INHERITANCE IN A RELATIONAL OBJECT-ORIENTED DATABASE SYSTEM

by

ELLEN MNUSHKIN

March 1992

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirement for the degree of
Master of Science

School of Computer Science
McGill University
Montreal

ABSTRACT

This thesis shows that major object-oriented features can be implemented in a relational database to improve its ease of use. Such object-oriented features as class hierarchy, inheritance of attributes and methods, polymorphism, and collections of classes, are captured by three meta-relations in which the relationships between classes, association of methods with classes, and the composition of collections, are recorded. Each of these meta-relations implies a modification of the relational algebra operators so that they can be applied both to the independent relations and to the relations which are part of the class hierarchy.

We describe the relational algebra implementation of all the modifications to the relational operators necessary to accommodate the object-oriented features mentioned above. New syntax is suggested when these modifications are not sufficient.

RÉSUMÉ

Cette thèse montre que les caractéristiques majeures orientees-objet peuvent etre implementées dans une base de données relationnelle en fin de faciliter son usage. Ces caractéristiques orientées-objet, comme la hiérarchie de classes, l'heritage d'attributs et de méthodes, polymorphisme, et les collections de classes, sont representes par trois meta relations dans lesquelles la relation entre les classes, l'association de méthodes avec les classes, et la composition des collections sont enregistrés. Chacune de ces meta relations implique une modification des opérateurs d'algèbre relationnelle qui peuvent être appliques aux relations indépendantes et aux relations qui font partie d'une hiérarchie de classes.

Nous décrivons l'implantation en algebre relationnelle des modifications aux opérateurs opérationnels nécessaires pour ajouter les caracteristiques orientees objet mentionnées ci-dessus. Une nouvelle syntaxe est suggérée dans les cas où ces modifications ne sont pas suffisantes.

ACKNOWLEDGEMENT

I would like to thank my supervisor Professor T.H. Merrett for his encouragement, technical advice, support and many hours of productive scientific discussions throughout my graduate work at McGill University. He read promptly and carefully earlier drafts of this thesis and his suggestions and criticism contributed greatly to its final form. I am grateful for his guidance and his genuine interest in my research.

Many people have contributed in one way or another to facilitate the work on this thesis. The closely related work of Xian Xiang Hu provided me with the opportunity to test my results, and served as a basis for chapter 6. Heping Shang and Mariza Komioti have been helpful in clarifying a number of topics. I am specially thankful to my husband, Massoud Barzinpour, for his continuous moral support and encouragement. His pride of my achievements and his belief in my ability to do better, have been a strong motivation for me.

Finally, I express my gratitude to professor Merrett for his financial assistance.

TABLE OF CONTENTS

Chapter 1

Introduction

In this thesis we are exploring the possibility of the implementation of the key object oriented features in the relational database model

Before this topic is pursued, it is necessary to have an overview of the main characteristics of relational databases, object oriented programming languages and databases (Sections 1.1 to 1.3 of Chapter 1). Section 1.4, Thesis Aims and Outlines, describes briefly these key object-oriented features. It also presents the considerations involved in the design of these features in the Relix relational database. Several commercially available object oriented programming languages and databases are examined in Section 1.5 with the special emphasis on these features.

1.1 Relational Model

In the relational database model, information is represented in a table format with the following characteristics:

- all rows are distinct
- the ordering of the rows is immaterial
- each column is labelled by a unique name making their order insignificant
- all values in each row and under each column are simple, such as integers, characters and booleans. The meaning of "simple" depends on the operations performed

Each row is called a **tuple** and a column is referred to as a **domain** or an **attribute**. A domain is a pool of values from which the values of an attribute are drawn.

The relational approach to data modelling was proposed by Codd in 1970 [Codd70]. A relation was to be used as a model for a file. A set of relations were to represent data in databases. Thus, in this context, a database is a set of time-varying relations.

One of the great advantages of the relational view of data is data independence. To the user, data appears in the form of tables which can be implemented in many ways, such as incidence matrices, networks, sequential or inverted files, and others. The user need not worry about the physical structure of the relations.

1.1.1 Relational Algebra

The relational approach gives us not only the relational model for data representation but also the **relational algebra** to process data with.

Relational algebra is a collection of high level operations on relations. There is no concept of tuple, each relational operator takes as operands whole relations and returns a new relation as its result. Loops are avoided during the processing of relations. Relations are considered as atomic objects by the relational operations. This simplifies the notation and the manipulation that must be performed and is also very appropriate for the data on secondary storage.

Codd originally defined a set of such operations and showed that those operations were "relationally complete" [Codd72]. Since then the operators of the algebra have been placed in two groups. The first of these groups includes the traditional set operators: union, intersection, difference and cartesian product. The second group is composed of special relational operators: select, project (unary operators) and several families of joins (binary operators). The latter operators are defined as follows.

select - yields a new relation whose tuples satisfy the given selection constraint imposed on the tuples of the given relation,

project - yields a new relation which is defined on a subset of domains of the given relation,

join - combines two given relations in a third relation by joining together tuples with common attributes.

1.2 Object-Oriente ! Programming Languages

Object-oriented languages deal with objects. The common properties (structure and behaviour) of a set of objects can be collected in a class. A class can be seen as a template from which objects are created. The individual objects described by a class are called its **instances**. The instance object has the structure (instance variables) and the behaviour (methods) defined in its class [Weg87]. It is important to keep in mind the distinction between an object and a class. an object is a run-time element which will be created during a system's execution; a class is a purely static description of a set of possible objects - the instances of the class [Meye88a].

Inheritance is a reusability mechanism for sharing properties between objects. The key idea of class inheritance is the provision of a simple and powerful mechanism for defining new classes (**subclasses**) that inherit (re-use) structure and behaviour from the existing classes (**superclasses**), and possibly add some instance variables and methods of their own [Nier89].

Let us consider the two classes, `Employee` and `Student`. Both classes have common structure (for example, `name, address, marital status`), and a common behaviour (for example, they can `change address`, and may `get married`). Additionally, each class has some specific structure and behaviour. An employee can be permanent or temporary, and so forth: therefore we need a method to change his status. For example, a student is registered in a faculty, and should be able to change his program. In an object oriented system we can model this relationship by creating a superclass `Person` which comprises the common properties of `Employee` and `Student`. Then we declare a class `Employee` with its special properties (variables: `status`, etc ; methods: `change work status`, etc.), to be a subclass of `Person` so that `Employee` can inherit all the properties of `Person`. Thus, even though the variable `name` is not defined inside the `Employee` class, it can still be referenced through the inheritance link to the `Person` class. Similarly we declare `Student` to be a subclass of `Person`.

With **simple inheritance**, a subclass may inherit properties of a single superclass. A natural extension to simple inheritance is **multiple inheritance** which allows a subclass to inherit properties of multiple superclasses.

Inheritance is a powerful organizational principle. The **class hierarchy** represents a specialization relationship between a superclass and its subclasses. It also captures similarities and differences among various classes of entities: the similarity between two classes is expressed by *common* ancestors (superclasses) in their class hierarchy, and the difference is indicated by having *different* superclasses. This means that an object-oriented language provides a user interface for the definition and manipulation of relationships among pairs of classes. This in turn means that application programmers need not explicitly program and manage these relationships [Kim91].

Another advantage of the inheritance mechanism is, of course, the possibility of code sharing. Both the programmer and the implementation can take advantage of that. The programmer need not write the inherited methods again, and the program will be shorter. Shorter programs result in a more compact code stored in the computer memory [Yon87].

Methods defined on a superclass are inherited by its subclasses. Furthermore, a subclass may redefine the inherited method by modifying its implementation while retaining the name of the method.

Polymorphism is defined as the ability of operations to be performed on more than one class. Polymorphic methods are methods whose actual parameters can be of more than one class. Cardelli and Wegner [CaWe85] distinguish between two global categories of polymorphism, namely, **universal polymorphism** and **ad-hoc polymorphism**.

Universally polymorphic functions work uniformly on a possibly infinite range of types as long as these types exhibit some common structure. A Count function which counts the elements of a set irrespective of the type of the elements is an example of this kind of polymorphism.

In contrast, **ad-hoc polymorphic methods** work on a finite set of different and potentially unrelated classes (which may not exhibit a common structure). Furthermore, the methods may behave in unrelated ways for each class.

Overloading is a category of ad-hoc polymorphism. Here the same method name is used to denote different methods and the context (class of argument) is used to decide which implementation is denoted by a particular instance of the name. Thus, the same method name can map to different code bodies [UnSc90]. For example, method `Display` may be defined on classes `Circle` and `Point`, but its implementation and behaviour are different in each class.

Class inheritance is closely related to polymorphism. The same operations that apply to instances of a parent class also apply to instances of its subclasses. A method redefined in the subclass is an overloaded method [Nier89]

Encapsulation is the principle which states that a class can only be accessed via its external interface. It strictly distinguishes between the implementation of a class that is only visible to its implementor (and therefore hidden), and its interface, which describes the only way in which users can view the class [UnSc90].

1.3 Object-Oriented Databases

The next-generation database management systems, sometimes called object-oriented DBMSs, are designed to widen the applicability of database technology to new kinds of applications in which traditional DBMSs are claimed to be inadequate These applications include computer-aided software engineering (CASE), mechanical and electrical computer aided design (CAD), computer-aided manufacturing (CAM), scientific and medical applications, graphics representation, office automation, and business applications [Catt91]

The data modelling features of the next-generation DBMSs might be regarded as combining the best features of relational database systems and object-oriented programming languages. Database systems like O_2 [Deux91], ObjectStore [Lamb91], and GemStone [Butt91], which evolved from programming language architecture, provide "database programming languages". That is, they are designed as an extension of object-oriented programming languages (OOPLs) to provide persistence, concurrency control, queries, and other database features for programming language objects Database systems like POSTGRES [Ston91] extend the functionality of relational database systems with procedure calls, efficient object references, hierarchy of objects, and other object oriented programming language capabilities [Catt91].

An object-oriented database (OODB) can represent not only data, relationships and constraints on the data, as can any conventional database, but it also allows encapsulation of data and of programs that operate on the data. Further, it provides a uniform framework for the treatment of arbitrary user-defined data types It is a system which provides database-like support (for persistence, transactions, querying, etc.) for objects, that is, encapsulated data and operations [Kim91].

An object-oriented database supports a set of core object-oriented concepts which are found in most object-oriented programming languages and systems [Kim91].

1. Every object encapsulates a state and a behaviour, where the state of an object is the set of values for the attributes of the object and the behaviour of an object is the set of methods which operate on the state of the object. The state of an object may be accessed, and its behaviour invoked, from outside the object only through explicit message passing.
2. All objects that share the same set of attributes and methods are grouped in a class.
3. All classes are organized in a rooted, directed acyclic graph, or a hierarchy (called a class hierarchy or an inheritance hierarchy). A class inherits all the attributes and methods from its direct and indirect ancestors in the class hierarchy. A message sent to an instance of a class may be bound to a method defined in a superclass of the class.

1.4 Thesis Aims and Outline

This thesis demonstrates how the relational database model can be used to incorporate key object-oriented features, such as inheritance of attributes and methods, polymorphic methods, and collective classes, in the relational model.

