-sensitive items such as nodes in a DAG, commands in the command menu, and class names in the class index window. A mouse sensitive item has its associated action which is supposed to be performed after being clicked by the left button. The right button is used for invoking an pop-up mode which is designed for a particular mode.

The Symbolics supports a system command menu. We retain the system command menu because the user may want to do various things such as inspecting directories, files, etc., during a schema-change session. The system menu can be invoked by clicking the right button twice as shown in Figure 4.

We also intentionally provide some redundancy in the contents of pop-menus and the command menu, i.e., some commands are supported in different menus. That is purely for the user's convenience. Consider the pop-up menu in Figure 5. The pop-up menu is invoked when the user clicks the right button on an area that is not a mouse-sensitive item. The contents of the pop-up menu is exactly same as the contents of the command menu window.

Now we are ready to consider the details of PSYCHO.

## *5.1 Schema Manipulation*

All schema-change operations in the taxonomy in section 2.3 are embedded in the pop-up menu in Figure 6. The pop-up menu (from now on, we call this menu the "basic operation menu") is invoked by clicking the right button upon nodes of DAG which are mouse sensitive items. The first seven items in the basic operation pop-up menu corresponds to (3.1), (3.2), (3.3), (1), (2.1), (2.2), and (2.3) in the taxonomy of ORION schema-change operations, respectively. In this section we will explain the behavior of the seven items by showing examples. The other three items are for schema browsing and user navigation, and are covered in the next section.

Suppose the user clicks the following items on a node of a DAG.

- *Create a New Subclass:* The system will highlight the selected node and display a small box in which the user types a class name as shown in Figure 7. After the user types a class name, the system redraws a DAG including the newly created node as shown in Figure 8.
- *Delete This Class:* The system will drop this class definition and redraws the resulting DAG. Figure 9 shows the resulting DAG after deleting "TENURED-PROF" class. We note that "ASSO-PROF" and "FULL-PROF" now become subclasses of "PROFESSOR" in accordance with the semantics of the ORION schema-change operation (3.2).
- *Rename a class:* The system will highlight the selected node and display a small box in which the user types a new name for the selected node. After the user types a class name, the system redraws a DAG including the renamed node
- *Change the contents of this class:* The system will highlight the selected node and display another pop-up menu showing operation choices for changing the definition of the selected node as shown in Figure 10. Within this pop-up menu, the user can add, delete, or modify instance variables, instance methods, class variables, and class methods.

  *Addition:* When the user clicks the item "add a new instance variable", the template in Figure 11 is displayed. The user fills the slots in the template menu and creates a new instance variable by clicking "Yes" option in the "** Create It! **" item. Class variables are dealt with the same manner. If the user clicks "add a new instance method", a method code editor window is created as shown in Figure 12. The user can create a new instance method, test the new instance method, or modify the new instance method in the method code editor. Class methods are dealt with the same manner.

*Modification:* When the user clicks the item "add a new instance variable", the pop-up menu having the instance variables of the selected class is displayed as shown in Figure 13. The user may choose a particular instance variable in the pop-up menu, say "TACOURSE". Then a template describing the current status of "TACOURSE" pops up as shown in Figure 14. The user can modify the contents of an instance variable by changing the values in slots of the template of Figure 14. Class variables, instance methods, and class methods are dealt with the same way.

*Deletion:* As shown Figure 14, the template having the contents of an instance variable has the "Drop It!" item. By choosing "Yes" option in the template, the user can delete the instance variable from the selected node. However, if the variable is an inherited one, the request is rejected and an warning message will be displayed. Class variables, instance methods, and class methods are dealt with the same way.

- *Make Another Class as a New Superclass of the Class:* The system highlights the selected node and waits for the user to click on another node as shown in Figure 15. The system provides graphical feedback helping the user to choose a valid node. The second selected node will become a new superclass of the first selected node. After the user chooses the second node, the system redraws the DAG to include the new edge. Figure 16 shows the class lattice having the new edge between "HIGHLY-PAID" and "RA-SHIP".
- *Delete SuperSub Class Relationship:* The system highlights the selected node and waits for the user to click on another node as shown in Figure 17. The edge between the first selected node and the second selected node is dropped from the DAG. After the user chooses the second node, the system redraws the DAG to exclude the deleted edge. Figure 18 shows the class lattice after dropping the edge between "HIGHLY-PAID" and "RA-SHIP".
- *Change Superclass Ordering:* The system highlights the selected node and waits for the user to click on two more nodes as shown in Figure 19. The edge between the first selected node and the second selected node and the edge between the first selected node and the third node will be exchanged in the DAG. After the user chooses the third node, the system redraws the DAG resulting from exchanging the two edges. Figure 20 shows the class lattice resulting from edge replacement.

## *5.2 Schema Browsing and User Navigation*

Consider the basic operation menu in Figure 5 again. The bottom three items in the menu are for navigation.

- *Make This Class Top:* The system places the selected node in the top center of the screen and draws all the subclasses of the selected node. The display that results when the user selects "Make This Class Top" on the node "STAFF", is shown in Figure 21. The purpose of this command is to allow the user to concentrate on the selected node and its *subclasses*.
- *Make The First Parent Top:* The system locates the first superclass P of the selected node in the top center of the screen and draws all the subclasses of P. The display that results when the user selects "Make The First Parent Top" on the node "STAFF", is shown in Figure 22. "UNIV-EMPLOYED" is the first superclass of "STAFF".
- *Make This Class Bottom:* The system locates the selected node in the bottom center of the screen and

draws all the superclasses of the selected node. The display that results when the user selects "Make This Class Bottom" on the node "TA-SHIP", is shown in Figure 23. The purpose of this command is to allow the user to concentrate on the selected node and its *superclasses*.

We note that PSYCHO support three types of user navigation facility.

1. The above three commands in the basic operation menu
2. The 5 navigation commands in the command menu window
3. The class index window

In general the commands in category 1 are used when the user wants to reorganize DAG which is already partially visible on the screen. The commands in the category 2 and 3 are useful when the user wants to jump to a class which is located far from the current screen position. If the user knows the exact location of a particular class, he can access the class using the 5 navigation commands in the command menu window. Even if the user does not know the exact location of a class on the screen, he can access the class by searching for the class and clicking it in the class index window. Figure 24 illustrates the display that results when the user clicks the class "GRADUATE-STUDENT" from the class index window. Figures 25-29 demonstrate the scrolling capability of PSYCHO.

## *5.3 Graphical Feedback*

If the system uses lengthy dialogues to interact with the user, an experienced user may feel frustrated. Rather than tedious dialogues, PSYCHO provides several types of graphical feedback.

In Figure 15, the blinking nodes are candidates of new superclasses for the highlighted (selected) node. In Figure 17, the blinking nodes indicate that incoming edges from the highlighted (selected) node can be dropped. Also in Figure 19, the blinking nodes indicate that incoming edges from the highlighted (selected) node can be exchanged. Besides those, if the user tries to create an edge (i.e., IS-A relationship) and the resulting DAG happens to have a cycle, PSYCHO rejects his request with graphical feedback showing the cycle.

## *5.4 Integrity Checking*

The ORION system does not allow schema-changes which violate the invariants of the framework [BKKK87]. In case of unacceptable operations, rather than receiving error messages from ORION directly, PSYCHO explains why such requests are not acceptable. PSYCHO eliminates erroneous schema-change requests by checking the validity of requests before submitting them to ORION.

The result of validity checking is represented in the form of graphical feedback or pop-up messages. For example, as we mentioned earlier, if the user tries to create an edge and the resulting DAG happens to have a cycle, PSYCHO draws a bold cycle to show the cyclicity resulting from the request. Again in Figure 15, if the user clicks on the node "PROCTOR" as a second node, PSYCHO explains that the second node "PROCTOR" is already a superclass of the first selected node "HIGHLY-PAID".

Name conflicts are also checked by PSYCHO. In Figure 6, suppose the user is trying to rename the node "PROCTOR" into "UNIV-EMPLOYED" which already exists, PSYCHO explains the situation of name conflicts. The same is applied to rename operations such as (1.1.3) (1.2.3) (1.3.3) (1.4.3).

## *5.5 Sketch Mode and Transaction Mode*

Besides schema-change mode, PSYCHO supports two other useful modes: *sketch mode* and *transaction mode*. In the schema-change mode, every graphical action is immediately submitted to ORION and ORION performs the corresponding command. However, in the sketch mode, PSYCHO does not communicate with ORION in the middle of a session. The purpose of this mode is improved performance. Since some of schema-change operations are expensive, undoing the previous schema-change causes high system overhead. The problem is even greater when the user undoes several previous changes to a schema. In the sketch mode, the user can freely manipulate a schema through trial and error because schema-changes are virtual. The user can give a command "go-ahead-and-do-it" at the end of the session. This sketch mode is also useful in the stage of initial database design when the user does not have a complete understanding of applications.

The motivation of transaction mode is similar to that of sketch mode. In transaction mode, one schema-change transaction consists of one more schema-change operations. Each graphical action is submitted to the ORION schema manager and to the ORION transaction manager and ORION updates the schema definition with corresponding operations. In case of system failures, the system guarantees recovery. Also the user can abort the schema-change transaction any time in the middle of a transaction.

## 6. Towards An Integrated Database Environment

We are planning to extend PSYCHO to produce an integrated graphical database environment. In this section we describe briefly some of the extensions we are considering.

### *6.1 Graphical Query Interface*

ORION queries are predicate-based lisp expressions and support relational algebra-like operations. Since the ORION model has the notions of composite object and multiple inheritance, ORION queries should be able to express complex predicates to navigate DAG structure of the class lattice and tree structure of composite objects. Hence the ORION query language subsumes the power of existing query languages such as SQL or CODASYL/DML. Since however ORION queries navigate underlying complex structures such as the class lattice, composite objects, etc., the user may find it difficult to pose complicated queries. Consider a query asking instances of hundreds of classes or a query to a composite object having thousands of subcomponents. We believe that a graphical query interface and graphical representation of objects will enhance the friendliness of object-oriented query languages.

### *6.2 Graphical Version Controller*

There is a general consensus that version control is one of the most important functions in application domains, such as integrated CAD/CAM systems and Office Information Systems. Users in such environments often need to generate and experiment with multiple versions of an object, before selecting one that satisfies their requirements. So far we have considered two types of graph structures, class hierarchies and composite object hierarchies. The another graph structure to be considered is "version derivation hierarchy". A unit of version may be a CAD object, a software module, or multimedia document which are all considered as composite objects in the ORION model. It is difficult to support version management in a user-friendly manner. We believe that a graphical representation of versions and the relationships among them can assist users with version management.

### *6.3 Composite Object Browser*

This tool will be embedded in the previous two tools. Restructuring or querying composite objects will be performed graphically in this environment.

## 7. Object-Oriented Implementation

In this section, we present how PSYCHO is implemented. PSYCHO is implemented in an object-oriented fashion using Flavors, which is an object-oriented programming feature of ZetaLisp.

### *7.1 Flavors and ZetaLisp*

The flavor system is the Symbolics machine's mechanism for defining objects. An object can receive messages and act on them. A flavor is a class of active objects, one of which is called an instance of that flavor. A set of instance variables and a set of messages are associated with a flavor. As such, Flavors are similar to ORION classes in several senses. The following example illustrates a sample flavor "automobile", a sample method "old-model-years" which computes the number of years during which a car is used, and instantiation of automobile flavor.

```
(defflavor automobile
            (year-model
            price
            manufacturer
            mass)
            (vehicle)
:gettable-instance-variables
:settable-instance-variables)

(defmethod (automobile :old-model-years) (currentyear)
            (– currentyear year-model))

(setq mycar (make-instance 'automobile
            :year-model 83
            :price 3000
            :manufacturer 'hyundai
            :mass 2500))
```

The Symbolics Flavors system supports hundreds of built-in flavors which are useful in building graphical interfaces. Static menus, dynamic menus, windows, lisp windows, and command menus are all built in flavors.

In PSYCHO, we define a flavor for nodes of DAGs, which is an internal data structure for drawing figures:

```
(defflavor node
            (children
            parents
            name
```

```
                x-coord
                y-coord))
                ()
      :gettable-instance-variable
      :settable-instance-variable
      )
```

The instance variables children and parents have a list of immediate subclasses and superclasses respectively. The instance variable name has the name of a class. (x-coord, y-coord) will represent a position where a node is located on the screen. A number of methods are defined on the flavor node, such as add-child, add-parent, and so on.

Besides the node flavors, PSYCHO has over 10 flavors and associated methods, which are used for managing PSYCHO architecture. Methods are written in ZetaLisp. Since PSYCHO is implemented in an object-oriented fashion, it is fairly easy to modify the functionality of PSYCHO or to extend features of PSYCHO.

### *7.2 Interfacing with ORION Schema Manager*

The ORION schema manger is written in CommonLisp because of the portability issue. Whereas PSYCHO is written in ZetaLisp because graphic functions are only avaliable through Flavors of ZetaLisp in Symbolics. Fortuneately, functions in CommonLisp and functions in ZetaLisp can call each other within Symbolics. By attaching "zl:" to ZetaLisp functions and "cl:" to CommonLisp functions, the lisp interpreter can tell whether a function is borrowed from CommonLisp or ZetaLisp. The concept is called *package* concept. When the user represent schema-changes graphically on PSYCHO, correponding ORION functions should be invoked. When PSYCHO calls ORION function, say delete-class, the corresponding syntax is "orion::delete-class ...". "orion" is the package name.

## 8. Discussion

In this section, we discuss several issues related to PSYCHO. First, we introduce systems similar to that of PSYCHO. Second, we criticize PSYCHO and finally we make some observations on graph representation theory and its relation to graphical schema manipulation.

### *8.1 Related Works*

In recent years, many AI tools have based on object oriented programming. They also provide visual aids for browsing class lattices. PSYCHO is different from the class browsers of AI tools because PSYCHO is designed from a database perspective: transaction mode, sketch mode, query mode, etc. In the visual aids of AI tools, the user can modify the structure of class lattice, but only a few operations are allowed. User navigation on DAG in the visual tools is not as dynamic as PSYCHO. In general the existing AI tools disallow run-time class modification.

Another motivation behind this type of visual tools is that much time is spent learning the large library that is an integral part of most object-oriented language systems. Thousands of built-in classes and methods can not be easily mastered without a powerful visual tool.

**KEE knowledge base browser:** KEE is an AI tool for knowledge engineering, from Intellicorp [INTEL84]. KEE consists of a set of software tools to assist users in building their own knowledge-based systems. KEE supports various programming paradigms including rule-based programming and object-oriented programming. The object-oriented nature of KEE is based on the *Frame* data structure which can have a number of slots and rules and associated procedures. Slots, rules, and procedures all can be inherited or defined locally. Figures 29 and 30 show the KEE knowledge base browser and sample frame structures respectively.

**LOOPS class browser:** LOOPS is a knowledge programming language from Xerox PARC [STEF83]. LOOPS class browser is conceptually similar to KEE knowledge base browser. Figure 31 shows an example of a class hierarchy in the LOOPS class browser.