The aim of this thesis is to unify relational and object-oriented paradigms 1) to establish exactly which concepts and corresponding syntax are needed to capture the important object-oriented features, and 2) to show that relational and object-oriented approaches are not incompatible. The design of the following features is addressed:

- ***Classes*** are identified with relations;
- ***Object identity*** (Id) is given by ordinary attributes, which may or may not be keys of the relations in which Id's are defined;
- ***Class hierarchies*** are represented as relations; in particular:
  - an ***ISA hierarchy*** is introduced for the inheritance of
    - attributes, and
    - methods;
  - a ***HASA hierarchy*** is introduced to provide subobjects and the broadcasting of methods to all subobjects.

- ***Methods*** are identified with constant, function, or procedure-valued attributes of a relation.
- A high level relational language such as Relix has no concept of "tuple" or corresponding syntax (it deals only with "relations", as is suitable for a language geared to collective or "bulk" data on secondary storage). Therefore, the object-oriented approach incorporated in Relix will have no concept of "object" or corresponding syntax, but only the concept of "class". It can thus be thought of as a "class-oriented" relational language.

There are two possible approaches to achieve the aims specified above.

1. Introduce minimal new syntax into Relix to express object-oriented features such as inheritance;
2. Implement everything as procedure calls in Relix with no new syntax (except that presently Relix does not have procedures or functions, so we will propose how a procedure call is made).

When a new syntax is necessary, we will show that it can be implemented in the Relix database language itself without reverting to the base language (C) in which Relix is implemented.

Object Identity.

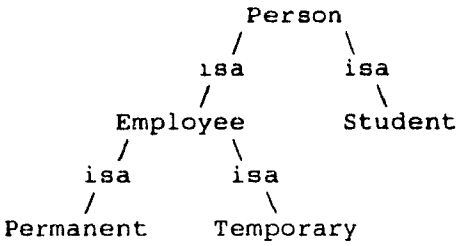
A key concept of object-oriented programming is that each object has a unique identity by which it can be referenced. In most of the commercially-available OODBs the object's identity is its physical address,and objects refer to each other by a pointer In a relational database the concept of a pointer does not exist. In order to be consistent with the philosophy of the relational approach, we choose to represent the object identity by an attribute of a relation.

In Relix all the attributes of a relation can be manipulated at the programmer's discretion. We will consider whether it is necessary to have a somewhat restricted access to the Id field in order to maintain the system integrity.

Class Hierarchy.

The relational model does not support the concept of inheritance hierarchy. For example, the relationship between the relations `Person, Employee, Permanent, Temporary`

and Student is described graphically below. They form an ***ISA hierarchy***.




```
Person (id, Name)  Employee (id, Emp#)  Permanent (id, Salary)
         1  Mike              1  123                 1    30
         2  Ann               2  345                 2    25
         3  Pete              3  456
         4  Kate

Temporary (id, Hours)     Student (id, Stud#)
            3   10                  4  91012A
```

Knowing the relationship between these relations, we can put together the information in these tables to conclude that Ann is a permanent employee whith salary of 25.

Before we talk about using a class hierarchy, we need to find a way of recording the links between classes. We need to adopt a new syntax for arranging classes into a hierarchy.

ISA Hierarchy.

Once the inheritance mechanism is in place, the features that are described in the superclass relation may be referred to by the subclass relations. The following expression includes a selection and a projection of inherited attributes from a subclass relation Permanent

[Name, Salary] **where** Emp# = 123 in Permanent

The inherited attributes of a subclass relation can be updated. For example, we might want to change the name of a Permanent Employee whose salary is 25.

update Permanent **change** Name ← "Anne" **using (where** Salary = 25 **in** Permanent)

HASA Hierarchy.

Suppose that an international organization wants to pass a new international law. The committee members represent different countries. To pass a new law, each committee member must first get his country to accept it. The countries are broken down into two categories by their distribution of power: democracies and dictatorships. The procedure by

which a new law is accepted is different in each category of country.

We want to send a message "Pass new law" to all countries represented on the committee. The Continent inheritance hierarchy is shown in Figure 1.1 below.

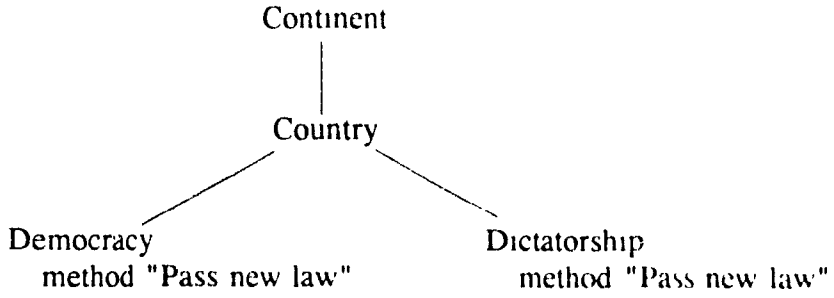


Figure 1.1 Continent inheritance hierarchy.

The inheritance hierarchy does not provide the capability of sending a message to a group of related objects that do not necessarily belong to the same class. It is obvious that the inheritance hierarchy cannot satisfy all the needs of the user

To provide the capacity to delegate messages downward, we need a notion of collective classes and a collection hierarchy

In our example, a ***collective class*** Committee will keep a list of the countries represented on the committee. The ***collection hierarchy*** specifies the classes of objects found in the collective class In the Figure 1.2 below, the Committee collection hierarchy indicates that the `Committee` collective class contains objects of two classes. `Democracy` and `Dictatorship`.

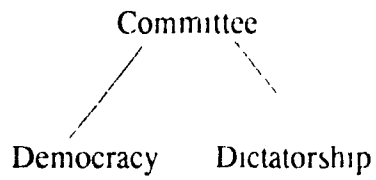


Figure 1.2 Committee collection hierarchy.

The collection hierarchy is used to ensure that the method sent to the collective class can be interpreted by each subobject according to the class of that object. In other words, the method applied to each subobject of the collective class has the implementation defined for that object's class or for its superclasses in the inheritance hierarchy.

Methods

Presently the Relix database language does not have an implementation of methods (functions or procedures). In this thesis we are not concerned with the actual implementation of methods, which is a topic for another thesis, rather, we assume that the methods are already available to the user. We propose the syntax of associating a method with a class.

```
function Valid_number() on Employee
 { if Emp# < 3 then return (True)
   else                return (False)
 }
```

Here the method `Valid_number()` is applicable only to the class on which it is defined (`Employee`) and to all its subclasses, namely `Permanent` and `Temporary`. It cannot be applied to `Person`

A method is invoked by actualizing, in the operand relation, the virtual attribute to which that method was assigned (Virtual attributes are discussed in more details in Chapter 2.) In the example below, the function `Valid_number()` is invoked in the `Employee` relation.

```
let validity be Valid_number ();
result ← [Emp#, validity] in Employee;
```

```
result (Emp#,validity)
          1     True
          2     True
          3     False
```

A ***polymorphic method*** is a method which has a different implementation for different classes of arguments The function `Valid_number()` can be written to validate both the employee number and the student number, which are possibly of different types (one is numeric and the other is alphanumeric).

```
function Valid_number() on Student
 { if the last letter of Stud# = "A" or "H" then return (True)
   else                                      return (False)
 }
```

The system should determine which implementation of this function to use when the function is applied to a class.

1.4.1 Thesis Outline

All of the above proposals are discussed in detail in the body of this thesis which is organized into 7 chapters

Section 1.5 of Chapter 1 reviews the commercially available object-oriented programming languages and database systems with the special emphasis on their implementation of inheritance and collection hierarchies, polymorphism, and encapsulation

Chapter 2 gives a general overview of Relix database programming language developed at McGill University. This chapter presents all of the features of Relix which are going to be used in the later chapters

Chapter 3 explores the correspondence between classes in object oriented programming languages and relations in relational database management systems (***classes***) The representation of ***Object Identity*** as an attribute of the relation is discussed. The chapter then introduces the ***class hierarchy*** and its representation in relational format The mechanism of attribute inheritance (***ISA hierarchy***) is explained on the examples of traditional relational operations like projection, selection, join and update

Chapter 4 suggests the syntax for the declaration of ***methods*** that are applicable to the limited number of classes It focuses on method inheritance among classes of the ISA hierarchy, and on the use of polymorphic functions and procedures

Chapter 5 introduces the collection hierarchy (***HASA Hierarchy***), its declaration, and application.

Chapter 6 demonstrates how Relix, enhanced with the features described in the previous chapters, can be used to implement a graphics editor written in Objective-C This chapter compares the implementation in both languages on the basis of class hierarchy and simple operations like creating and selecting objects, moving selected objects, and hiding and showing constraints.

Chapter 7 concludes our discussion with a summary of the main results and gives directions for further research.

1.5 Overview of Object-Oriented Programming Languages and Databases

In this section we will examine a number of object-oriented programming languages (OOPL) and databases (OODB). Among the commercially available OOPLs we will discuss Simula, Smalltalk [Gold83, Shaf91], Eiffel [Jell90, Meye88a, Meye88b], C++ [Stro86, Mull89], Trellis [Shaf86], and CLOS [Brac90, Moon89]. Among the OODBs we will discuss O_2 [Deux90, Deux91], ObjectStore [Lamb91], GemStone [Bret89, Butt91, CoMa84, Penn87], ORION [Bane87, KimB88] and POSTGRES [Rowe87, Ston86, Ston91].

Object-oriented databases support the following features that are not present in object-oriented programming languages:

- persistent objects,
- schema modifications (changes to the definition of the behaviour of a class and changes to the structure of the class lattice);
- methods associated with a class, which can be applied to every object of a class (whereas in OOPLs a method can be applied only to one object);
- each object has a unique, never-changing ID.

Simple inheritance is a standard feature of all OOPLs and OODBs. Most of the systems discussed in this overview support multiple inheritance in addition to simple inheritance.

All systems mentioned above provide opportunities to develop data aggregates. System-defined collection classes like sets, bags, lists and arrays specify a number of predefined procedures on the collection classes, such as adding a new element, verifying the existence of an element, enumerating elements, and removing an element. Composite objects organize a collection of related objects into a hierarchical structure which captures the IS-PART-OF relationship between an object and its parent.

1.5.1 Using Existing Classes as Building Blocks for New Classes

New classes in object-oriented languages and databases can be created by inheriting the properties of existing, more general classes, and adding some properties specific to the new class. This is the principle of inheritance.

The variables of the classes can be of simple or structured data type. A basic (simple) data type is a data type whose values are regarded as nondecomposable, like integer or character. A structured (composite) data type consists of component elements which are related by some structure. Examples are an array of integers, a set of characters, and a structure which includes an integer and a character. A class definition can be viewed as a type declaration.

The notion of ***clients*** and ***suppliers*** binds class variable to specific classes. In Eiffel [Meye88a] a class A is said to be a ***client*** of a class B, and B is a ***supplier*** of class A, whenever A contains an entity declaration of the form e:B.

```
class PERSON feature
  f_name, l_name: STRING;
end -- class PERSON
```

```
class BOOK feature
  title: STRING;
  author: PERSON;
end -- class BOOK
```

In this example, BOOK is a client of PERSON because BOOK contains the attribute declaration: author: PERSON.

A class may be its own client, for example, an EMPLOYEE class might be of the form

```
class EMPLOYEE feature
  dept: STRING;
  manager: EMPLOYEE;
end -- class EMPLOYEE
```

The client-supplier relationship exists in all OOPLs since the instance variables can be of any type - simple (string, integer, etc.) or user defined.