**GEV:** GEV [NOV82] is a tool which allows the user to display, inspect, and edit structured data written in GLISP [NOV83] which is used for knowledge description. GLISP is a lisp dialect that includes abstract data types. We note that GEV allows changes to actual data as well as changes to data types. DAG representation is not supported in GEV. GEV is initiated by giving it a pointer to an object and its type. GEV interprets the data and displays the contents of data on the screen as shown in Figure 32.

**Flavor Examiner:** This Symbolics utility helps users examine the structure of flavors [SYMB84]. The flavor examiner consists of six panes as shown in Figure 33. The top left pane is the flavor history. The top right pane is the method history. The user enters a flavor name or method name into the bottom pane. The three middle panes are examiner panes that list the answer to a query (i.e., interesting flavors or methods). When the user selects "Edit" in the right end of a pane, the system puts the contents of the pane into a zmacs-like buffer. When the user selects "Lock" in the right end of a pane, the contents in the pane can not be updated. In Figure 33, the user is browsing the system built-in flavor, "tv:essential-window". The definition of "tv:essential-window" is displayed. This tool is rather text-oriented because the user has to type what he wants. DAG representation is not supported in the flavor examiner.

## *8.2 Self Criticism*

We recognize that PSYCHO has several defects. First PSYCHO is not machine independent, but runs only on top of Symbolics because PSYCHO uses many built-in flavors. Second the DAG representation algorithm of PSYCHO is so primitive that sometimes PSYCHO does not use empty space of the screen properly. However, intelligent DAG representation is computationally hard and may benefit some heuristic techniques. Most existing graph browsing systems have trivial layout algorithms [ROWE86]. Third, the user can not see a total map of the DAG, i.e., a map having all classes of a database. A zooming facility would allow the user to view the overall DAG structure, but we have not yet incorporated such a feature into PSYCHO.

## *8.3 Graph Representation Theory*

Our experience from PSYCHO and our previous experience from PICASSO [KKS86], which is a graphical query interface, emphasize the need for further work in *graph representation theory.* In particular *planarity* and *colorability* problems are important in graph representation.

Planar graphs are easier to deal with than non-planar ones. Planar DAGs generally represent a clearer picture of the database than an equivalent non-planar DAGs. Of course, not all DAGs are planar, so we must concern ourselves with minimizing an appropriate measure of non-planarity, such as the number of crossing edges.

Another useful measure is colorability. We note that it is practical to think in terms of color since color displays are becoming increasingly popular. For black-and-while displays, the number of colors would be restricted to the number of gray tones that are conveniently distinguishable. For example, classes (nodes) and relationships among classes (edges) can be represented in a particular color. A graph representation is k-colorable if using k colors, no intersecting edges have the same color and no crossing edges have the same color. We note that the colorability problem is NP-complete. Thus, heuristic solutions need to be investigated.

## 9. Summary

In this paper, we presented a graphical language PSYCHO which is designed as a friendly interface for schema design in the ORION object-oriented database system. It is providing us with the opportunity to evaluate the ORION schema evolution framework. After introducing the ORION data model and system, we presented a framework of schema evolution in object-oriented databases. We provided a detailed description of PSYCHO using numerous sample sessions. Finally we discussed the implementation of PSYCHO and several other related issues.

We summarize the technical merits of PSYCHO below:

- Full support of over 34 schema modification operations.
- Easy browsing and navigation on class lattices.
- Representation of complex schema: Since PSYCHO provides the ability to reorganize class lattices on the screen and to scroll the screen, schemas which are too big to be displayed on the screen can be manipulated.
- Graphical feedback: PSYCHO provides various types of useful visual response during operations.
- Integrity Checking: PSYCHO checks the validity of requested operations by doing associated computations, such as cycle detection, name conflict detection, etc.
- Object-Oriented Implementation: Since PSYCHO was implemented using an object-oriented language, the architecture of PSYCHO is extensible.

**Acknowledgements and Credits:** We would like to thank Won Kim and Jay Banerjee at MCC for providing technical assistance in support of PSYCHO development. We also thank Nat Ballou at AI program at MCC and Wan Lik Lee at the University of Texas at Austin. Wan Lik Lee implemented the first version of schema manager of ORION and Nat Ballou improved the first version and interfaced the improved schema manager and PSYCHO.

## 10. References

[AFSA86] Afsarmanesh, H., Knapp, D., McLeod, D., and Parker, A. "An Object-Oriented Approach to VLSI/CAD," in *Proc. Intl Conf. on Very Large Data Bases*, August 1985, Stockholm, Sweden.

[AHLS84] Ahlsen M., Bjornerstedt, A.,Britts, S.,Hulten, C., and Soderlund, L. "An Architecture for Object Management in OIS," *ACM Trans. on Office Information Systems*, vol. 2, no. 3, July 1984, pp. 173-196.

[BOBR83] Bobrow, D.G.. and Stefik, M. "The LOOPS Manual", *Xerox PARC*, Palo Alto, CA., 1983.

[BAN87] Banerjee, J., et al., "Data Model Issues in Object-Oriented Appplications" *ACM Transactions on Office Information Systems*, March, 1987.

[BKKK86] Banerjee, J., Kim, H.J., Kim, W., and H.F. Korth, "Schema Evolution in Object-oriented Persistent Databases", *Proc. 6th Advanced Database Symposium*, Tokyo, Japan, 1987.

[BKKK87] Banerjee, J., Kim, W., Kim, H.J., and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-oriented Databases", *in Proc. ACM SIGMOD Conf. on the Management of Data*, San Francisco, CA, May 1987.

[CURR84] Curry, G.A. and Ayers, R.M. "Experience with Traits in the Xerox Star Workstation," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 519-527.

[GOLD81] Goldberg, A. "Introducing the Smalltalk-80 System," *Byte* , vol. 6, no. 8, August 1981, pp. 14-26.

[GOLD83] Goldberg, A. and Robson, D. "Smalltalk-80: The Language and its Implementation", Addison-Wesley, Reading, MA 1983.

[FISH87] Fishman, D.H. and et al., "Iris: An Object-Oriented Database Management System" *ACM Transactions on Office Information Systems*, Vol. 5., No. 1, 1987.

[IEEE85] "Database Engineering", *IEEE Computer Society*, vol. 8, no. 4, December 1985 special issue on Object-Oriented Systems (edited by F. Lochovsky).

[INTE84] "The Knowledge Engineering Environment", KEE manuals, *IntelliCorp*, 1984.

[KIM85] Kim, W. "CAD Database Requirements," *MCC Technical Report*, July 1985.

[KKBK86] Kim, H.J., Korth, H., Banerjee, J. and Kim, W. "Property Inheritance Graph: A Formal Model of Multiple Inheritance in Object-oriented Databases," *Unpublished memo*, Dept. of Computer Science, University of Texas at Austin, Texas, Dec. 1986.

[KKS85] Kim, H.J., Korth, H. and Silberschatz, A., "PICASSO: A Graphical Query Language", *TR-85-30*, Dept. of Computer Science, University of Texas at Austin, (also to appear in *Software Practice and Experience*, 1988), Texas, 1985.

[KS86] Korth, H. and Silberschatz, A. "Database System Concepts", *McGraw-Hill Book Company*, 1986.

[NOV82] Novak, G. "The GEV Display Inspector/Editor", *Heuristic Programming Project*, HPP-82-32, Computer Science Department, Stanford University, 1982.

[NOV83] Novak, G. "GLISP: A Lisp-based Programming System with Data Abstraction", *The AI Magazine*, Fall 1983, pp. 37-47.

[MOP85] Maier, D., Otis, A., and Purdy, A. "Object-oriented database development at Servio Logic", *Database Eng.*, Vol. 8, No. 4, 1985.

[ROWE86] ROWE, L., et al., "A Browser for Directed Graphs", *UCB/CSD 86/292*, University of California, Berkeley, April, 1986.

[STEF83] Stefik, M. and et al. "Knowledge Programming in LOOPS" *The AI Magazine*, Fall 1983, pp. 3-13.

[STEF86] Stefik, M. and Bobrow, D.G. "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, Winter 1986, pp. 40-62.

[SYMB84] "FLAV Objects, Message Passing, and Flavors", *Symbolics*, Inc., Cambridge, MA, 1984.

[WOEL86] Woelk, D. Kim, W., and Luther, W. "An Object-Oriented Approach to Multimedia Databases," *in Proc. ACM SIGMOD Conf. on the Management of Data*, Washington D.C., May 1986.