Inheritance and the client relationship correspond to different needs. Inheritance

means "is", client means "uses". Inheritance is appropriate if every instance of A may also be viewed as an instance of B. The client relation is appropriate when every instance of A simply possesses one or more attributes of B. Clients of A only see A from its interface; hence clients are protected against future changes in A's implementation. When a class inherits from A, it gains access to A's implementation, which gives that class more power, but no protection.

1.5.2 Multiple Inheritance and Resolution of Ambiguities

All of the OOPLs and OODBs reviewed here except C++ and ObjectStore support multiple inheritance of attributes and methods. In some OOPLs like Smalltalk [Shaf91] and POSTGRES [Rowe87], if the ambiguities arise because a class inherits the same attribute name from multiple parents, creation of the new class is rejected.

Trellis [Shaf86] allows the same name to be defined in different parents of a class, but requires the programmer to resolve explicitly the ambiguities by prefixing the method or attribute name by the name of the parent class from which that method or attribute should be inherited. For example, the `Display` operation in `Bordered_Window` class first calls the `Display` operation in `Window` class and then draws a border around the window.

```
operation Display (me)
   /* Display an object in this window with a border around it
   is
     begin
       Window'Display (me);  /* use Window's Display
       Display_border (me),   /* display the border
     end;
```

The ***me*** keyword indicates that the operation is an instance operation. The invocation `Window'Display` can only occur in a subclass of `Window`.

In Eiffel [Meye88a] name clashes may be removed by introducing one or more ***rename*** subclauses in the ***inherit*** clause. For example, if both `Employee` and `Student` classes contain an attribute `Name` and a function `Calc_age`, then a class `Working_Student`

may still inherit from both as follows

```
class Working_Student
inherit
  Employee
    rename Name as E_Name, calc_age as E_calc_age
  Student
feature ..
end
```

Within both `Working_Student`, the clients and subclasses of `Working_Student`, the `Name` attribute from `Employee` will be referred to as `E_Name`, and the one from `Student` as `Name`. Renaming makes it possible to refer to the same feature under different names in different classes.

In ORION [KimB88, Bane87] an explicit conflict resolution is incorporated in the definition of the class by a ***:methods*** clause

```
(make_class Classname
    :superclasses ListOfSuperclasses
    :attributes ListOfAttributes
    :methods ListOfMethodSpec)
```

Here `ListOfMethodSpec` is a list of pairs `(MethodName, Superclass)` where `MethodName` is the name of a method to be inherited from the `Superclass`. Thus the `:method` construct allows the user to specify which methods are to be inherited from which superclasses. If the `:methods` construct is not specified, all the methods are inherited from all the superclasses, and conflicts are resolved on the basis of superclass ordering.

If an instance variable or a method with the same name appears in more than one superclass of a class C, the one chosen by default is that of the first superclass in the list of immediate superclasses of C, as specified by the programmer. Since this default conflict resolution schema hinges on the permutation of the superclasses of a class, ORION allows the user to change explicitly this permutation at any time.

CLOS [Brac90] resolves inheritance conflicts by defining an order of precedence which determines which class's characteristics dominate.

```
(defclass Person () (name))
(defclass Graduate (Person) (degree))
(defclass Doctor (Person) ())
(defclass Research Doctor (Doctor Graduate) ())
```

The *defclass* construct includes the name of the new class, a list of superclasses, and a list of its instance variables. To avoid inheritance ambiguities, CLOS linearizes the ancestor graph of a class to produce an inheritance list in which each ancestor occurs only once. The graph of `Research-Doctor` is linearized to `Research-Doctor, Doctor, Graduate, Person`. Using linearization, a CLOS multiple inheritance hierarchy is reduced to a collection of inheritance lists, one for each class. Each list can be interpreted using simple inheritance. Classes appearing earlier in this list are considered more specific than classes appearing later, and the characteristics of more specific classes dominate.

1.5.3 Method Overriding and Accessing Superclass's Methods

In all the languages mentioned in Section 1.5, a subclass may override the superclass's methods by providing a different implementation for that method under the same name. In C++ a subclass can provide a new implementation of a method inherited from its superclass only if that method was defined in the superclass as VIRTUAL.

In Smalltalk, C++, and ObjectStore, the subclass may access the superclass's methods with the keyword *super* even if that method is redefined in the subclass. In Trellis, the method's name should be prefixed by the name of the superclass, only the subclasses can reference the superclass's methods.

Self in Smalltalk is a pseudo-variable referring to the receiver of a message. When ^ is used in a method, it indicates that the value of the next expression is to be the value of the method.

When a method contains a message whose receiver is `self`, the search for the method

for that message begins in the instance's class, regardless of which class contains the method containing `self`. Consider an example.

```
class name:  One                    class name:  Two
superclass:  Object                 superclass:  One
methods                             methods
  test                                test
    ^1                                  ^2
  result1
    ^self test
```

An instance of each class will be used to demonstrate the method determination for messages to `self`. `Example1` is an instance of class `One` and `example2` is an instance of class `Two`.

```
example1 ← One new.
example2 ← Two new.
```

expression	result
example1 test	1
example1 result1	1
example2 test	2
example2 result1	2

The pseudo-variable `self` can be used to implement recursive functions. An example of the method `Factorial` is given below:

```
Factorial
    self = 0 ifTrue: [^1].
    self < 0 ifTrue: [self error: 'Factorial invalid']
          ifFalse: [^self * (self - 1 ) Factorial]
```

The receiver is an integer. The first expression returns 1 if the receiver is 0. The second expression notifies the user of an error if the sign of the receiver is negative. Otherwise, the value returned is the receiver multiplied by the factorial of one less than the receiver

```
self * (self − 1) Factorial
```

Super in Smalltalk refers to the receiver of the message, just as `self` does. However, when a message is sent to `super`, the search for a method does not begin in the receiver's class. Instead, the search for a method begins in the superclass of the class containing the method. The use of `super` allows a method to access methods defined in a superclass even if those methods have been overridden in subclasses. Messages to `super` will be explained using two more example classes.

```
class name:  Three
superclass:  Two
methods
  test
    ^3
  result2
    ^self result1
  result3
    ^super test
```

```
class name:  Four
superclass:  Three
methods
  test
    ^4
```

We will illustrate the results of sending the messages test, result1, result2 and result3 to instances of classes Three and Four.

```
example3 ← Three new.
example4 ← Four new.
```

expression	result
example3 test	3
example4 result1	4
example3 result2	3
example4 result2	4
example3 result3	2
example4 result3	3

In Eiffel [Meye88a], the superclass's method can be accessed by renaming that method in the current class. For example, in class `Three` we will rename the method `test` of class `Two` in order to enable access to it. The predefined entity name `Current` allows the reference to the class of the current instance. It is similar to `self` in Smalltalk.

```
class Three
inherit  Two
         rename test as two_test
feature
   test:INTEGER is
      do Result := 3
      end;
   result2:INTEGER is
      do Result := Current.result1
      end;
   result3:INTEGER is
      do Result := two_test
      end;
end -- class Three
```

1.5.4 Methods

Methods can be associated with a specific class, in which case they are inherited by all the subclasses of that class. This type of methods can only be used with the objects of the classes on which these methods are defined. Method inheritance means that the building blocks from which a class is constructed may include behaviour in the form of methods as well as structure.

In Eiffel [Meye88a], attributes and routines are grouped under the same category of ***features***.

```
class POINT
feature
    x, y: REAL;
    scale (factor:REAL) is
        -- Scale by a ratio of factor
      do
        x := factor * x;
        y := factor * y;
      end;  -- Scale

    distance (other:POINT): REAL is
        -- Distance from current point to other
      do
        Result := sqrt ((x - other.x)^2 + (y - other.y)^2)
      end;  -- distance
end  -- class POINT
```

The text of an Eiffel class [Meye88a] always refers to a current instance of the class. Most of the time this current instance is anonymous: in a class, a feature name which appears unqualified denotes the corresponding feature of the current instance. So in class POINT, the unqualified feature name x (i.e. just x, not p.x for some p of type POINT) would denote the corresponding feature of the current instance.

In CLOS [Brac90, Moon89] each method specifies its own applicability condition, the most common kind of which is a test of whether the argument passed to that method is a member of a particular class on which this method is defined. Thus all instances of a class have the same behaviour, because the same methods are applicable to each of them.

```
(defclass Person () (name))
(defmethod Display ((self 'Person))
   (display (slot-value self 'name)))

(defclass Graduate (Person) (degree))
(defmethod Display ((self Graduate))
     (call-next-method)
     (display (slot-value self 'degree)))
```

The argument list of the `defmethod` expression defines the class on which the method is defined. Method combination is supported by `call-next-method`, which plays the role of `super` in Smalltalk. `Call-next-method` provides access to the second most specific method in the inheritance chain with the same message selector. For example, when a `Display` message is sent to a `Graduate` student whose name is A.Smith and who is doing a Ph.D. degree, "A.Smith Ph.D." is displayed.

POSTGRES [Rowe87,Ston86,Ston91] system allows its methods to be written in languages like C or Lisp, and in POSTQUEL query language. The former methods are called ***user-defined***, and the latter are termed ***POSTGRES methods***

A ***user-defined method*** is defined to the system by specifying the names and types of the arguments, the return type, the language it is written in, and where the object code is stored. For example, the definition

define procedure AgeInYears (date) **return** int4
is (language = "C", **filename** = "CalculateAge")

declares a procedure `AgeInYears` which is written in C and whose object code is stored in the filename "CalculateAge". This procedure takes an argument of the type `date` and returns an integer value. The argument and return types are specified using POSTGRES types, such as characters, integers, etc. When the procedure is called from a relation, it is executed once for each tuple of that relation, taking the specified attributes as input arguments.

POSTGRES stores the information about a procedure in the system catalogues and dynamically loads the object code when it is called in a query. The following query uses the `AgeInYears` procedure to retrieve the names and ages of all people:

retrieve (P.Name, Age = AgeInYears (P Birthdate)) **into** AgeInfo
from P **in** Person

Consider that the relation `Person` has two tuples: one for each Ann and Pete. The result of the evaluation of the query above by invoking a procedure `AgeInYears` on each tuple of `Person`, will give the relation `AgeInfo`.

```
Person (Name,Birthdate)     AgeInfo (Name,Age)
        Ann  55/01/20                Ann  36
        Pete 65/09/30                Pete 26
```

User-defined procedures can also take tuple-variable arguments. For example, the following command defines a procedure, called Comp, which takes a tuple of the EMPLOYEE class and computes the employee's compensation according to some formula which involves several attributes in the tuple (e.g. the employee's status, job title, salary):

define procedure Comp (EMPLOYEE) **returns** int4
is (language = "C", **filename** = "Comp")

The argument of class EMPLOYEE represents a reference to a tuple in the EMPLOYEE relation. This procedure is called in the following query:

retrieve (E.Name, Compensation = Comp(E)) **from** E **in** EMPLOYEE

A data structure which contains the names, types, and values of the attributes in the tuples is passed to the C function that implements this procedure.

POSTGRES methods are procedures stored in the attributes of POSTGRES relations. The system provides two kinds of procedure type-constructors: variable and parameterized. A **variable procedure-type** allows a different POSTQUEL procedure to be stored in each tuple, while **parameterized procedure-types** store the same procedure in each tuple but with different parameters.

We illustrate the use of a ***variable procedure-type*** by showing how to determine the student's major. Suppose that a DEPARTMENT relation was defined with the following command:

create DEPARTMENT (Name = char[25],Chair = char[25],...)