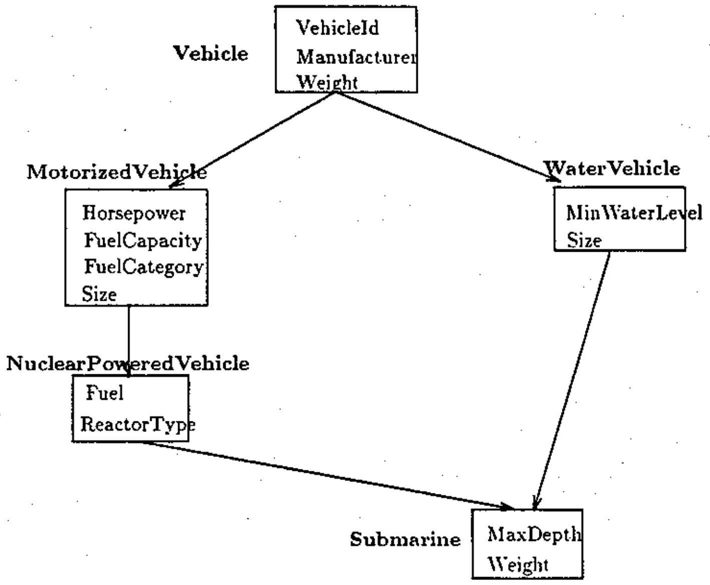


Figure 1: Resolution of name conflicts among instance variables

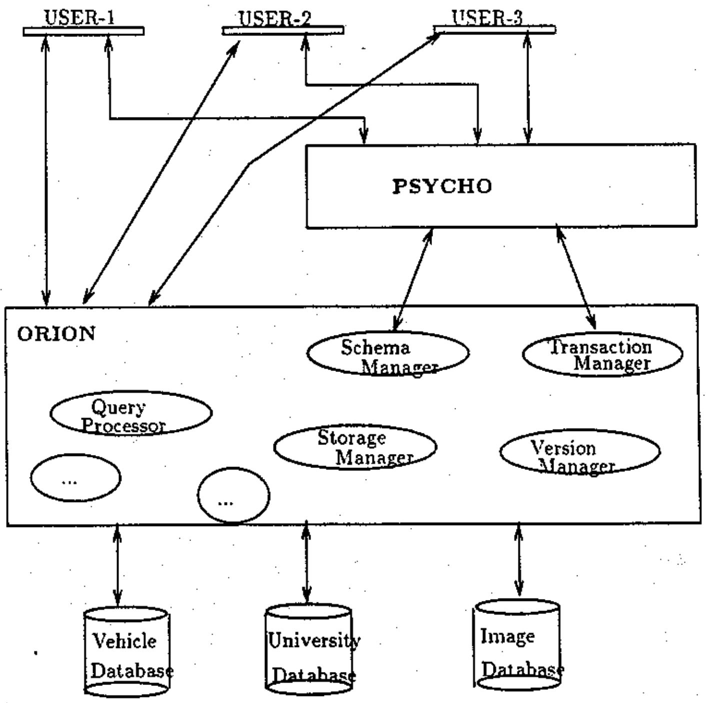


Figure 2: System diagram of PSYCHO/ORION environment

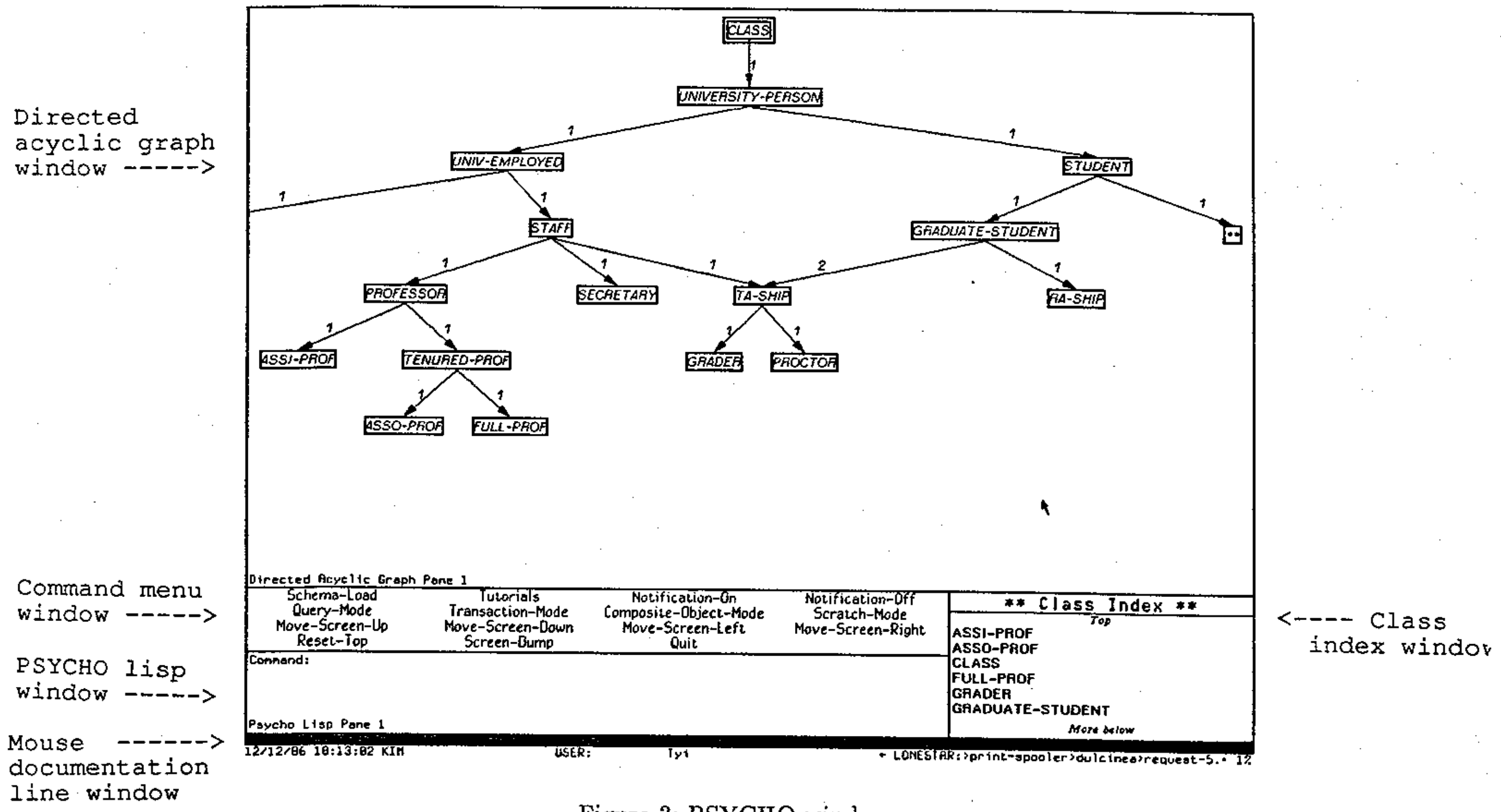


Figure 3: PSYCHO window

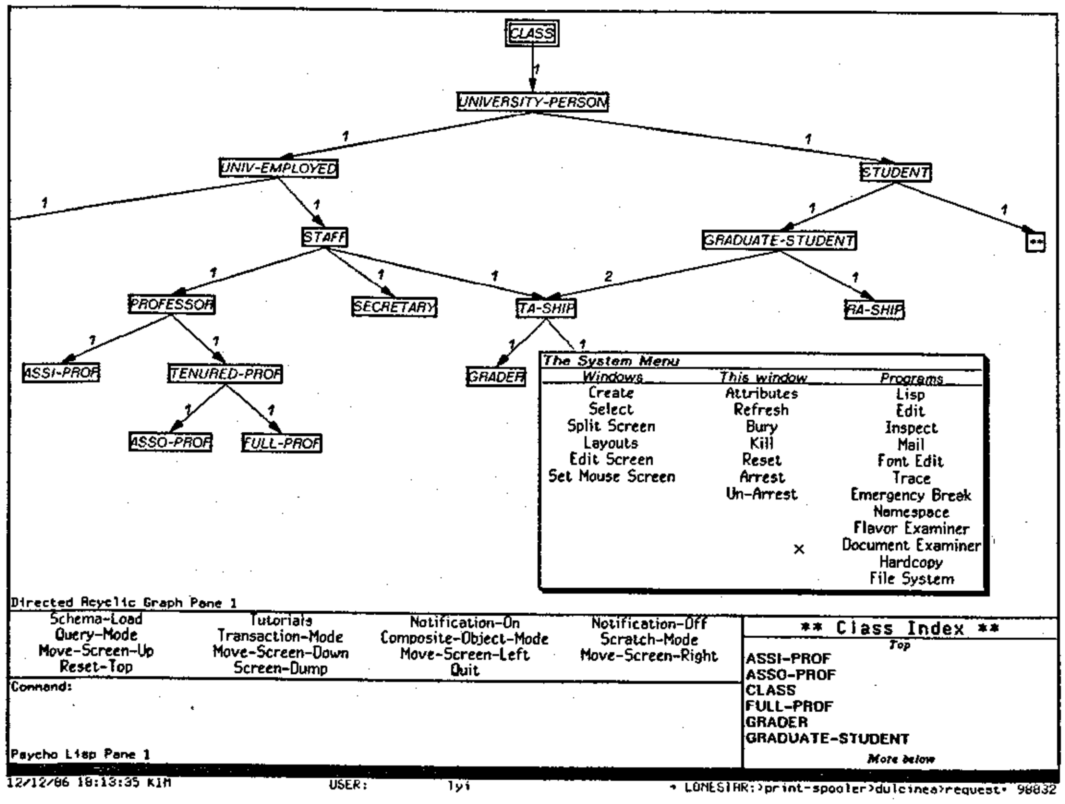
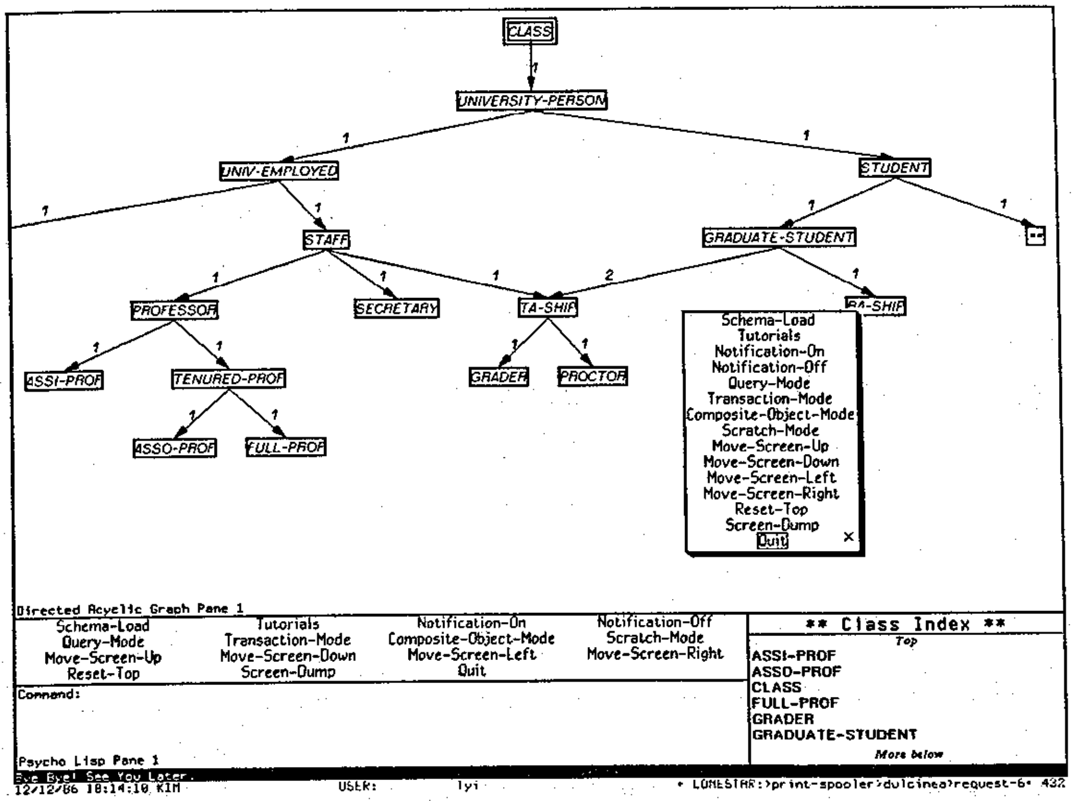
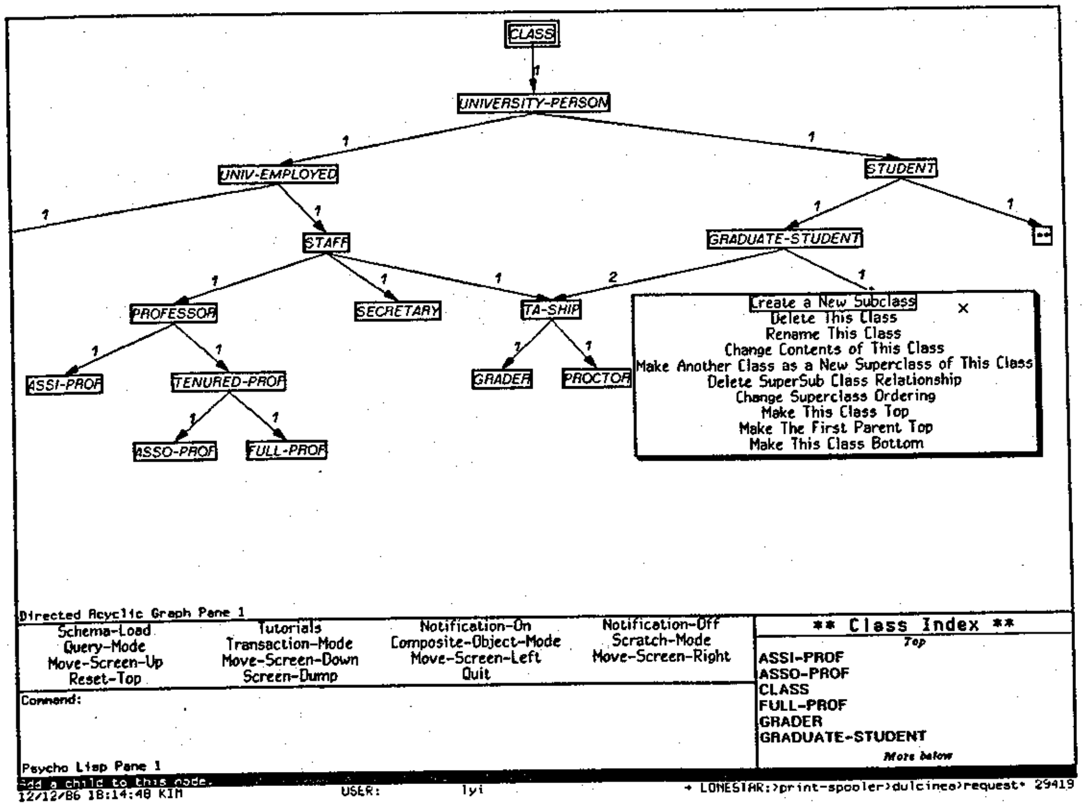


Figure 4: Symolics system menu



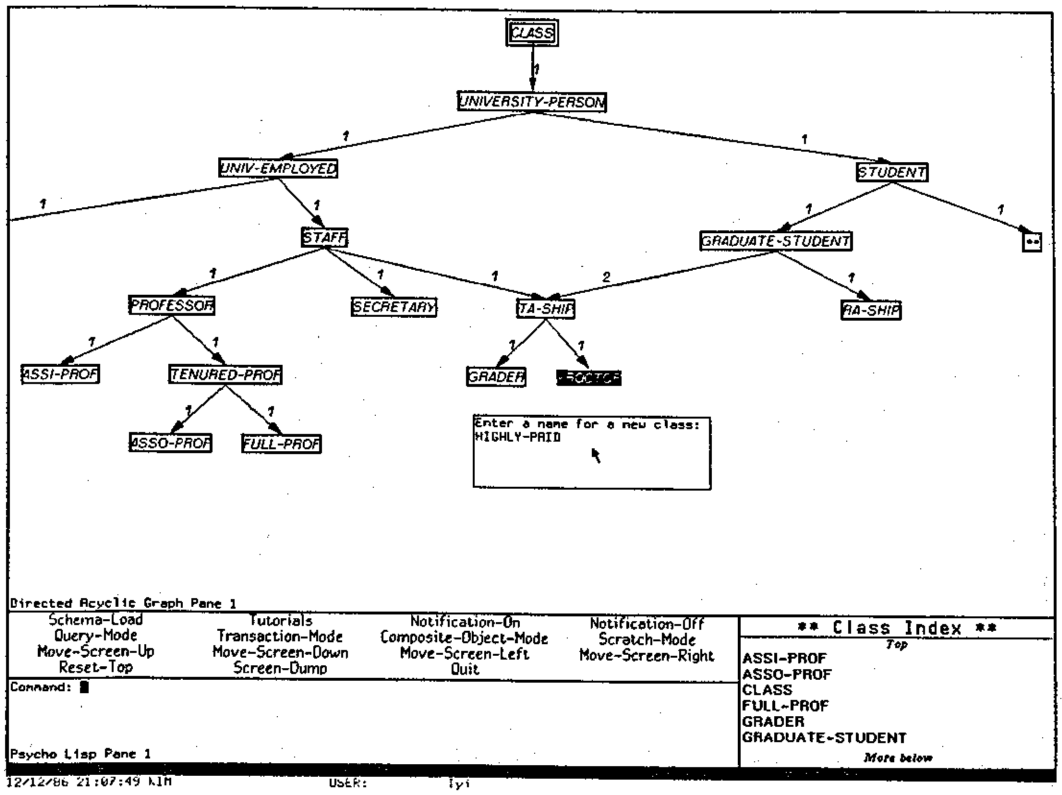
```
Bye Bye! See You Later.
12/12/86 18:14:10 KIM          USER:     lyi          • LONESTAR:>print-spooler>dulcinea>request-6• 432
```

Figure 5: Pop-up menu for non mouse-sensitive items in the DAG window



```
Add a child to this node.
12/12/86 18:14:48 KIM          USER:     lyi          • LONESTAR:>print-spooler>dulcinea>request• 29419
```

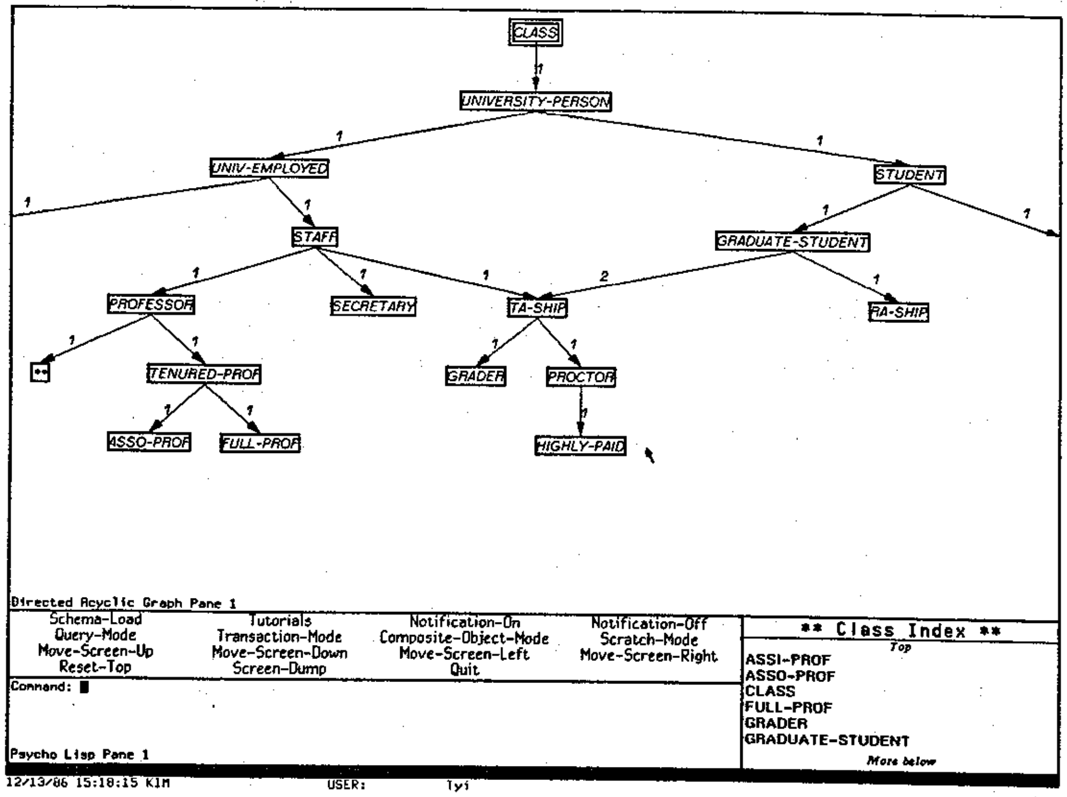
Figure 6: Pop-up menu for mouse-sensitive items in the DAG window



```
Enter a name for a new class:
HIGHLY-PAID
```

```
12/12/86 21:07:49 KIM          USER:          Tyi
```

Figure 7: Pop-up window for providing the name of a new class



```
12/13/86 15:10:15 KIM          USER:          Tyi
```

Figure 8: Class lattice resulting from creating a new class "HIGHLY-PAID"