A student's major(s) can then be represented by a procedure in the STUDENT relation that retrieves the appropriate DEPARTMENT tuples(s). The attribute Majors would be declared as follows:

create STUDENT (..., Majors=postquel,...)

Data type postquel represents a procedure-type. The value in Majors will be a query which fetches those tuples of the DEPARTMENT relation which represent the student's majors. When that query is executed, Majors will store the tuples fetched by the query. Thus Student is

a nested relation. The following command appends a student to the database who has a double major in Math and C.Sc.:

append STUDENT (Name= "Smith",..., Majors =
"retrieve (D.**all**) **from** D **in** DEPARTMENT
where D.Name = "Math" **or** D.Name = "C Sc."")

A query which references the Majors attribute returns the string that contains the postquel command. Two notations are provided that will execute the query and return the result, rather than the string with the postquel command. First, nested-dot notation implicitly executes the query:

retrieve (S.Name, S.Majors.Name) **from** S **in** STUDENT

This prints a list of names and majors of students. The result of the evaluation of the query, stored in `Majors`, is implicitly joined with the tuple specified by the rest of the target list In other words, if a student has two majors, this query will return two tuples with the Name attribute repeated. The implicit join is performed to guarantee that a relation is returned

The second way to execute the query is to use the ***execute*** command

execute (S.Majors) **from** S **in** STUDENT **where** S.Name = "Smith"

returns a relation which contains DEPARTMENT tuples for all of Smith's majors.

Parameterized procedure-types are used when the query to be stored in an attribute is nearly the same for every tuple. The query parameters can be taken from other attributes in the tuple or they may be explicitly specified. For example, suppose an attribute in STUDENT represents the student's current class list. Given the definition for ENROLMENTS,

create ENROLMENT (Student = char[25],Class = char[25])

Bill's class list can be retrieved by the query

retrieve (ClassName = E.Class) **from** E **in** ENROLMENT
where E.Student = "Bill"

This query will be the same for every student,except for the constant that specifies the

student's name

A parameterized procedure-type can be defined to represent this query as follows:

define type classes **is**
    retrieve (ClassName = E Class) **from** E **in** ENROLMENT
    where E.Student = $.Name
end

The dollar-sign symbol ($) refers to the tuple in which the query is stored (i.e. the current tuple). The parameter for each instance of this type (i.e., a query) is the Name attribute in the tuple in which the instance is stored. This type is then used in the create command,

create STUDENT (Name = char[25],...,ClassList = classes)

to define an attribute which represents the student's current class list. This attribute can be used in a query to return a list of students and the classes they are taking:

retrieve (S.Name, S.ClassList ClassName)

Notice, that for a particular STUDENT tuple, the expression "$.Name" in the query refers to the name of that student. The symbol "$" can be thought of as a tuple-variable which is bound to the current tuple.

1.5.5 Collection Classes

A **collection** groups related objects together, so that they can be referred to as a single entity. The elements of the collection can belong to different classes. Collection classes are system-defined and support four categories of messages for accessing elements: 1) adding new elements; 2) removing elements; 3) testing occurrences of elements; and 4) enumerating elements. Also the size of the collection can be determined. The enumeration messages are useful when the same operation needs to be applied to all the elements of the collection. For example, `Figure` can include such graphical objects as point and circle. To move every object of the `Figure` by a certain distance, apply the `move` method to every object of the `Figure`.

The Smalltalk-80 syntax for this problem is as follows:

```
class name:                Point
superclass:                Graphical_objects
instance variable names:   x, y
methods
  newx:xInt y:yInt
    /* To create an instance of a point, both x and y coordinates are specified. */
    x ← xInt.
    y ← yInt.

  moveHoriz: hInt Vert: vInt
    /* Both coordinates of the point are changed by the corresponding amount. */
    x ← x + hInt.
    y ← y + vInt.


class name:                Circle
superclass:                Graphical_objects
instance variable names:   center, radius
methods
  newCenter: aPoint rad: Int
    /* Create an instance of a Circle class. The center instance variable should be
       just like the instance aPoint of the class Point. */
    center ← aPoint copy.
    radius ← Int.

  moveHoriz: hInt Vert: vInt
    /* When moving a circle, only its center needs to be moved.*/
    center moveHoriz: hInt Vert: vInt.


class name:                Figure
superclass:                Object
instance variable names:   components
methods
  new
    /* This method makes the variable components to be a Set. */
    components ← Set new.

  addComponent: anObject
    /* Add a new object of any class to the components set. */
    components add: anObject.

  moveHoriz: hInt Vert: vInt
    /* Apply the method moveHoriz: Vert: to each element of the components set. */
    components do: [:each | each moveHoriz: hInt Vert:vInt].
```

A `geometricFigure` consists of a `point` and a `circle`. First, create each graphical object, and then add it to the set of components of the `geometricFigure`.

```
point1 ← Point newx:0 y:0.
point2 ← Point newx 0 y 10
circle1 ← Circle newCenter:point2 rad:3.
geometricFigure ← Figure new.
geometricFigure ← addComponent: point1.
geometricFigure ← addComponent: circle1.
```

If we want to move the whole `geometricFigure` by a certain distance, we will apply the `moveHoriz: Vert:` method to the `geometricFigure`.

```
geometricFigure moveHoriz:2 Vert:4.
```

Collections can be thought of as lists of objects which can be stored in any of several forms: sorted or unsorted, ordered or unordered, with or without duplicates. Sets and lists are subclasses of the Collection class and are supported by such languages as Trellis, O_2, Smalltalk, C++, ObjectStore, GemStone and POSTGRES.

In some languages, like ObjectStore, collections cannot store objects of different types (classes). So the Figure example above cannot be implemented in those languages.

In GemStone, a class defines the structure of its instances, but rarely keeps track of all those instances. Instead, collection objects --Arrays, Bags, Sets-- serve to group those instances. An object may belong to more than one collection [Bret89].

1.5.6 Sending a Message to All the Subclasses of a Class

In POSTGRES [Rowe87] a * after the class name indicates that the query should be run over the class and all its subclasses. For example, if we have the following hierarchy of classes,

create EMP (name = c12, salary = float, age = int)
create SALESMAN (quota = float) **inherits** EMP

then, if we wanted the names of all salesmen or employees over 40, the notation would be:

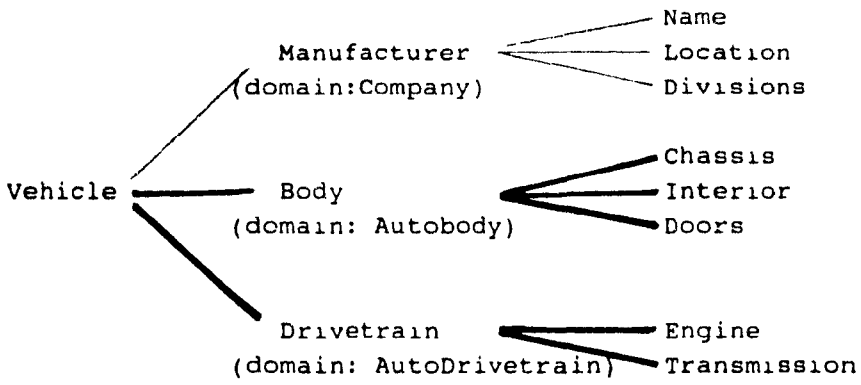
retrieve (E.name) **from** E **in** EMP* **where** E.age > 40

1.5.7 Composite Objects

A **composite object** is a collection of related instances. These instances form a hierarchical structure which captures the IS-PART-OF relationship between an object and its parent. ORION [Bane87] uses the knowledge of composite objects not only to enforce the semantics of composite objects but also to cluster physically the constituent objects of composite objects, so as to minimize the I/O cost of retrieving composite objects.

A composite object has a single root object, and the root references multiple children objects, each through an instance variable. Each child object can in turn reference its own children objects, again through instance variables. A parent object exclusively owns children objects, and as such the ***existence of children objects depends on the existence of their parent***. Children objects of an object are thus dependent objects. The instances that constitute a composite object belong to classes in the inheritance hierarchy. This hierarchical collection of classes is called a **composite object schema**.

Below we illustrate a composite object schema for vehicles.



The classes that are connected by bold lines form the composite object schema. The root class is the class `Vehicle`. Through instance variables `Body` and `Drivetrain`, `Vehicle` instances are linked to their dependent objects, which belong to classes `Autobody` and `AutoDrivetrain`. The class `Vehicle` has another instance variable called `Manufacturer`, but it is not a link to a dependent object because `Manufacturer` is not exclusively owned by the `Vehicle`. The instances of `AutoBody` and `AutoDrivetrain` are connected in turn to other dependent objects. A `Vehicle` composite object, then, is an instance of the class `Vehicle`, as well as an instance of `AutoBody` and `AutoDrivetrain` classes.

Each non-leaf class of a composite object schema has one or more instance variables, called **composite instance variables**, that serve as links to dependent classes In other words, a composite object schema is created through composite instance variables which have dependent classes as their domains. The link between a class and the domain of a composite instance variable of the class is a **composite link**. For example, the `Vehicle` class has a composite link to the class `AutoBody` through the instance variable `Body`. The instance variable `Body` has as domain the class `Autobody`, and it has the composite link property

The object which is referenced through a composite instance variable is a dependent object whose existence depends on the existence of the referencing object. For example, the body of a vehicle is not only owned by one specific vehicle, but it also cannot exist without the vehicle. This means that a dependent object cannot be created if its owner does not already exist As such, a composite object must be instantiated in a top-down fashion: the root object of a composite object must be created first, then the objects at the next level, and so on. When a constituent object of a composite object is deleted, all its dependent objects must also be deleted

The composite link property of an instance variable of a class is inherited by subclasses of that class. For example, if the class `Automobile` is a subclass of `Vehicle`, it inherits the instance variable `Body` from `Vehicle`. Furthermore, because `Body` is a composite link in the `Vehicle` class, it will also be a composite link in the `Automobile` class.

1.5.8 Schema Modification

A schema is described by the set of class definitions connected by the superclass / subclass relationship, i.e. it is represented by the class definitions and the structure of the class lattice [UnSc90] Schema change operations fall into three categories [NgRi89]:

1. changing class definitions, i.e , adding and deleting instance variables;
2. adding and deleting classes in the lattice;
3. modifying the class lattice by changing the relationship between classes.

In OOPLs schema modification is not allowed, since classes are types and they do not contain any objects.

Below we review the semantics of the schema change operation in ORION [Bane87]

1 Changing class definition.

1.1 Add a new instance variable V to a class C.

If V causes a name conflict with an inherited instance variable, V will override the inherited one. If the old instance variable was defined locally in C, it is replaced by the new definition. V is inherited by all subclasses of C. If there is a name conflict in a subclass, V is not inherited. Existing instances of the class and subclasses receive the user-specified default value, if there is one, or the nil value.

1.2 Drop an instance variable V from a class C.

V must have been defined in the class C; it is not possible to drop an inherited instance variable. V is dropped from C and all its subclasses. If C or any of its subclasses has other superclasses which have an instance variable of the same name as V, C inherits one of those instance variables. In the case that V must be dropped from C or any of its subclasses without a replacement, existing instances lose their value for V.

2 Adding and removing classes.

2.1 Add a new class C.

If no superclasses are specified for a newly-added class, the root class OBJECT is the default superclass of the new class. If multiple superclasses are specified, all instance variables and methods from all superclasses of C are inherited by C.

2.2 Drop an existing class C.

All the subclasses of C become immediate subclasses of the superclasses of C. The definition of C is dropped, together with all its instances. If the class C was the domain of an instance variable V_1 of another class C_1, V_1 is assigned a new domain, namely, the first superclass of the dropped class C.

3 Modifying the class lattice

3.1 Make a class S a superclass of a class C.

The addition of a new edge from S to C must not introduce a cycle in the class lattice. C and its subclasses inherit instance variables and methods from S.

3.2 Remove a class S from the superclass list of a class C.

The deletion of an edge from S to C must not cause the class lattice DAG to become disconnected. In the case that S is the only superclass of C, then C is made an immediate subclass of each of S's superclasses. Thus C does not lose any instance variables or methods that were inherited from the superclasses of S. C only loses those instance variables and methods that were defined in S. Dropping of those variables is discussed in 1.2.

GemStone [Bret89] allows a class to be added as a leaf of the class hierarchy. To add an interior node to the hierarchy, the new class's name, its superclass, and its immediate subclasses are specified. GemStone's schema modification semantics is very similar to that of ORION except for the following differences.

1.2 Drop an instance variable V from a class C.

The modification is not propagated to subclasses of the modified class. The same effect may be achieved by repeatedly applying the operation to the modified class's subclasses.

2.1 Add a new class C.

To add an interior node to the hierarchy, the new class's name, its superclass, and its subclasses are specified. The subclasses must currently be immediate subclasses of the given superclass. The representation specified by the new class will be the same as that specified by the superclass. No new variables are introduced into the class and the constraints on inherited variables remain unchanged. Note that new instance variables can be added subsequently.

2.2 Drop an existing class C.

A class may not be removed if it has any instance GemStone has a message by which an object can change its class to a subclass of its current class; this message can be used to remove all instances of a given class. The superclass of all instances of the removed class is changed to the superclass of the removed class.

In O_2 [Deux91] the schema designer is not forced to follow any order when creating classes. Therefore, classes can be momentarily incompletely specified. For instance, a class C can be defined with an attribute of class C', which is not defined But instances of that class cannot be created unless the class definition is well defined. Deletion of a class which is not a leaf in the inheritance hierarchy is forbidden. A class is deleted only if a) it has no instances, and b) no other classes are dependent on it through composition or specialization Every time a class is deleted, all the methods associated with it are invalidated.

1.5.9 Persistent Objects and Object Identity

In all the OOPLs objects are not persistent They are referenced by a pointer.

In the OODBs like O_2, ObjectStore and GemStone there are two types of objects: persistent and transient. By default, all the objects are transient unless they are made persistent sometime after their creation and before the end of the execution of the program Persistent objects are assigned a name which becomes their object identity In POSTGRES [Rowe87, Ston91], each instance of a class has a unique (never changing) system-defined identifier (OID), thus making all the objects persistent. The OID can be accessed, but not updated by the user.

1.5.10 Inheritance of Features, Private and Public Options

In C++ and O_2 all the attributes and methods by default are private, and are only accessible to the methods defined in the declaration of that class. In O_2, features explicitly declared ***public*** are inherited. On the other hand, in Simula and Trellis all the features are inheritable by default, unless they are explicitly defined as ***hidden*** or ***private*** respectively. In Smalltalk, Eiffel and POSTGRES, all of the features of the superclass are inherited by the subclass. In Eiffel, features and methods listed in the export clause are available to the clients of this class. Inherited features can be exported too.

In Trellis and in C++ ***subtype-visible*** methods are inherited and can be redefined, but are not visible outside the defining type and its subtypes. This type of methods is not as restrictive as private but not as general as public.

In C++, a class may name other classes to be its ***friends***; this allows access to the private members of the class by the methods of these friend classes.

Chapter 2

Relix Overview

Relix, **Rel**ational database on **Unix**, is a database language developed in the Aldat project. The Aldat project [Merr77] explores extensions and applications of the relational algebra. The extensions have evolved through a careful empirical process of developing applications of the existing formalism. Extension has been done only where necessary, and only if the extension fits into a simple conceptual framework. The basis of Aldat is described in [Merr84]. This chapter outlines the subset of Relix that is used in this thesis.

2.1 Domains and Relations

An attribute of the relation is associated with a set of values called a domain. Relix supports the following domain types:

intg - an integer type

real - a real type

strg - a variable length character string

bool - boolean type

stmt - executable statement

Before an attribute can be used in the declaration of a relation, it should be declared to be of one of the above types. Let us declare several attributes.

domain Age **intg**;

domain Name **strg**;

domain Occup **strg**;

A relation can be created in several ways.

- It can be assigned the value of a file located in the secondary storage, using the relation declaration syntax,

  relation Person (Name, Age, Occup) ← "P";

  The newly-created relation `Person` may have these tuples:

```
Person (Name,  Age,  Occup    )
        Ann    25    Employee
        Pete   31    Student
        Kate   26    Student
        John   27    Employee
        Jake   21    Student
```

- It can be assigned the value that is a result of a relational algebra operation on existing relations. The expression R ← T replaces the contents of the relation R with the contents of the relation T if relation R has already existed, and creates R being identical to T if relation R did not exist before this operation. In both cases R is assigned the same attributes as T.

  The expression R <+ T appends the tuples of T to R if both relations are defined on the same attributes. Otherwise an error occurs.

Example 2-1. Consider four relations and two assignment statements:

```
Info           Sales          Cs202          Final
(Name, Age)    (Dept, Sale)   (Name, Mark)   (Name, Mark)
 Jack   30      Toy    200     Pete   80      Pete   75
                Deli   150     Vera   70      Vera   70
                                               Nick   60


Info ← Sales;                          Cs202 <+ Final;
```

The results of each assignment are shown below.

```
Info (Dept, Sale)                 Cs202 (Name, Mark)
      Toy   200                          Pete   80
      Deli  150                          Vera   70
                                         Pete   75
                                         Nick   60
```

■

There are two types of relations in Relix: system- and user-defined. Above we discussed the user-defined relations `Person`, `Info`, `Sales`, etc. These two types of relations are easily distinguishable by their name. The names of system-defined relations as well as their domains is prefixed with the dot (.), while user-defined relation names are not. The most important system relation used in this thesis is `.rd (.rel_name,.dom_name)`,

which contains information about the structure of each relation in the system. That is, for every relation `.rd` lists the domains on which that relation is defined. For example, relation `Info` in the Example 2-1 is represented in the `.rd` system relation by two tuples .

```
.rd (.rel_name, .dom_name)
        Info       Dept
        Info       Sale
```

2.2 Projection and Selection Operations

Projection is a **vertical operation** on a relation which specifies a subset of the attributes of a relation. Note: the resulting relation is defined on the projected attributes and contains no duplicate tuples. To find all the occupations of the people in the Person relation, project `Person` onto its attribute `Occup`.

Occupations ← [Occup] **in** Person;

```
Occupations ( Occup    )
             Employee
             Student
```

Note: the relation `Person` has five tuples and the relation `Occupations` has only two tuples This is because the values "Employee" and "Student" appear two and three times, respectively, in the attribute `Occup` of `Person`.

Selection is a **horizontal operation** which extracts a subset of relation's tuples in which every tuple satisfies a given criterion. We can isolate all the employed people into a relation `Working_People` by selecting corresponding tuples from `Person`.

Working_people ← **where** Occup = "Employee" **in** Person;

```
Working_people (Name, Age, Occup    )
                Ann   25    Employee
                John  27    Employee
```

Relix allows the ***project*** and ***select*** operations to be combined in a single T-Selector expression. For instance, we can find the names of people who are under 30 years of age

Under_30 ← [Name] **where** Age < 30 **in** Person;

```
Under_30 (Name)
          Ann
          Kate
          John
          Jake
```

2.3 Join

As relations are the generalizations of sets, Relix has relational operators which are generalizations of the set operators. The set-valued set operations such as union, intersection, difference, etc. belong to the class of μ-joins. The logic-valued set operations such as inclusion, empty intersection, etc are extended to the class of σ-joins. Only those join operations that are used in this thesis are discussed in detail below.

2.3.1 μ-joins

The **natural join** (***ijoin***) is the most common member of the μ-join family. The two operand relations are joined on the common attribute and only the tuples that have the same value of that attribute in both relations are written into the resulting relation.

Example 2-2 Given two relations,

```
Stud_info (Name, Course,exam1,exam2)    Person (Name, Age, Occup    )
          Pete  CS202   70    60                 Ann   25   Employee
          Kate  Math251 80    70                 Pete  31   Student
          Jake  Phys420 90    60                 Kate  26   Student
                                                 John  27   Employee
                                                 Jake  21   Student
```

the natural join of these relations associates tuples of `Person` with those tuples of `Stud_info` that have the same value of `Name`.

Students ← Person **ijoin** Stud_info;

```
Students (Name, Age, Occup,  Course, exam1,exam2)
         Pete  31   Student Cs202    70    60
         Kate  26   Student Math251  80    70
         Jake  21   Student Phys420  60    10
```

■

The attributes participating in the join (in this example, Name) are called the **join attributes**, and may be specified implicitly or explicitly. If the join attributes are not specified, then we have two cases: a) if the two relations share some common attributes, then the join is computed on those commonly named attributes (see Example 2-2 above); b) if relations do not have any common attributes, then the natural join computes a cartesian product of those relations.

Example 2-3. Explicit specification of join attributes.

Given two relations, Income_by_city,

```
Income_by_city (City,     Status, income)
                Montreal  employee  35
                Montreal  owner     50
                Toronto   employee  40
                Toronto   owner     60
```

and Person (Example 2-2). Find the salary that the working people can expect in the cities mentioned in the Income_by_city relation.

Personal_incomes ← Person [Occup **ijoin** Status] Income_by_city;

```
Personal_incomes (Name, Age, Occup,    City,    income)
                  Ann    25  employee  Montreal  35
                  Ann    25  employee  Toronto   40
                  John   27  employee  Montreal  35
                  John   27  employee  Toronto   40
```

■

The **union join** (***ujoin***) corresponds to the set union. In general, the union join consists of three disjoint sets of tuples: the center, the left wing, and the right wing. For the given operand relations, R(X,Y) and S(Y,Z), these three sets of tuples are each defined on the attributes X,Y,Z. The ***center*** is the join of the relations R and S. The ***left wing*** consists of all the tuples from R that match no tuple from S, augmented by the null value in the attribute Z. The ***right wing***, conversely, consists of all the tuples from S that match no tuple from R, augmented by null values in the attribute X [Merr84].

Example 2-4. Given two relations,

```
Staff (Name,Emp#,Position)    Alumnae (Name,Major,Grad_year)
       Pete 123  Professor              Pete  C.Sc.  1980
       Ann  345  Secretary              Kate  Psyc   1982
       Jake 567  Janitor                John  Chem   1988
```

find the complete information about all the attendants of the university reunion, both the staff and the alumnae.

Attendants ← Staff **ujoin** Alumnae;

```
Attendants (Name,  Emp#,  Position,  Major,  Grad_year)
            Pete   123    Professor  C.Sc.    1980   | center
            Ann    345    Secretary  NULL     NULL   | left wing
            Jake   567    Janitor    NULL     NULL   |
            Kate   NULL    NULL      Psyc     1982   | right wing
            John   NULL    NULL      Chem     1988   |
```

■

The **difference join** (*djoin*) corresponds to the set difference. In general, difference join consists of right wing tuples (see the definition above).