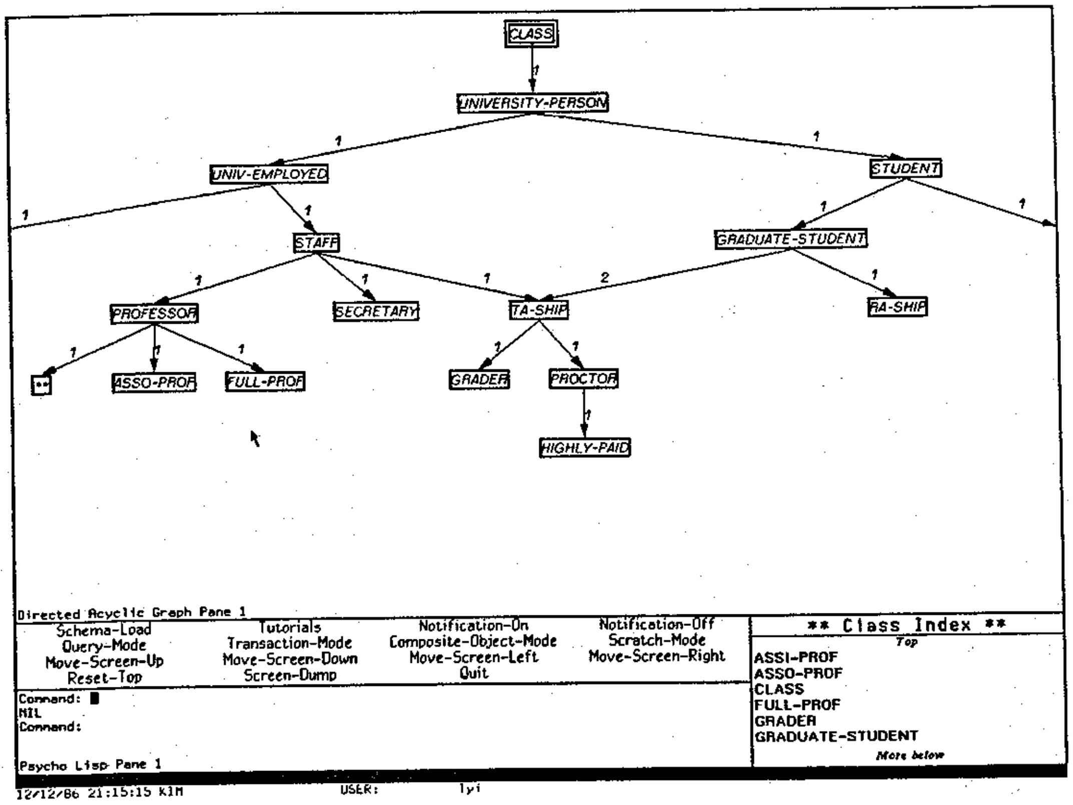


Figure 9: Class lattice resulting from deleting the class "TENURED-PROF"

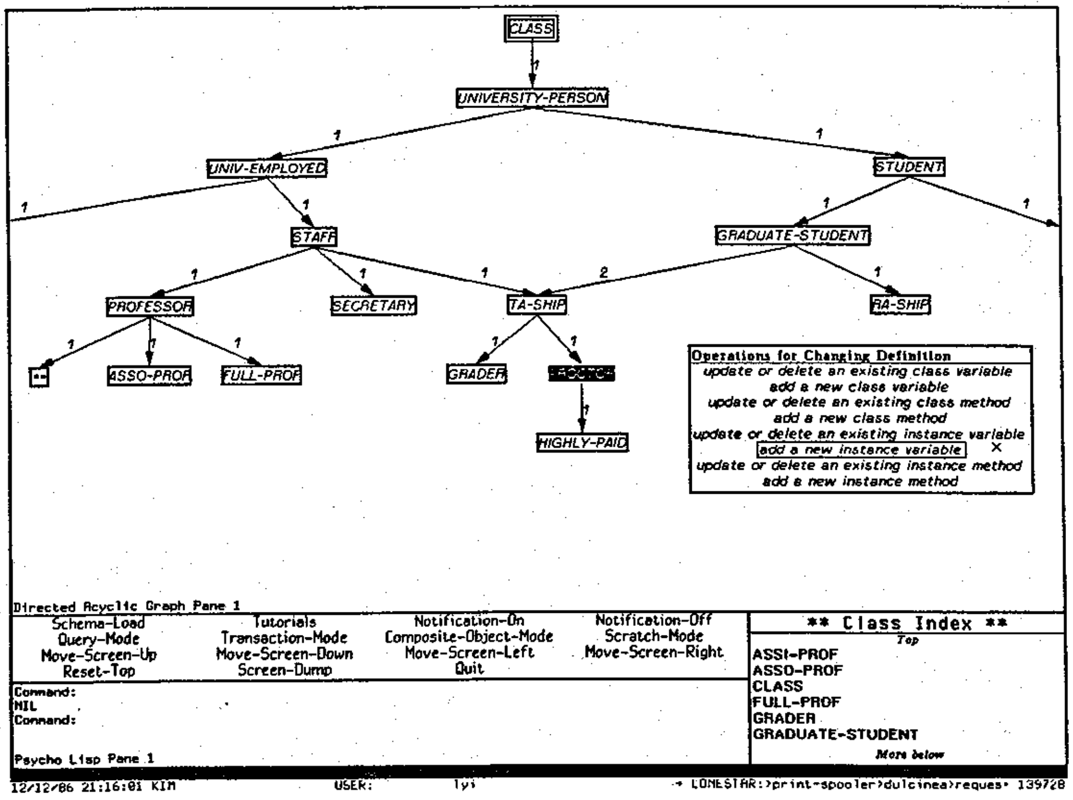
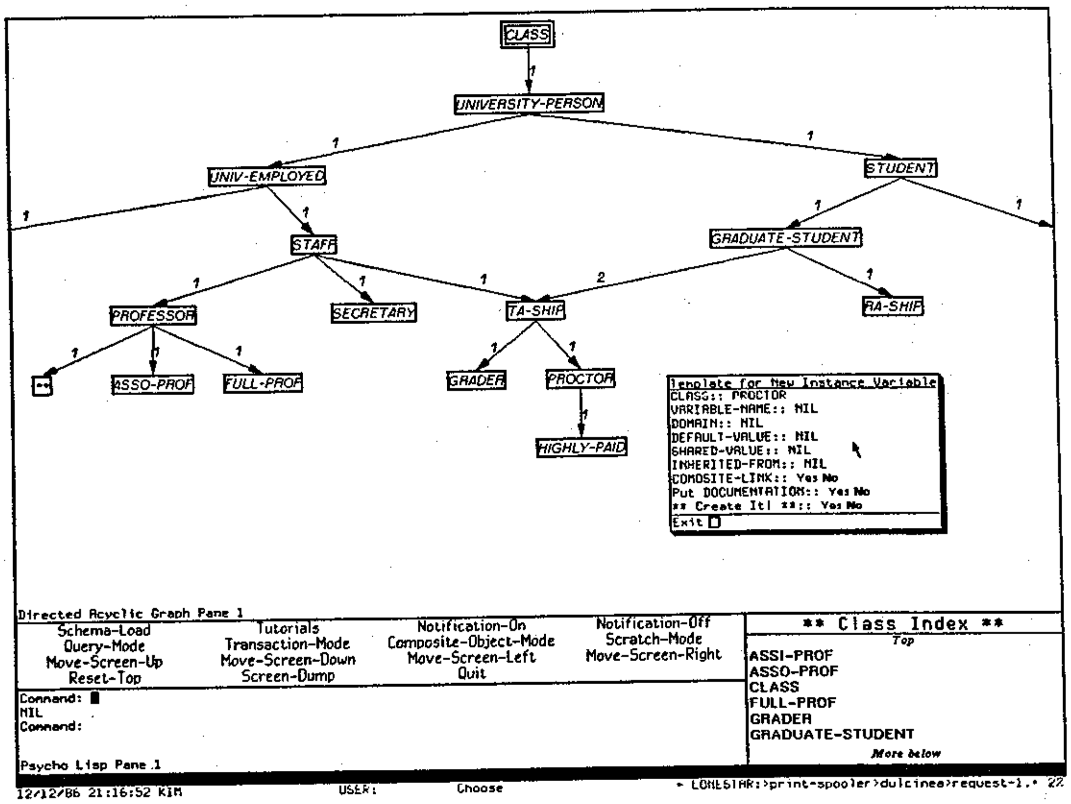


Figure 10: Pop-up menu displaying operations for changing a class definition



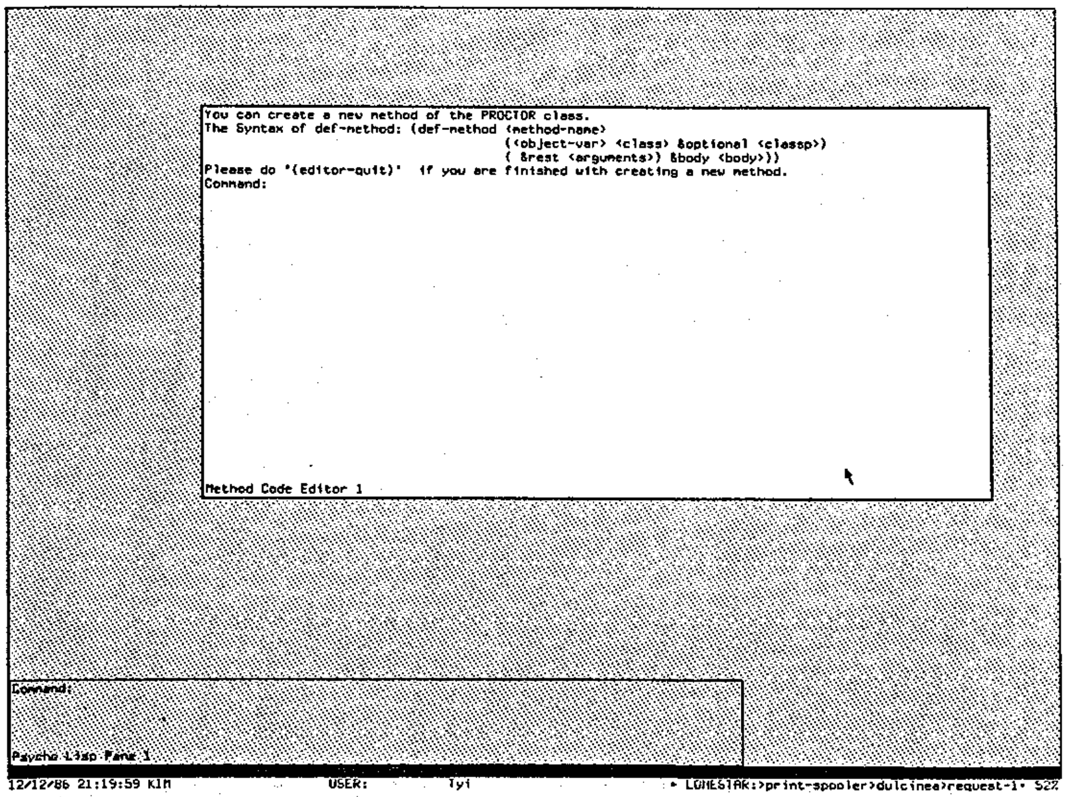
```
Template for New Instance Variable
CLASS:: PROCTOR
VARIABLE-NAME:: NIL
DOMAIN:: NIL
DEFAULT-VALUE:: NIL
SHARED-VALUE:: NIL
INHERITED-FROM:: NIL
COMOSITE-LINK:: Yes No
Put DOCUMENTATION:: Yes No
** Create It! **:: Yes No
Exit
```

```
ASSI-PROF
ASSO-PROF
CLASS
FULL-PROF
GRADER
GRADUATE-STUDENT
```

```
Command:
NIL
Command:
```

```
12/12/86 21:16:52 KIM          USER:          Choose          * LONESTAR:>print-spooler>dulcinea>request-1.* 22
```

Figure 11: Template for a new instance variable

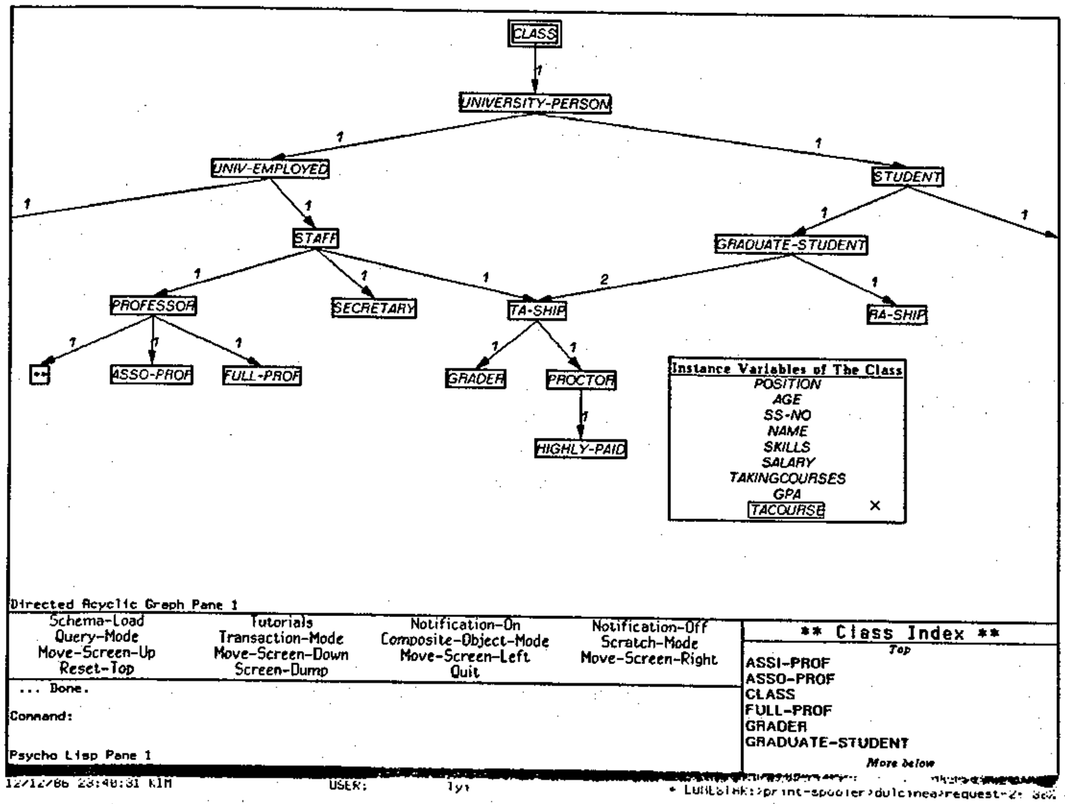


```
You can create a new method of the PROCTOR class.
The Syntax of def-method: (def-method <method-name>
                                      ((<object-var> <class> &optional <classp>)
                                      ( &rest <arguments>) &body <body>))
Please do '(editor-quit)'  if you are finished with creating a new method.
Command:
```

```
Command:
```

```
12/12/86 21:19:59 KIM          USER:          Tyi          * LONESTAR:>print-spooler>dulcinea>request-1* 52%
```

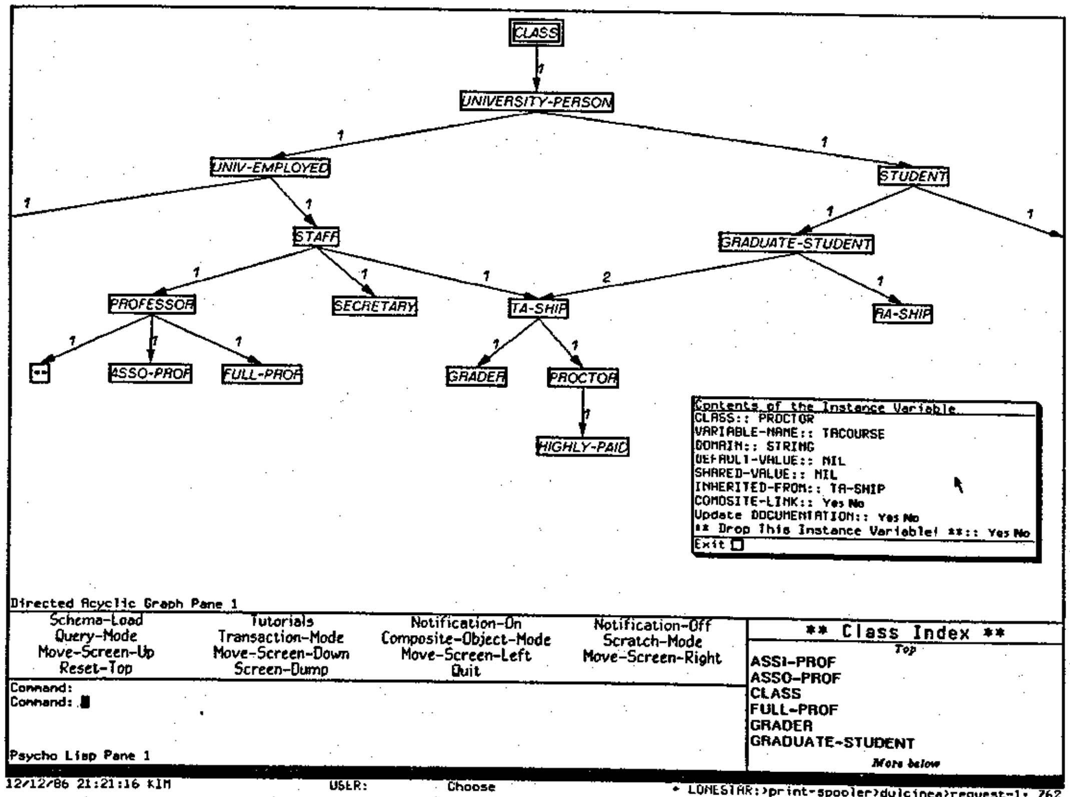
Figure 12: Method code editor



```
... Done.
Command:
```

```
12/12/86 23:48:31 KIM          USER:          lyi          • LONESTAR::print-spooler>dulcinea>request-2• 38%
```

Figure 13: Pop-up menu displaying instance variables of the class, "PROCTOR"

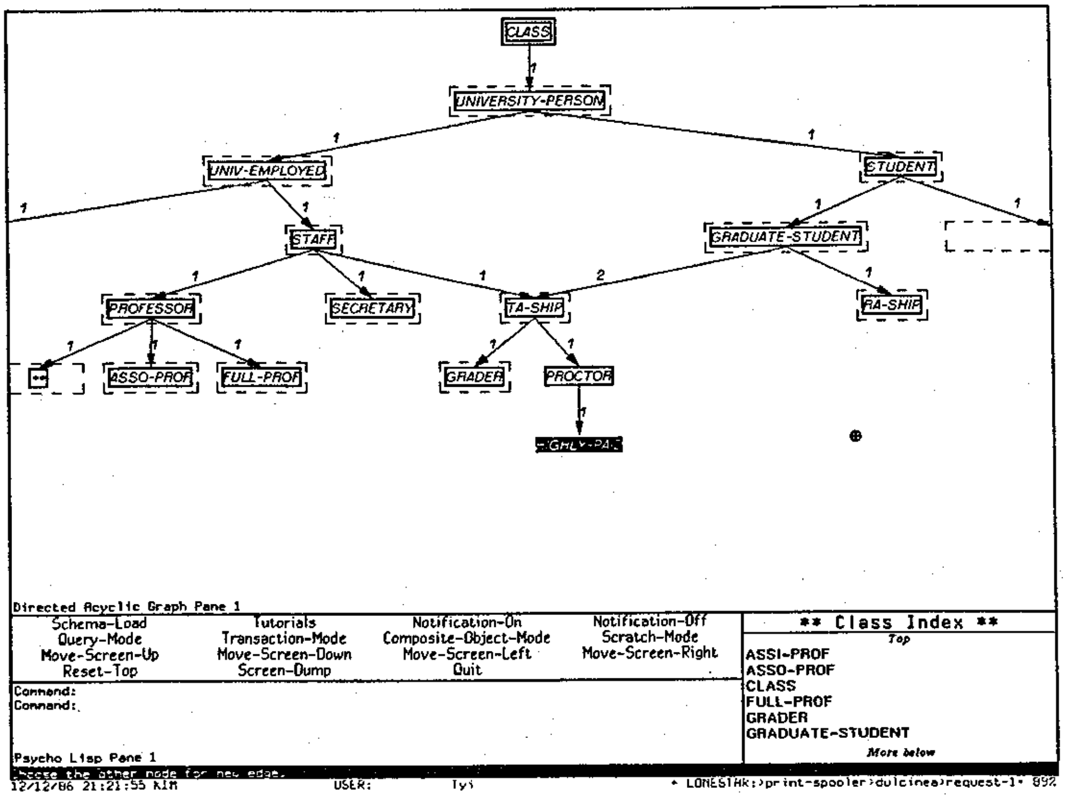


```
Contents of the Instance Variable
CLASS:: PROCTOR
VARIABLE-NAME:: TACOURSE
DOMAIN:: STRING
DEFAULT-VALUE:: NIL
SHARED-VALUE:: NIL
INHERITED-FROM:: TA-SHIP
COMPOSITE-LINK:: Yes No
Update DOCUMENTATION:: Yes No
** Drop This Instance Variable! **:: Yes No
Exit □
```

```
Command:
Command: █
```

```
12/12/86 21:21:16 KIM          USER:          Choose          • LONESTAR::print-spooler>dulcinea>request-1• 76%
```

Figure 14: Template describing the selected instance variable, "TACOURSE"

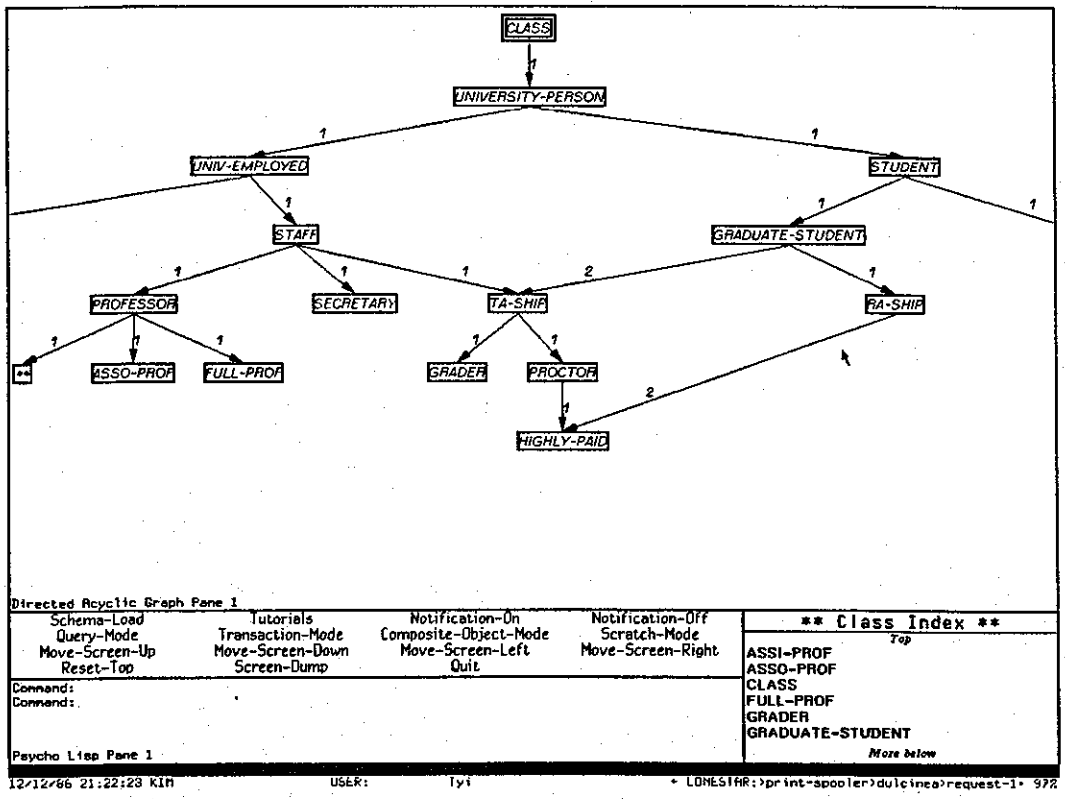


```
Directed Acyclic Graph Pane 1
Command:
Command:

Psycho Lisp Pane 1
Choose the other node for new edge.
12/12/86 21:21:55 KIM          USER:      Tyi          • LONESTAR:>print-spooler>dulcinea>request-1• 89%
```

ASSI-PROF
ASSO-PROF
CLASS
FULL-PROF
GRADER
GRADUATE-STUDENT

Figure 15: Graphical feedback in the middle of adding a new edge

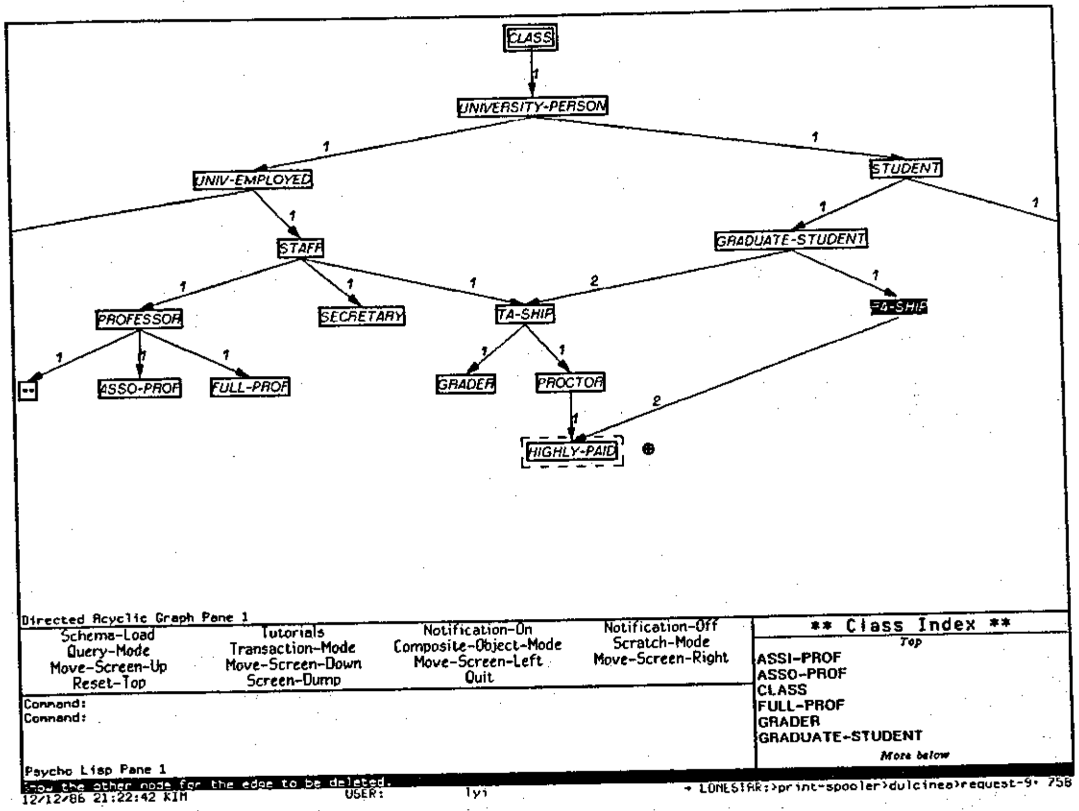


```
Directed Acyclic Graph Pane 1
Command:
Command:

Psycho Lisp Pane 1
12/12/86 21:22:23 KIM          USER:      Tyi          • LONESTAR:>print-spooler>dulcinea>request-1• 97%
```

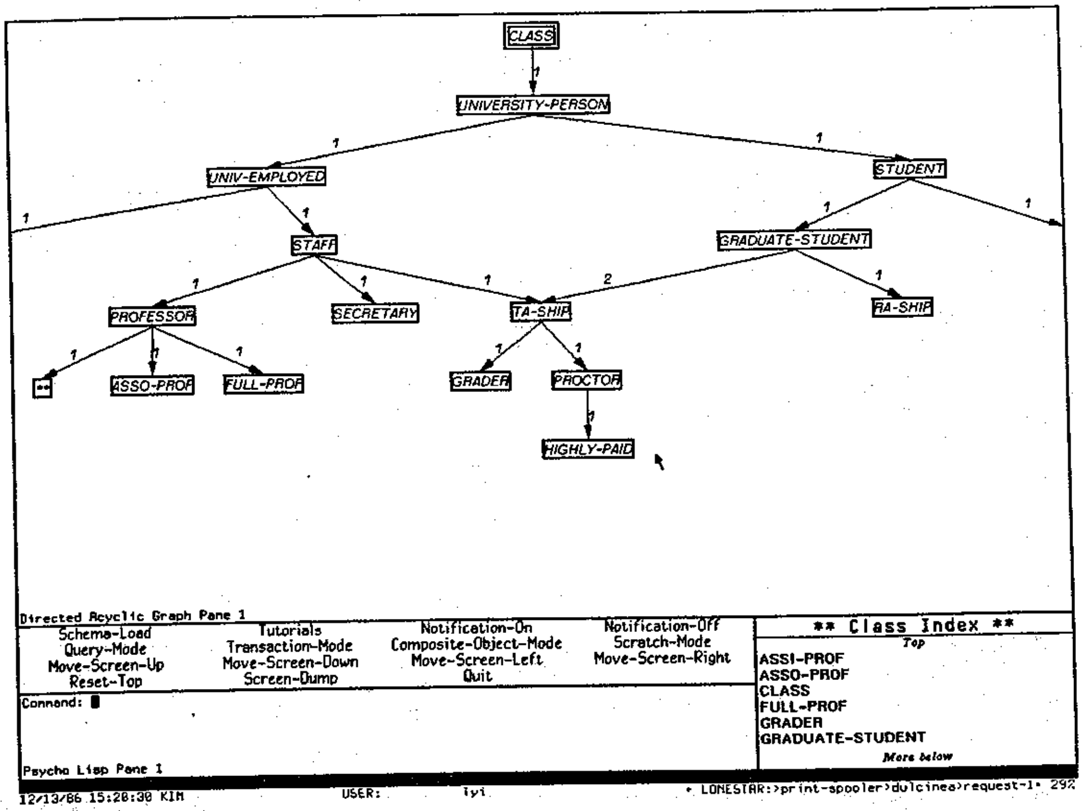
ASSI-PROF
ASSO-PROF
CLASS
FULL-PROF
GRADER
GRADUATE-STUDENT