The **left join** (*ljoin*) consists of the center and the left wing tuples

Example 2 5. Difference join.

Consider the relations in Example 2-4. Find all the employees who are not alumnae.

Not_alumnae ← Staff **djoin** Alumnae;

```
Not_alumnae (Name, Emp#, Position)
             Ann    345  Secretary
             Jake   567  Janitor
```

■

Example 2-6. Left join.

Consider the relations in Example 2-4. Find the educational background of the university employees.

Education_of_Staff ← Staff **ljoin** Alumnae;

```
Education_of_Staff (Name, Emp#,Position,  Major,Grad_year)
                    Pete  123  Professor  C.Sc.   1980
                    Ann   345  Secretary  NULL    NULL
                    Jake  567  Janitor    NULL    NULL
```

■

2.3.2 σ-joins

A feature common to all σ-joins is that the join attributes are excluded from the resulting relation.

The **natural composition** (*icomp*) is similar to the natural join operation. It extracts the tuples whose join attributes' values are identical in both relations.

Example 2-7. Natural composition.

Consider the relations in Example 2-4. Find the employees who are alumnae.

Alumnae_Staff ← Staff **icomp** Alumnae;

```
Alumnae_Staff (Emp#,Position,  Major,Grad_year)
               123  Professor  C.Sc.   1980
```

■

2.4 Update

There are three types of update operations on relations: add new tuples, delete tuples, and change attribute values of some tuples.

To add the tuples in S to the relation R, we write

update R **add** S;

To delete the tuples that are both in S and in R from R, we write

update R **delete** S;

To see the results of the execution of the update operations described in this section, consider the following examples relations.

```
Dept_sales                          Org_sales
(Dept      month  sales)           (Dept      month  sales)
 Toys      April  200               Toys      April  250
 Video     May    500               Video     April  450
 Stereo    May    400               Hardware  May    250
 MansWear  April  350               MansWear  April  350
```

In the following examples assume that `Dept_sales` contains the above values every time we enter a new command. New tuples are marked in bold.

Example 2-8. ***Add*** clause in the Update statement.

update Dept_sales **add** Org_sales;

Result:

```
Dept_sales (Dept      month  sales)
            Toys      April  200
            Video     May    500
            Stereo    May    400
            MansWear  April  350
            Toys      April  250
            Video     April  450
            Hardwear  May    250
```

■

Example 2-9. ***Delete*** clause in the Update statement.

update Dept_sales **delete** Org_sales,

Result:

```
Dept_sales (Dept    month  sales)
            Toys    April  200
            Video   May    500
            Stereo  May    400
```

■

Changing a relation involves modifying values of the specified attributes in selected tuples. The general syntax of this operation is

update R **change** attr1 ← val1 {,attr2 ← val2,...} {**using** rel_expr};

where {} indicates optional syntax.

The syntax of the update command has two clauses: the ***change*** clause specifies all the modifications to the attributes, while the ***using*** clause specifies the selection criteria for the tuples to which the modifications are to be applied. In the absence of the ***using*** clause the modifications are applied to all the tuples of the relation being updated.

The tuples to be updated are selected by joining the relation resulting from the evaluation of the relational expression in the ***using*** clause with the relation being updated.

Example 2-10. ***Update*** statement with ***change*** clause.

update Dept_sales **change** sales ← 300;

Since the ***using*** clause is not specified, every tuple in the `Dept_sales` is changed.

Result:

```
Dept_sales (Dept      month sales)
           Toys       April  300
           Video      May    300
           Stereo     May    300
           MansWear   April  300
```

■

Example 2-11. ***Update*** statement with ***change*** clause and a relational expression in the ***using*** clause.

update Dept_sales **change** month ← "April" **using**

([Dept] **where** month = "April" **in** Org_sales);

Only those tuples of `Dept_sales` whose `Dept` is listed in the `Org_sales` for the month of April are updated.

Result:

```
Dept_sales (Dept      month sales)
           Toys       April  200
           Video      April  500
           MansWear   April  350
           Stereo     May    400
```

■

2.5 Domain Algebra

The domain algebra defines **virtual attributes** which may be actualised when needed When utilized to its full potential, the domain algebra provides facilities such as arithmetic, totalling, ordering, etc. A thorough description of the domain algebra may be found in [Merr84].

2.5.1 Scalar Operations

The simplest operation of the domain algebra generates the value in a tuple for the virtual attribute only in terms of the values in the same tuple of the operand attributes This operation works along a tuple and is sometimes referred to as a **horizontal operation** We define a scalar virtual attribute as follows:

let GPA **be** 4; ≪ define a constant attribute ≫
let seqn1 **be** seqn; ≪ rename an attribute ≫
let final_mark **be** (exam1 + exam2)/2;
let passed **be if** final_mark > 55 **then** true
else false;

It is implicit from the above examples that virtual attributes can be defined in terms of each other, for example, `passed` depends on the `final_mark`, which is in turn defined in terms of other attributes.

Virtual attributes are not associated with any relation until they are actualised, either by a projection or a selection. For example, expression

Stud_record ← [Name,exam1,exam2,final_mark,GPA] **in** Stud_info;

will create the relation

```
Stud_record (Name,exam1,exam2,final_mark,passed,GPA)
            Pete  70    60       65       True   4
            Kate  80    70       75       True   4
            Jake  60    10       40       False  4
```

The horizontal operator can be of various types.

- mathematical operator, such as +, −, *, /, ***mod***, **, ***abs***
- trigonometric function, such as ***cos***, ***sin***, ***tan***, ***log***, etc.
- logical operator, like <, <=, >, >=, =, ~=
- conditional assignment ***if..then..else..***
- concatenation of attributes and scalars by a ***cat*** operator.

A good illustration of the functionality of the **cat** operator is the task of combining the first and the last name fields together to obtain a full name.

Example 2-12. ***Cat*** operator.

Compose the `full_name` by concatenating the title "Dr. " with the fist name (`f_name`) and last name (`l_name`). When the virtual attribute `full_name` defined below

let full_name **be** "Dr. " **cat** f_name **cat** " " **cat** l_name;

is actualised in the relation `Professor`, it has these values:

```
Professors (l_name,f_name,dept)   full_name
            Peter   Smith  Chem    Dr. Peter Smith
            Mike    Wong   Phys    Dr. Mike Wong
```

■

2.5.2 Reduction

The reduction operations combine values from more than one tuple in a relation. They are sometimes referred to as the **vertical operations**.

The **simple reduction** produces a single result from the values of a single attribute of all tuples of an operand relation.

Example 2-13. ***Red*** operator.

Find the number of tuples in the `Stud_info` relation.

let no_students **be red** + **of** 1;

Find the average `final_mark` of the students.

let AVG **be** (**red** + **of** final_mark) / no_students;

When these virtual attributes are actualised by the expression

Stud_rec1 ← [Name,final_mark,no_students,AVG] **in** Stud_record;

the resulting relation looks like this:

```
Stud_rec1 (Name,final_mark,no_students,AVG)
          Pete      65          3         60
          Kate      75          3         60
          Jake      40          3         60
```

■

The **equivalence reduction** is like simple reduction, but produces a different result for different sets of tuples in the relation. Each set is characterized by all the tuples having the same value in some specified attribute -- an "equivalence class" in mathematical terminology. Subtotalling is an example.

Example 2-14. ***Equiv*** operator.

Count how many students have passed their courses and how many have failed.

let pass_fail_group **be equiv** + **of** 1 **by** passed;

stud_rec2 ← [Name,final_grade,passed,pass_fail_group] **in** Stud_record;

```
Stud_rec2 (Name,final_grade,passed,pass_fail_group)
          Pete      65        True        2
          Kate      75        True        2
          Jake      40        False       1
```

■

Only commutative and associative operators, like +, *, **and**, **or**, **max**, **min**, are permitted inside the reduction expression.

2.5.3 Functional Mapping

The second type of the "vertical" domain algebra operations involves an operand attribute and a controlling attribute. In this case, the controlling attribute serves to specify an order in which the tuples are to be processed. The two kinds of functional mapping are best presented through examples.

Simple functional mapping is illustrated by cumulative total.

Example 2-15. ***Fun*** operator.

Find the YTD income.

let YTD_income **be fun** + **of** salary **order** month;

Below we show the `YTD_income` virtual attribute actualised in the relation `Income`.

Income (month,	salary)	YTD_income
1	250	250
2	260	510
3	260	770

■

For a functional mapping to be properly defined, the operand attribute (for example, `salary`) must be functionally dependent on the controlling attribute (for example, `month`). The resulting virtual attribute is also functionally dependent on the controlling attribute.

Partial functional mapping extends simple functional mapping in the same way that equivalence reduction extends simple reduction.

Example 2-16. ***Par*** operator.

Find the cumulative sales in each department by the month.

let dept_cum_sale **be par** + **of** sales **order** month **by** dept;

The virtual attribute `dept_cum_sale` is actualised in the relation `Dept_sales` below.

Dept_sales (Dept	month	sales)	dept_cum_sale
Toys	April	200	200
Toys	May	250	450
Toys	June	240	690
Video	April	500	500
Video	May	670	1170

■

The permitted operators for the functional and partial functional mapping operations are +, -, *, /, **, **mod**, **pred**, **succ**, &, and |.

Of particular interest to us are the predecessor (***pred***) and successor (***succ***) operators. The ***pred*** (***succ***) operator gives the predecessor (successor) in the order indicated by the attribute of the ***order*** clause.

Example 2-17. ***Par pred of*** and ***par succ of*** operators.

For each month calculate the change in profit over the previous month's profit and the subsequent growth.

let last_sale **be par pred of** sales **order** num_month **by** Dept;

let next_sale **be par succ of** sales **order** num_month **by** Dept;

These virtual attributes are actualised in the relation `Dept_sales` as follows.

Dept_sales (Dept,num_month,sales)			last_sale	next_sale
Toys	4	200	240	250
Toys	5	250	200	240
Toys	6	240	250	200
Video	4	500	600	670
Video	5	670	500	600
Video	6	600	670	500

This example demonstrates the cyclic nature of the ***succ*** and ***pred*** operators. ■

2.6 Recursion

Recursion is one of the most powerful techniques discovered in computer science. A recursive routine is one which calls itself or in some way refers to itself. Thus a recursive relation is defined in terms of itself. The syntax for defining a recursive relation is as follows:

R **is** rel_expr;

In Relix a recursive relation is defined as a view. The keyword, ***is***, indicates that R is a view. The view is similar to a virtual attribute whose value is not computed until it is actualised in a relation There are two processes associated with views:

- Definition: the relation to the left of the ***is*** keyword is bound to the operation on the right of the ***is*** keyword. This definition is interpreted at the compile time
- Evaluation: the evaluation of the above definition is triggered. The evaluation happens when the view is executed at the run time using the current version of the relations in its definition.

Consider the statement, **R is S ijoin T**. A view in this case is a binding of the relation R with the operation (S **ijoin** T). The evaluation will take place every time R is referred to in the print or assignment statements.

Example 2-18. Given a relation,

```
Parent (Sr    Jr  )
       Dave  John
       John  Dima
       Dima  Alex
```

find all the groups of relatives in the `Parent` relation.