Figure 16: Class lattice resulting from adding a new edge



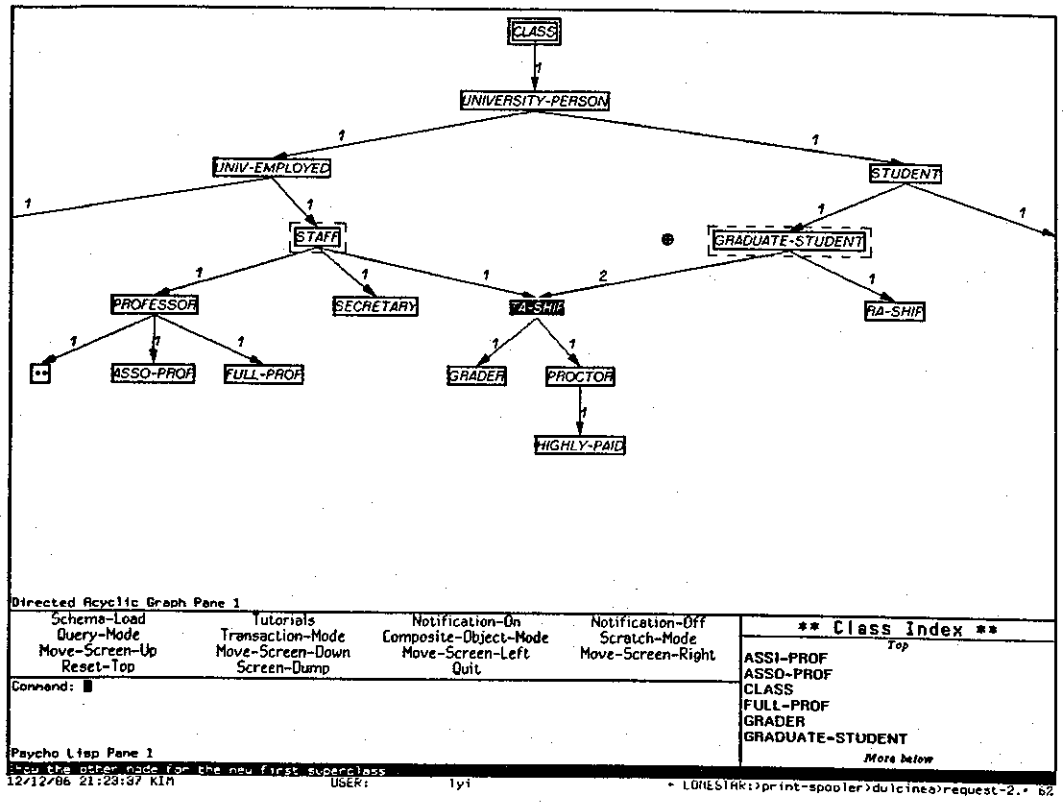
```
Command:
Command:
Show the other node for the edge to be deleted.
12/12/86 21:22:42 KIM        USER:      lyi       → LONESTAR:>print-spooler>dulcinea>request-9• 758
```

Figure 17: Graphical feedback in the middle of dropping an edge



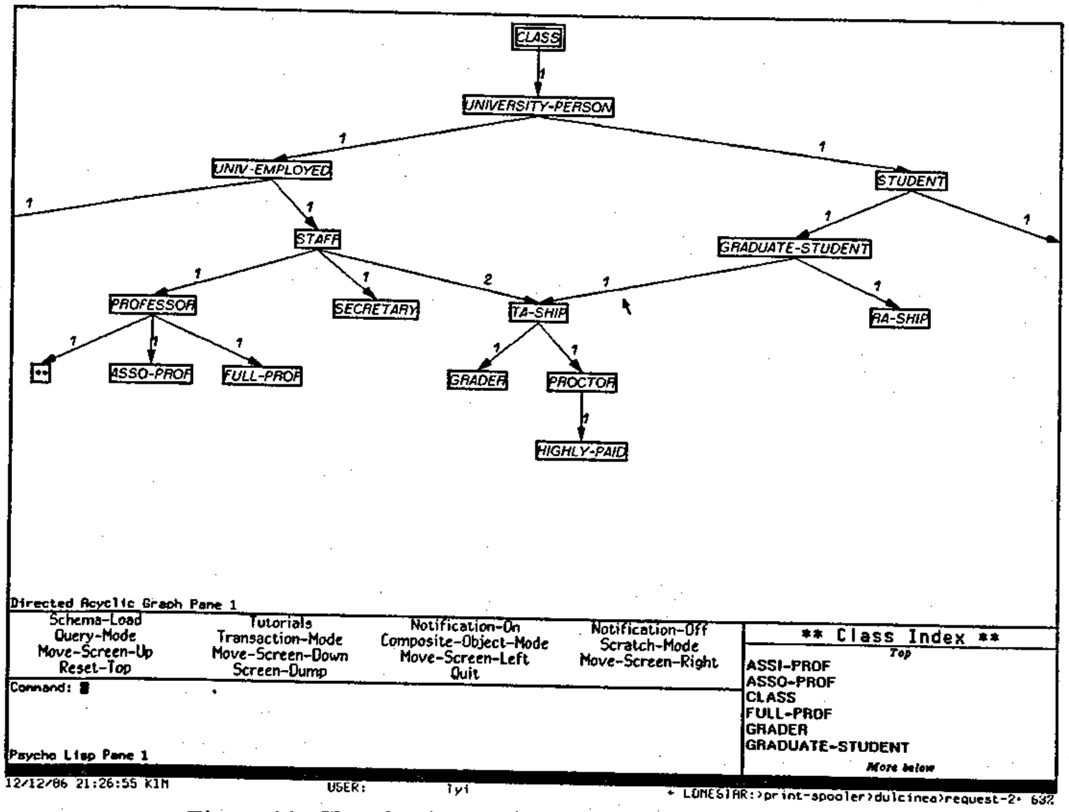
```
Command:
12/13/86 15:28:38 KIM        USER:      lyi       • LONESTAR:>print-spooler>dulcinea>request-1• 292
```

Figure 18: Class lattice resulting from dropping an edge



```
ASSI-PROF
ASSO-PROF
CLASS
FULL-PROF
GRADER
GRADUATE-STUDENT
Command: █
Show the other node for the new first superclass
12/12/86 21:23:37 KIM          USER:     lyi          • LONESTAR:>print-spooler>dulcinea>request-2.• 62
```

Figure 19: Graphical feedback in the middle of exchanging two existing edges



```
ASSI-PROF
ASSO-PROF
CLASS
FULL-PROF
GRADER
GRADUATE-STUDENT
Command: █
12/12/86 21:26:55 KIM          USER:     lyi          • LONESTAR:>print-spooler>dulcinea>request-2• 632
```

Figure 20: Class lattice resulting from exchanging two edges

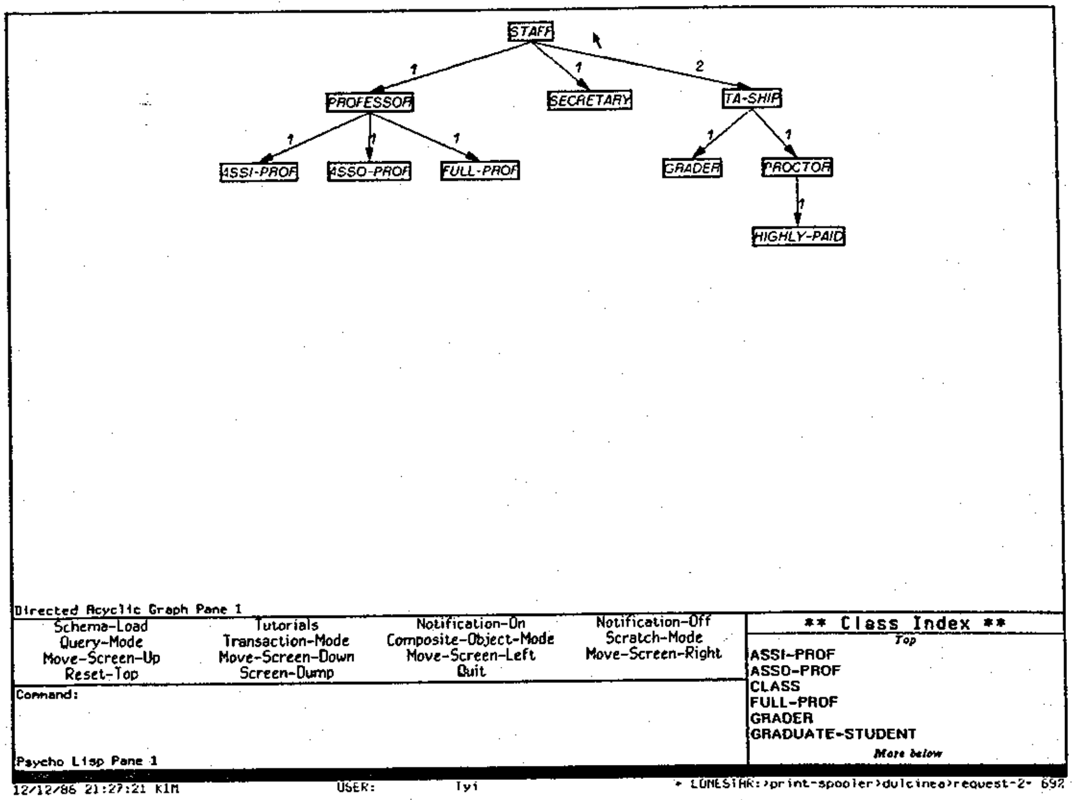


Figure 21: Making the node "STAFF" top

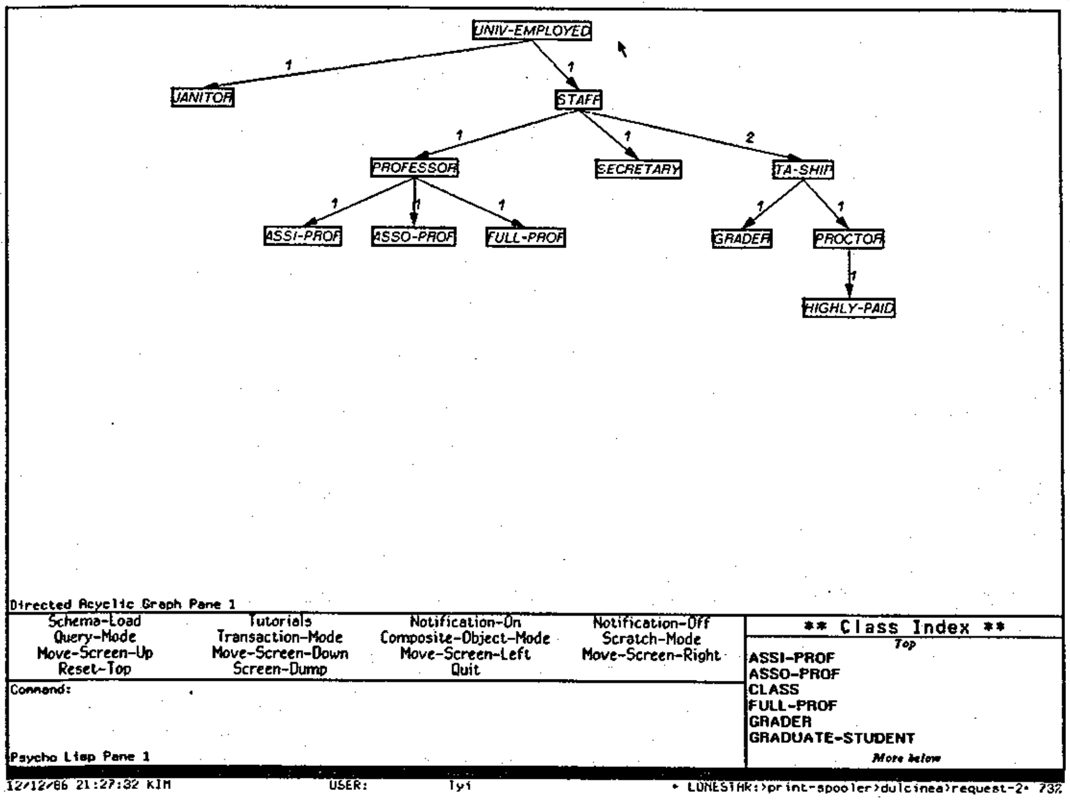


Figure 22: Making the first superclass of the node "STAFF" top

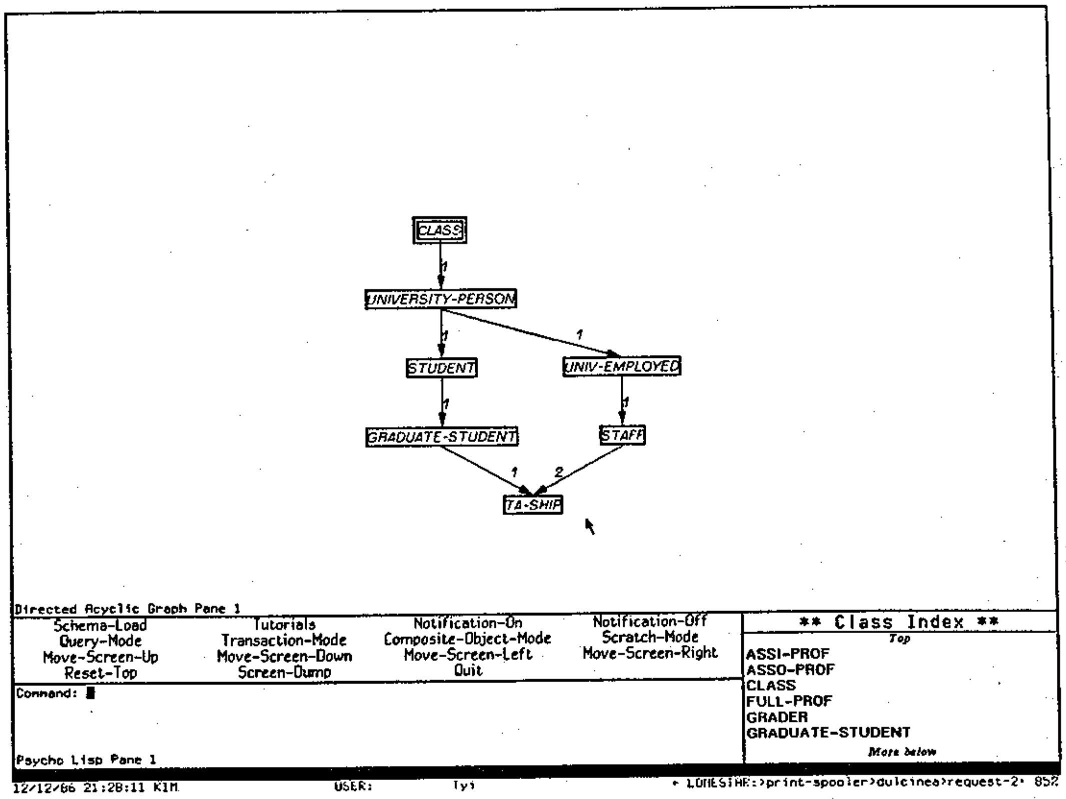


Figure 23: Making the node "TA-SHIP" bottom

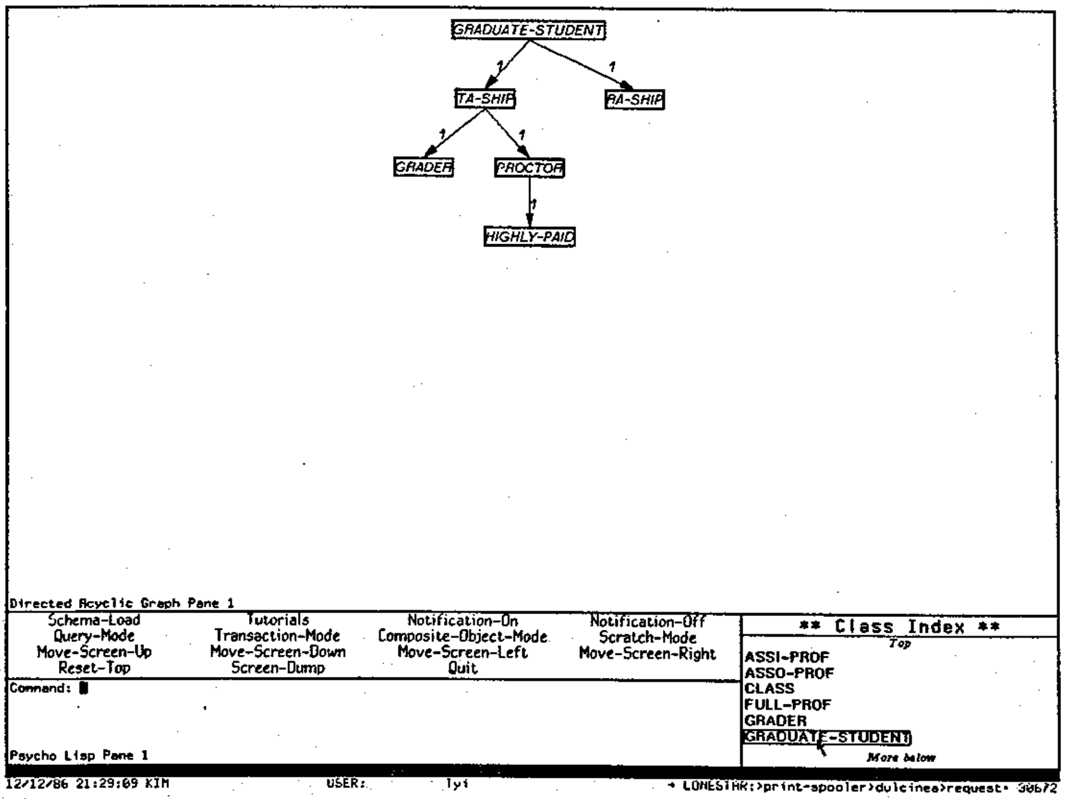


Figure 24: Clicking the class "GRADUATE-STUDENT" from the class index window

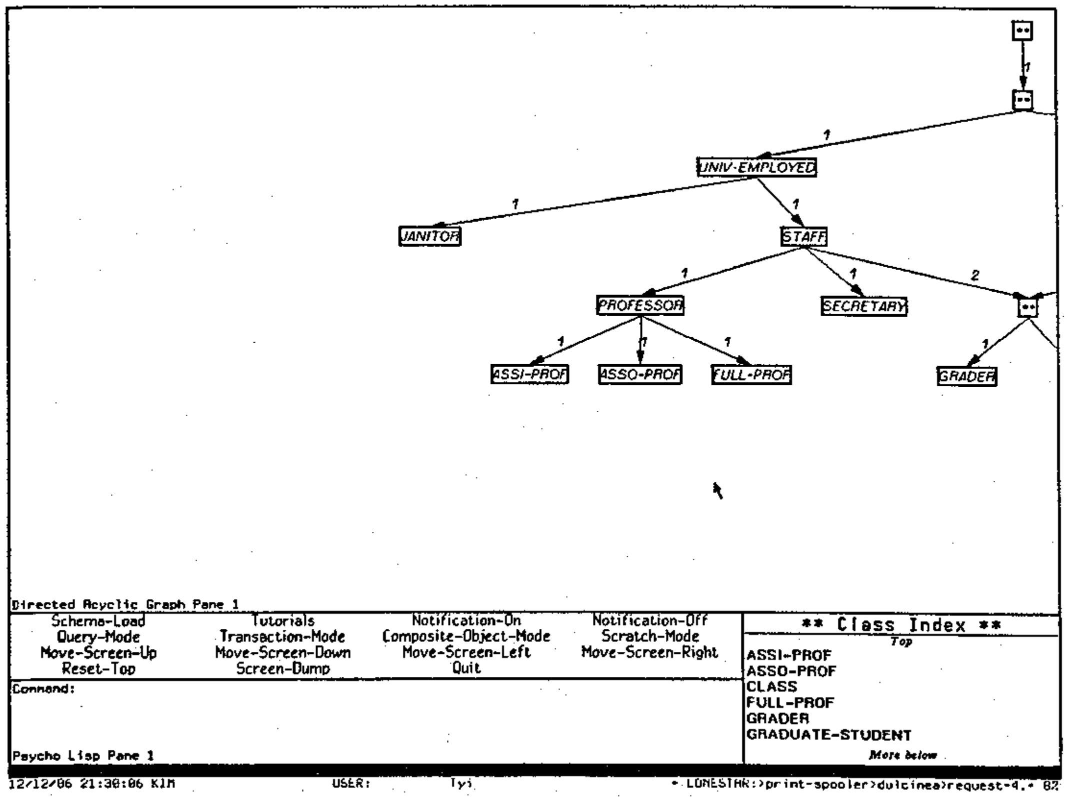


Figure 25: Moving the screen to the left

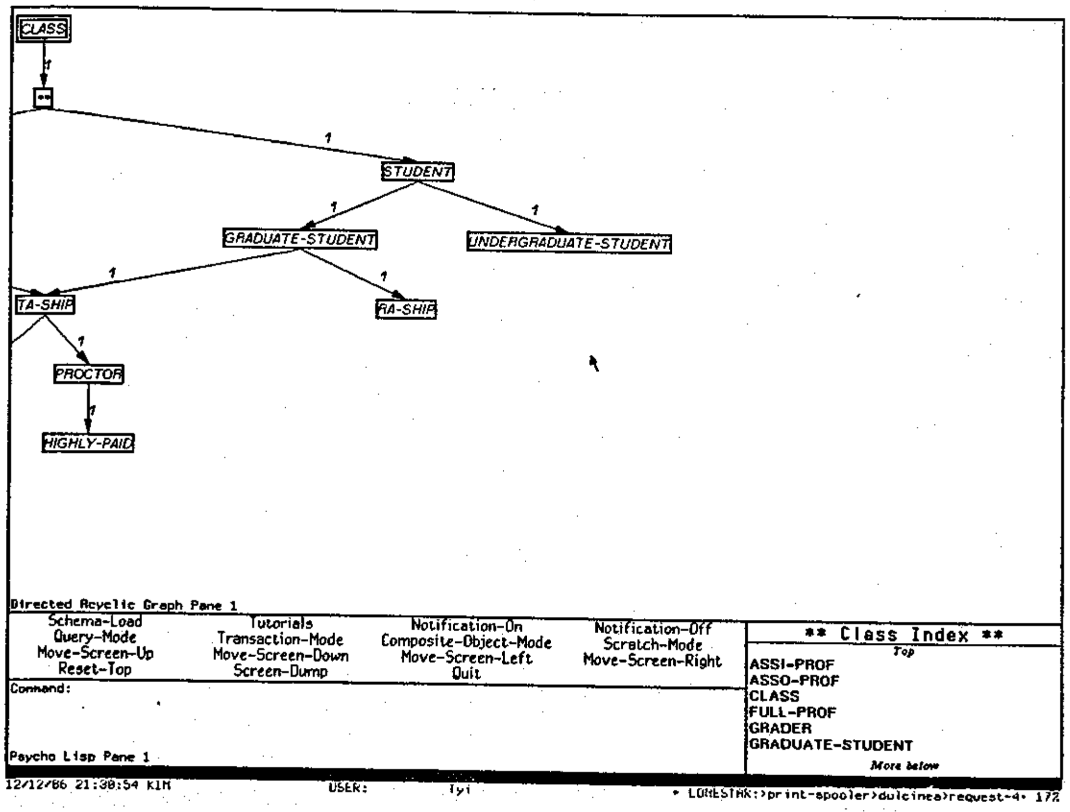
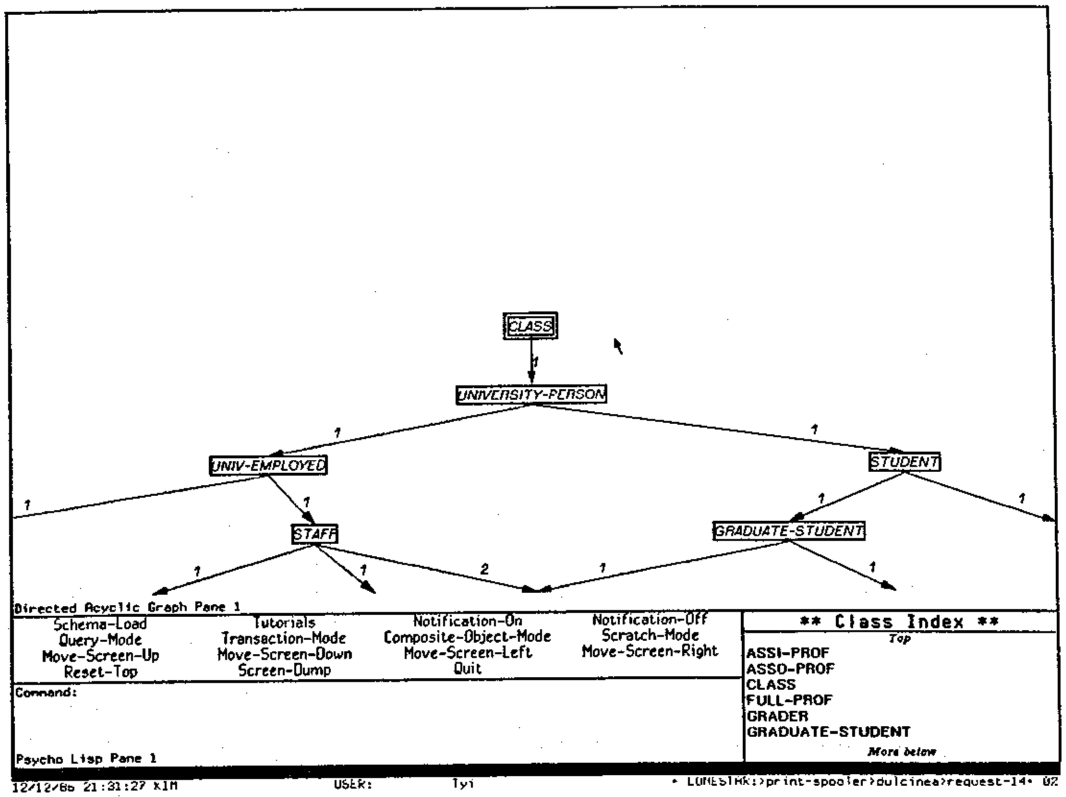


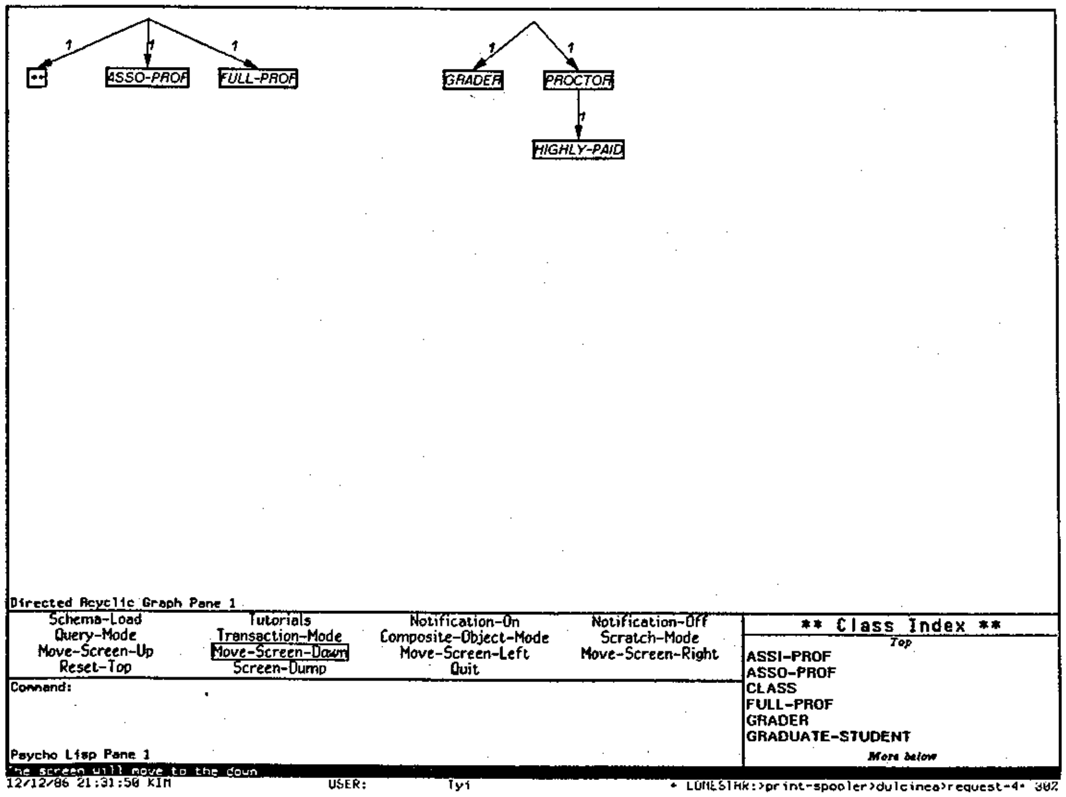
Figure 26: Moving the screen to the right



ASSI-PROF
ASSO-PROF
CLASS
FULL-PROF
GRADER
GRADUATE-STUDENT

```
12/12/86 21:31:27 KIM          USER:        Tyi                    • LONESTAR:>print-spooler>dulcinea>request-14• 0%
```

Figure 27: Moving the screen upward



ASSI-PROF
ASSO-PROF
CLASS
FULL-PROF
GRADER
GRADUATE-STUDENT

```
The screen will move to the down
12/12/86 21:31:50 KIM          USER:        Tyi                    • LONESTAR:>print-spooler>dulcinea>request-4• 30%
```