Ancestor **is** Parent **ujoin** (Parent [Jr **icomp** Sr] Ancestor);

As a result of this recursive process, `Ancestor` relation will contain six tuples. Issue a print command to evaluate the `Ancestor` view.

pr!!Ancestor

```
Ancestor (Sr    Jr  )
         Dave  John
         John  Dima
         Dima  Alex
         Dave  Dima
         Dave  Alex
         John  Alex
```

■

2.7 Metacode and Metadata

We can write Relix routines which interpret other Relix statements. These routines will be referred to as metacode routines.

A relational database consists of data and metadata. Data is represented by relations such as `Person`, `Stud_info`, etc. described above. Metadata is the data that describes or helps to interpret other data [Day85]. In a relational database metadata includes: the names of relations and their corresponding attributes; types and domains of attributes; physical storage and access paths for relations, etc.

Example 2-19

The `Requirements` relation specifies the criteria for belonging to each category of students at the university. Here the attributes store the names of the relations, `Grad_Students`, `Ugrad_Students`, `Students`, `Grad`, and `Ugrad` (see below).

```
Requirements (Category       Criterion1 Criterion2)
              Grad_Students   Students   Grad
              Ugrad_Students  Students   Ugrad
```

Metadata can be manipulated by domain algebra. The Requirements relation can be used to determine the complete information on people belonging to each category. To do this, we join the relations in the Criterion1 and Criterion2 fields.

let new_rel **be** Category **cat** " ← " **cat** Criterion1 **cat** " ijoin " **cat** Criterion2 **cat** ",";
Combine ← [new_rel] **in** Requirements;

As a result, the metadata stored in the relation Combine will be in the form of queries.

```
Combine ( new_rel                                    )
        Grad_Students ← Students ijoin Grad;
        Ugrad_Students ← Students ijoin Ugrad;
```

■

2.7.1 Stmt metadata

Relix can execute the contents of the attribute of a relation only if the attribute is defined to be of type ***stmt***. If that attribute is defined to be of type ***strg***, it can be cast to stmt type. During this process no changes to the value of the attribute occur. A ***stmt*** keyword preceding the name of the attribute signals to the system that the following attribute is to be treated as an executable statement

Example 2-20 Execution of the queries stored in the Combine relation in Example 2-13

let i **be** (**stmt** new_rel);
Res ← [i] **in** Combine;
Res;

Given the following relations,

```
Students          Grad             Ugrad
(Name Stud#)      (Stud# Year)     (Stud# Credits)
 Pete  123         123   2          234    30
 Kate  234         345   1          456    15
 Jake  345
 Vera  456
```

Two relations will be produced as a result of the actualization of the Res relation

```
Grad_Students            Ugrad_Students
(Name Stud# Year)        (Name Stud# Credits)
 Pete  123   2            Kate  234    30
 Jake  345   1            Vera  456    15
```

■

Chapter 3

Attribute Inheritance

The notion of inheritance is absent in conventional database systems. In this chapter we are proposing a relational model of classes, class hierarchy, and inheritance of attributes. The declaration of the inheritance hierarchy can be implemented by metacode in Relix, but we suggest a shorthand syntax which assumes that implementation.

In a relational data model, a class is represented by a relation where the attributes of well-defined types are the instance variables of that class, and the instances of that class are its tuples. (Some instances might be represented by more than one tuple). An instance object contains some general features (attributes of the superclass relation) and some specialized information (attributes of a subclass relation). Thus an instance object is spread across many relations of its inheritance hierarchy

In the following discussion we are going to concentrate on simple inheritance, where each class has at most one superclass

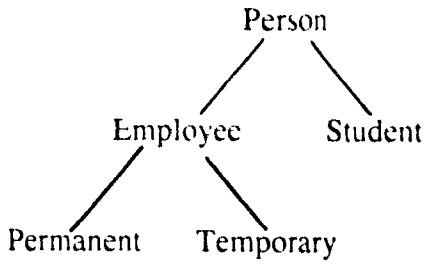
After introducing the relational representation of the inheritance hierarchy, this chapter will discuss the modifications to the basic relational operations (projection, selection, join and update) that are necessary to take advantage of the attribute inheritance. For details of each of the implementation algorithms discussed in this chapter, see Appendix A.

3.1 Representation of Inheritance Hierarchy

The first step in the implementation of inheritance is to determine in what format the representation of the class hierarchy will be stored in a relational database. A simple approach is to have a system meta-relation `.Hierarchy(.Subclass,.Superclass)` in which each tuple represents an *isa* relationship between two classes.

Example 3-1. Representation of the Person hierarchy.

Let us consider the following hierarchy:



There are four *isa* relationships in this hierarchy, so we can expect four tuples in the `.Hierarchy` relation.

```
.Hierarchy (.Subclass,.Superclass)
           Employee  Person
           Student   Person
           Permanent Employee
           Temporary Employee
```

■

As was mentioned above, subclasses inherit all the attributes of their superclasses. In a relational database model, that means that the attributes of the superclass are attributes of the subclass. To accomplish this, there must be a way of joining relations in the hierarchy, i.e. there must be a common field which links different levels of a hierarchy. An ID field serves this purpose.

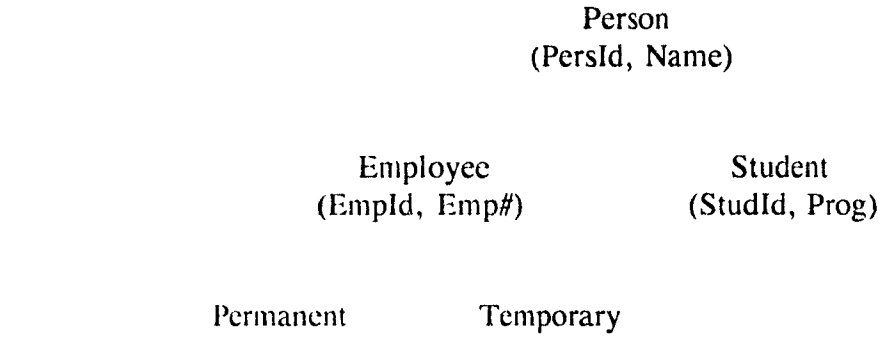
3.2 Object Identifier

In most commercially-available OODBs, objects are referenced by a pointer to their physical address in the memory. In a relational data model, the concept of pointers does not exist. The philosophy of the relational approach is to keep all the attributes and relations visible to the programmer: there are no hidden structures. Thus the identity field should be an attribute of each class relation. This approach was taken by POSTGRES [Ston91], which is an extension of a relational database system.

The object identifier (ID) is a system-defined surrogate which is generated during the creation of the class and is included as an attribute of that class. Since all the attributes are visible, ID can be used in all the relational operations.

The name of the ID attribute is user-defined. When user-defined names represent identity, we see a mix of addressability and identity. Addressability is external to an object. Its purpose is to provide a way to access an object within a particular environment and is therefore environment dependent. Identity is internal to an object. Its purpose is to provide a way to represent the individuality of an object independently of how it is accessed [Khos86]. Having user defined names for the ID field means that it is necessary to include the name of the ID field in the system meta-relation.

Example 3-2. For the hierarchy



the new format of the meta-relation is

```
.Hierarchy   (.Subclass, .SubId, .Superclass,.SuperId)
               Employee   EmpId   Person      PersId
               Student    StudId  Person      PersId
               Permanent  PermId  Employee    EmpId
               Temporary  TempId  Employee    EmpId
```

■

This meta-relation is now ready to be used. It specifies the superclass of each class of the hierarchy. It also gives the name of the ID field in each class, thus letting the inheritance mechanism know on which attributes to join subclasses with their superclasses. Later in this chapter we will discuss how metacode is used to implement the inheritance mechanism.

3.3 Declaration of Inheritance Hierarchy

A relation is considered independent of other relations unless it belongs to a hierarchy. New relations can be included in a hierarchy by specifying their superclass and

the names of ID fields in both subclass and superclass relations. The shorthand syntax for including a Subclass under the Superclass fits nicely into Relix philosophy. It will be referred to as an **inherit statement**

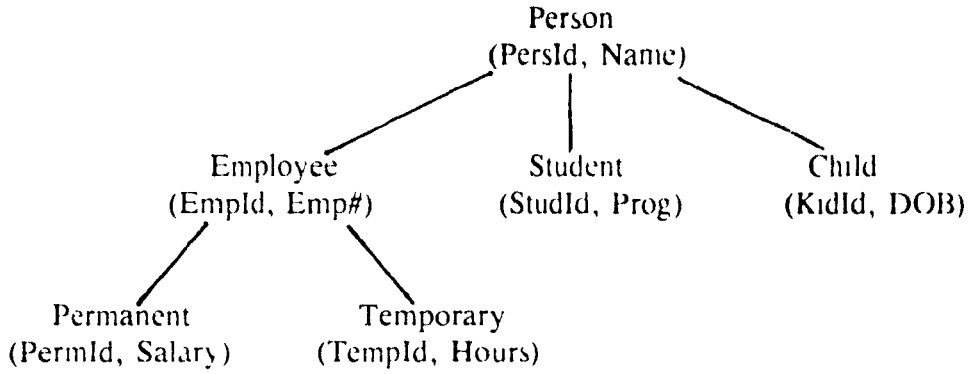
Subclass [SubId **isa** SuperId] Superclass

Example 3-3.

Include the relation `Child` into the ISA hierarchy under `Person`.

Child [KidId **isa** PersId] Person

The hierarchy of classes will be changed by including Child under the Person.



A new tuple will be added to the Hierarchy meta relation

```
.Hierarchy    (.Subclass,  .SubId,  .Superclass, .SuperId)
                Employee    EmpId    Person       PersId
                Student     StudId   Person       PersId
                Permanent   PermId   Employee     EmpId
                Temporary   TempId   Employee     EmpId
                Child       KidId    Person       PersId
```

■

If the names of the ID fields are not specified in the inheritance statement, a default field name "Id" is assumed. So an expression "Dog **isa** Mammal" translates into a tuple {Dog,Id,Mammal,Id} in the `.Hierarchy` relation

The semantics of the ***inherit statement*** is designed so as to disallow the insertion of relations between the existing classes In this way the specialization chain will not be broken. For example, including a class `Adults` as a subclass of `Person` and a superclass of `Employee` is not permitted

In all of the OODBs, the declaration of inheritance is done at the time of class definition, when no instances of that class exist. We will adopt the same approach and allow

only empty subclass relations to be used in the inherit statement. There is another reason why a relation can be placed into the ISA hierarchy only when that relation is empty. Since the ID field is generated by the system, the system should know at the time of creation of tuples of the relation whether to generate the values for that field or not.

At any time a class can be removed from the hierarchy, because it does not need to inherit either the attributes or the behavior from its superclasses. The same format of the inherit statement is used with the name of the relation on the left hand side of the ISA keyword and a reserved word ROOT on the right hand side. So to take the relation Permanent out of the hierarchy, we write

Permanent isa ROOT

When a class R is removed from the hierarchy, the attributes that could be referenced from the superclasses of R through the ISA links become inaccessible. The programmer has to keep in mind that some operations which were legal before the removal, become illegal after the removal

The procedure of removal of the relation from the hierarchy does not alter the contents of that relation This procedure breaks the link between it and its superclass by deleting the tuple representing that link from the Hierarchy relation. If the relation is a leaf class, the result of its removal from the hierarchy makes it an independent class. If the relation is an intermediate class, its removal from the hierarchy will make the branch rooted at it into a separate hierarchy

This might be useful in splitting a deep hierarchy tree. The user might need to work with different parts of the hierarchy in different stages of his program. For the purpose of simplification and effectiveness of join operations (fewer attributes in the intermediate relations, fewer relations to join) the user may choose to break the tree into several shorter subtrees This would be done in such a way that each subtree would contain all the necessary relations for that stage of the program.