Figure 28: Moving the screen downward

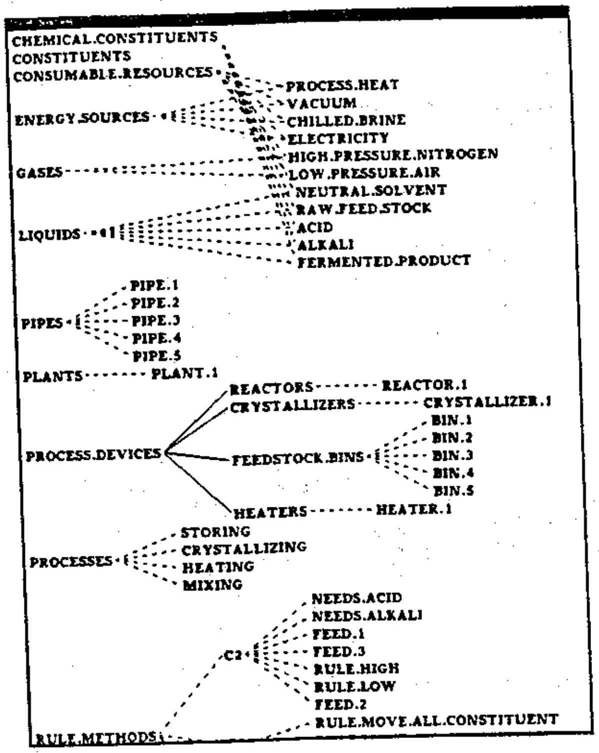
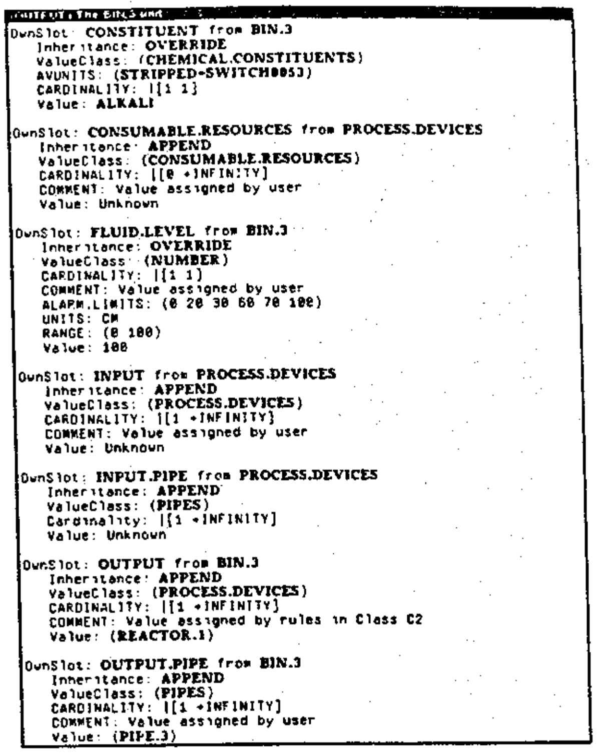


Figure 29: KEE knowledge browser



```
OUTPUT: The BIN.3 unit
OwnSlot: CONSTITUENT from BIN.3
   Inheritance: OVERRIDE
   ValueClass: (CHEMICAL.CONSTITUENTS)
   AVUNITS: (STRIPPED-SWITCH0053)
   CARDINALITY: |[1 1]
   Value: ALKALI

OwnSlot: CONSUMABLE.RESOURCES from PROCESS.DEVICES
   Inheritance: APPEND
   ValueClass: (CONSUMABLE.RESOURCES)
   CARDINALITY: |[0 +INFINITY]
   COMMENT: Value assigned by user
   Value: Unknown

OwnSlot: FLUID.LEVEL from BIN.3
   Inheritance: OVERRIDE
   ValueClass: (NUMBER)
   CARDINALITY: |[1 1]
   COMMENT: Value assigned by user
   ALARM.LIMITS: (0 20 30 60 70 100)
   UNITS: CM
   RANGE: (0 100)
   Value: 100

OwnSlot: INPUT from PROCESS.DEVICES
   Inheritance: APPEND
   ValueClass: (PROCESS.DEVICES)
   CARDINALITY: |[1 +INFINITY]
   COMMENT: Value assigned by user
   Value: Unknown

OwnSlot: INPUT.PIPE from PROCESS.DEVICES
   Inheritance: APPEND
   ValueClass: (PIPES)
   Cardinality: |[1 +INFINITY]
   Value: Unknown

OwnSlot: OUTPUT from BIN.3
   Inheritance: APPEND
   ValueClass: (PROCESS.DEVICES)
   CARDINALITY: |[1 +INFINITY]
   COMMENT: Value assigned by rules in Class C2
   Value: (REACTOR.1)

OwnSlot: OUTPUT.PIPE from BIN.3
   Inheritance: APPEND
   ValueClass: (PIPES)
   CARDINALITY: |[1 +INFINITY]
   COMMENT: Value assigned by user
   Value: (PIPE.3)
```

Figure 30: Sample frame definitions in KEE

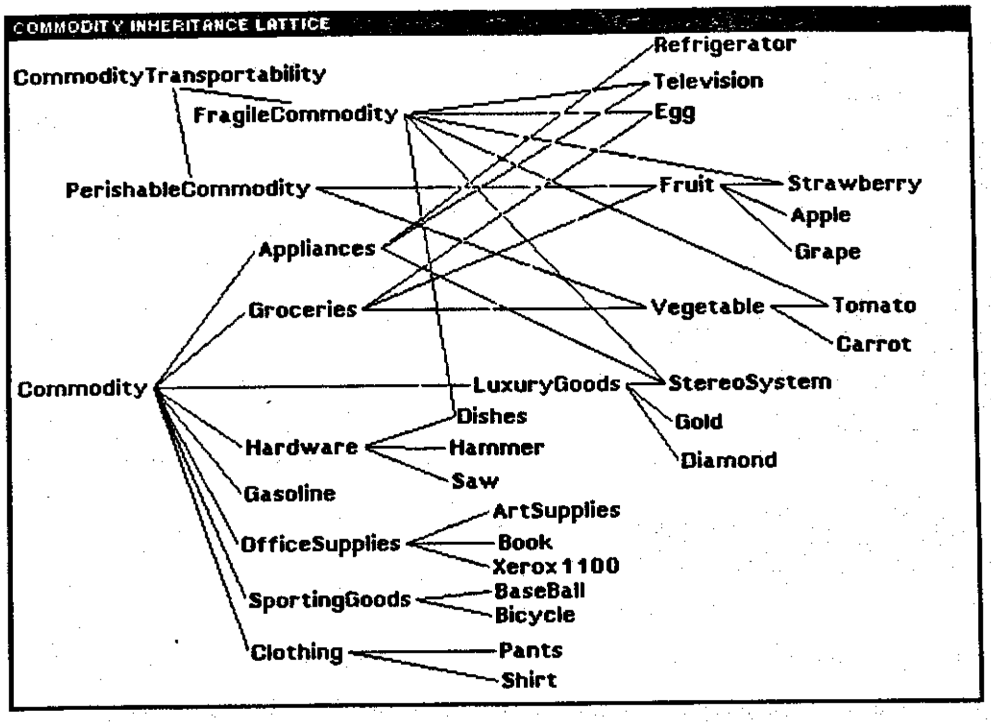
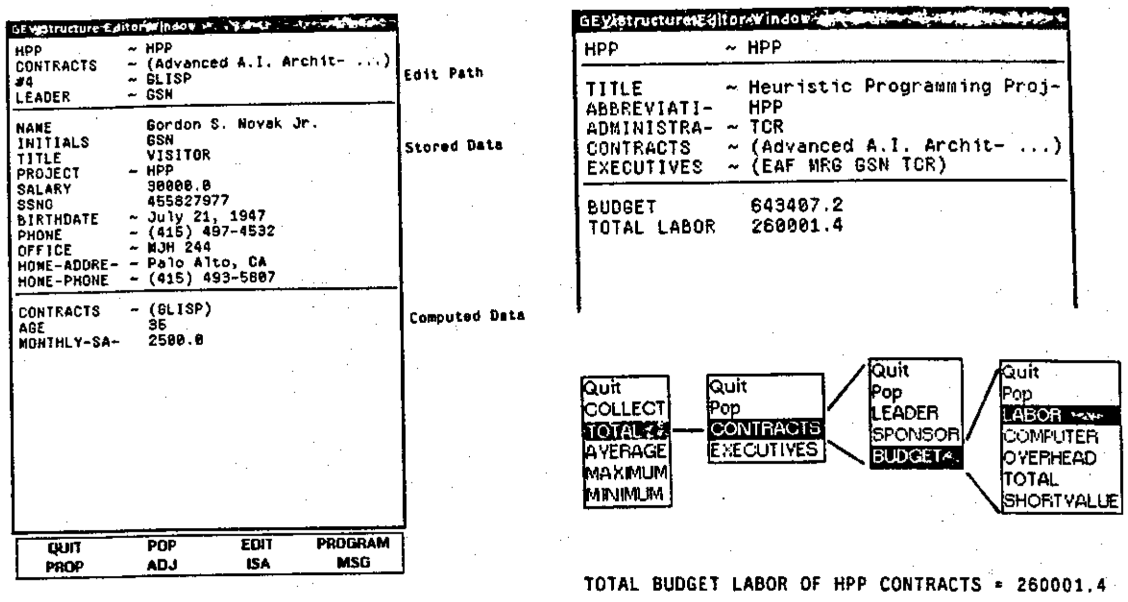


Figure 31: LOOPS class browser



```
GEV Structure Editor Window
HPP            ~ HPP
CONTRACTS      ~ (Advanced A.I. Archit- ...)
#4             ~ GLISP
LEADER         ~ GSN

NAME             Gordon S. Novak Jr.
INITIALS         GSN
TITLE            VISITOR
PROJECT        ~ HPP
SALARY           30000.0
SSNO             455827977
BIRTHDATE      ~ July 21, 1947
PHONE          ~ (415) 497-4532
OFFICE         ~ MJH 244
HOME-ADDRE-    ~ Palo Alto, CA
HOME-PHONE     ~ (415) 493-5807

CONTRACTS      ~ (GLISP)
AGE              35
MONTHLY-SA-      2500.0

QUIT    POP    EDIT    PROGRAM
PROP    ADJ    ISA     MSG
```

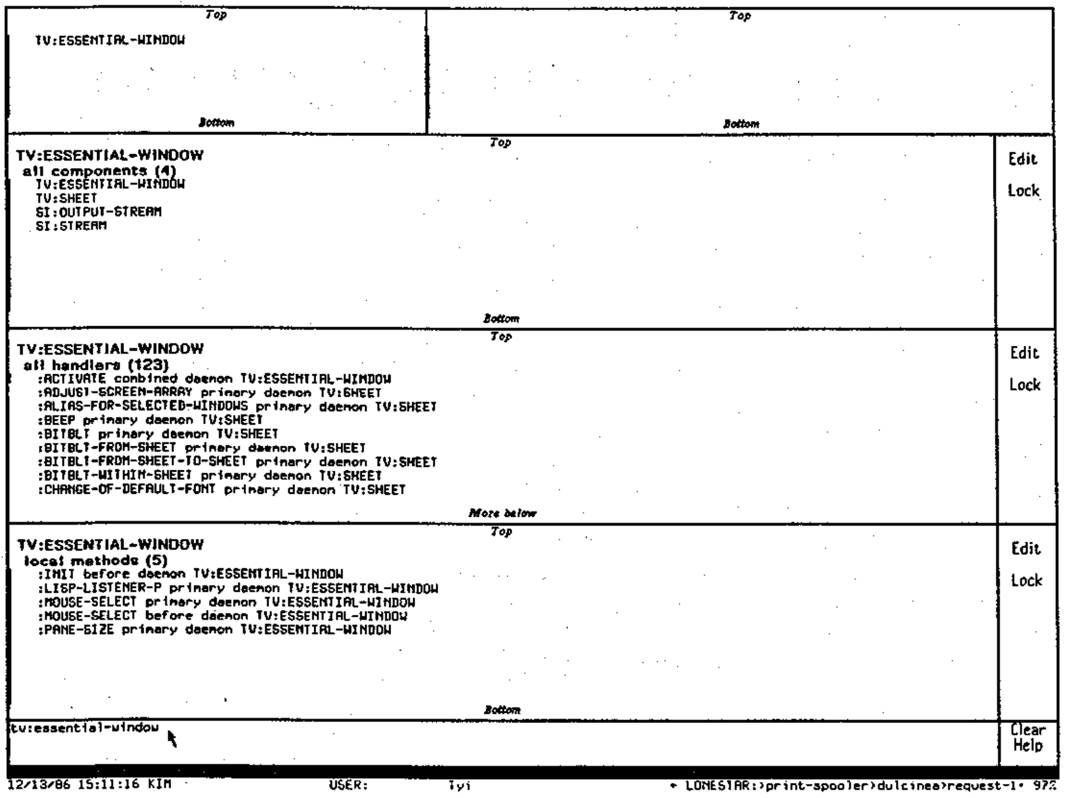
```
GEV Structure Editor Window
HPP           ~ HPP

TITLE         ~ Heuristic Programming Proj-
ABBREVIATI-     HPP
ADMINISTRA-   ~ TCR
CONTRACTS     ~ (Advanced A.I. Archit- ...)
EXECUTIVES    ~ (EAF MRG GSN TCR)

BUDGET          643407.2
TOTAL LABOR     260001.4
```

```
TOTAL BUDGET LABOR OF HPP CONTRACTS = 260001.4
```

Figure 32: GEV



```
                         Top                                              Top
  TV:ESSENTIAL-WINDOW

                        Bottom                                           Bottom
                                       Top
TV:ESSENTIAL-WINDOW                                                              Edit
 all components (4)                                                              Lock
   TV:ESSENTIAL-WINDOW
   TV:SHEET
   SI:OUTPUT-STREAM
   SI:STREAM
                                      Bottom
                                       Top
TV:ESSENTIAL-WINDOW                                                              Edit
 all handlers (123)                                                              Lock
   :ACTIVATE combined daemon TV:ESSENTIAL-WINDOW
   :ADJUST-SCREEN-ARRAY primary daemon TV:SHEET
   :ALIAS-FOR-SELECTED-WINDOWS primary daemon TV:SHEET
   :BEEP primary daemon TV:SHEET
   :BITBLT primary daemon TV:SHEET
   :BITBLT-FROM-SHEET primary daemon TV:SHEET
   :BITBLT-FROM-SHEET-TO-SHEET primary daemon TV:SHEET
   :BITBLT-WITHIN-SHEET primary daemon TV:SHEET
   :CHANGE-OF-DEFAULT-FONT primary daemon TV:SHEET
                                    More below
                                       Top
TV:ESSENTIAL-WINDOW                                                              Edit
 local methods (5)                                                               Lock
   :INIT before daemon TV:ESSENTIAL-WINDOW
   :LISP-LISTENER-P primary daemon TV:ESSENTIAL-WINDOW
   :MOUSE-SELECT primary daemon TV:ESSENTIAL-WINDOW
   :MOUSE-SELECT before daemon TV:ESSENTIAL-WINDOW
   :PANE-SIZE primary daemon TV:ESSENTIAL-WINDOW
                                      Bottom
tv:essential-window                                                              Clear
                                                                                 Help
12/13/86 15:11:16 KIM          USER:         Tyi        • LONESTAR:>print-spooler>dulcinea>request-1• 97%
```

Figure 33: Flavor examiner