Example 3-4. Spliting of class hierarchy.

A good example of such a situation is a highly bureaucratic organization with many levels of management Each manager is a subordinate of his higher-level manager and at the same time he has his own subordinates. At the bottom of the hierarchy are the unionized employees For the purpose of project management it is necessary to know the chain of command of each employee The union is only interested in the employee and his direct supervisor when the union handles complaints So for handling employee complaints, the union needs only the last two levels of the organizational hierarchy.

■

Since in a simple-inheritance system each subclass has only one superclass, there is a unique tuple for each subclass in the `.Hierarchy` relation. Therefore, when removing a leaf relation from the hierarchy of classes, the tuple corresponding to that relation in the `.Hierarchy` meta-relation is easy to locate

Example 3-5. Removal of a leaf class from the class hierarchy.

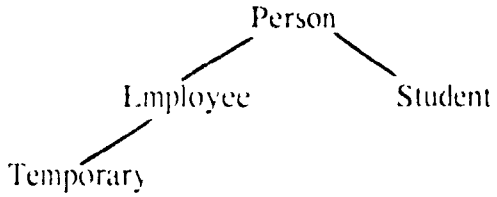
Remove the relation `Permanent` from the `Person` hierarchy (Example 3-2).

Permanent isa ROOT;

The tuple {Permanent,PermId,Employee,EmpId} will be deleted from the Hierarchy relation

```
.Hierarchy (.Subclass, .SubId, .Superclass,.SuperId)
            Student    StudId  Person       PersId
            Employee   EmpId   Person       PersId
            Temporary  TempId  Employee     EmpId
```

And graphically the Person hierarchy will look like this:



■

Example 3-6. Removal of a non-leaf class from the class hierarchy.

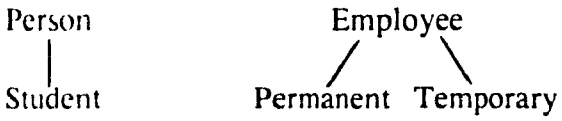
Remove the relation `Employee` from the `Person` hierarchy (Example 3-2).

Employee isa ROOT

The tuple {Employee,EmpId,Person,PersId} is deleted from the `.Hierarchy` meta-relation

```
.Hierarchy (.Subclass,.SubId, .Superclass,.SuperId)
            Student   StudId  Person      Persid
            Permanent PermId  Employee    EmpId
            Temporary TempId  Employee    EmpId
```

The branch rooted at `Employee` is made into an independent hierarchy. We now have two hierarchies: `Person` and `Employee`.



■

This syntax is suitable for the declaration of multiple inheritance. We can say that in the Person hierarchy in Example 3-2, Temporary employees are at the same time Students. The investigation and implementation of multiple inheritance is left for future research. When multiple inheritance is not supported by the language, care should be taken to ensure that every class has only one superclass.

3.4 Algorithm for Implementation of the Inherit Statement

Given an inheritance statement,

Child [CId **isa** PId] Parent

Parse the inherit statement and initialize the following scalars:

Child - the left most relation name;

CId - if there are square brackets, the attribute name after "[" and before ***isa*** keyword, else Id;

PId - if there are square brackets, the attribute name after the ***isa*** keyword and before "]", else Id;

Parent - the right most relation name.

Proceed as follows:

If Parent = "ROOT" then

Delete a tuple from the `.Hierarchy` meta-relation where `.Subclass = Child`

else

if the `Child` relation is empty then

add a tuple {`Child,CId,Parent,PId`} to the `.Hierarchy` meta-relation

else

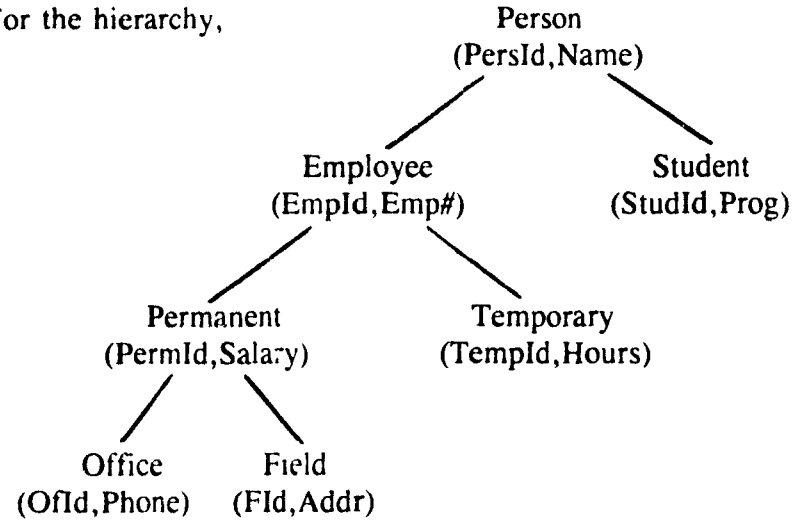
Error - attempt was made to include a non-empty relation.

Section A.1 of Appendix A gives a more detailed description of this process

3.5 Projection and Selection

Example 3-7. Selection and projection of inherited attributes.

For the hierarchy,



evaluate an expression

[Emp#] **where** Salary = 25 **in** Office

Note: the attributes `Emp#` and `Salary` do not belong to the relation `office`. They are found in the relations `Employee` and `Permanent` above `office` in the hierarchy. The system should search for these attributes in the superclasses of `office`.

■

Two approaches can be taken to fulfil the request to project an attribute from a relation higher up in the hierarchy. They vary in their efficiency.

The first approach is naive, but intuitive and straightforward. When an attribute being projected is not defined in the relation from which it is projected, we can join all the relations in the branch of the hierarchy tree to which that relation belongs, and then project the requested attribute.

Example 3-7a. Evaluation of the expression in Example 3-7.

The system generates an expression which contains the join of all relations in the hierarchy from `Office` to the root class `Person`:

(((Office [OfId **ijoin** PermId] Permanent) [PermId **ijoin** EmpId] Employee)
[EmpId **ijoin** PersId] Person),

and then evaluates the original expression by replacing `Office` by the above expression

[Emp#] **where** Salary = 25 **in** (((Office [OfId **ijoin** PermId] Permanent)
[PermId **ijoin** EmpId] Employee) [EmpId **ijoin** PersId] Person),

■

The second approach is to reduce the number of relations in the join by ignoring all the classes which do not contain any attributes referenced in the expression. This is called a **minimum join approach**. It is important to mention that minimum join should always include the original relation even if that relation does not contain any attributes referenced in the projection or selection list. This will ensure that only those objects whose specialization is specified by the user (i.e., only those `Employees` that have an `Office`) will be considered for the projection and selection operations. The `.Hierarchy` table is used to determine on which attributes these relations are to be joined.

Example 3-7b. Minimum Join

Only relations `Permanent` and `Employee` need to be joined with `Office`. Our original expression is evaluated on the generated join expression.

[Emp#] **where** Salary = 25 **in** ((Office [OfId **ijoin** PermId] Permanent)
[PermId **ijoin** EmpId] Employee);

■

It might be argued that pointers can be much faster than joins, but only for connecting **single** objects. Our approach deals always with sets of objects --classes and their subsets-- and because of possible large size of these sets, it is better to use joins.

3.6 Algorithm for Implementation of the Minimum Join Approach to Project and Select Operations

Given a statement,

R1 ← [projection] **where** selection-condition **in** R;

determine if the input expression can be evaluated in the current implementation of Relix (i.e., all the attributes in the projection list and the selection-condition are defined on R). If the evaluation is not possible, then proceed as follows:

1. Determine all the superclasses of R on which the attributes referenced in the projection list and the selection condition are defined. Generate a join expression in which these superclasses are joined with R
2. An executable expression is generated from the input expression by replacing R with the join expression of step 1

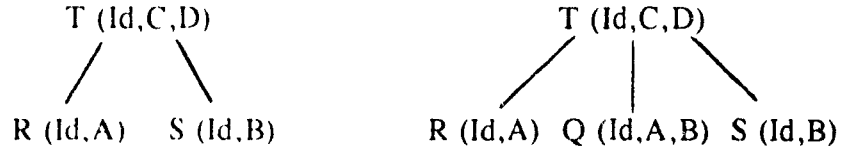
   R1 ← [proj] **where** selection-cond **in** (join expression);

Section A 2 of Appendix A presents a more detailed description of this process.

3.7 Join

The two operands of the join operation can be independent relations, relations from the same hierarchy, relations belonging to possibly different hierarchies, or a combination of above types. Let us consider several special cases where Q ← R **ijoin** S.

1. R and S belong to the same hierarchy and they have the same parent T.

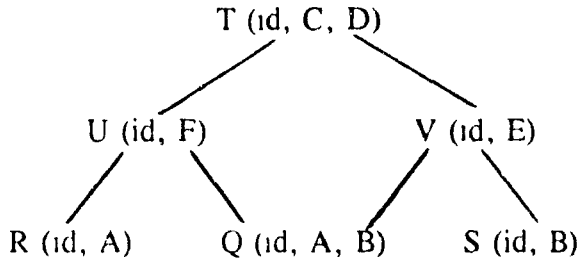


a) b)

Figure 3.1

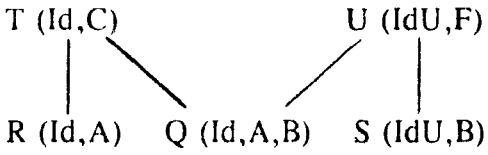
In order to avoid duplicating attributes of T, Q can be attached under T, thus inheriting its attributes (Fig. 3.1b).

2. R and S belong to different branches of the same hierarchy. We want Q to inherit from two relations: the parent of R and the parent of S. This is a case of multiple inheritance

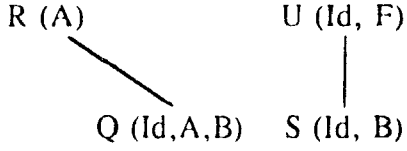


Note: one can think of this case of multiple inheritance as if, for the time of join, R and S were treated as independent relations. Thus Q will have all the combinations of attributes A and B. When the join is complete, we remember to include Q under the parents of both R and S, because Q already has all the necessary information about the attributes of R and S relations, but not about the inherited attributes

3. R and S belong to different hierarchies.



4. R is an independent relation and S belongs to a hierarchy.



There are several disadvantages of making Q inherit from the superclasses of R and S:

- the above arguments and suggested treatment might not be equally valid for all the μ- and σ-joins

  μ-joins. In the case 4 above, R **djoin** S will produce an independent Q because it will not have the IDs that are in the hierarchy to which S belongs.

  σ-joins. Any σ-join on the IDs will produce the relation Q, which will not have the ID field in it, and thus Q will be independent.

- a different treatment of each special case implies a complicated implementation of the join statement

Considering these important disadvantages, it is better to create Q as an independent relation containing all the attributes of the joining relations R and S This will allow a completely general implementation of all of the join statements A preliminary routine will expand all the relations belonging to a hierarchy into independent relations and will pass those independent relations to the current implementation of the join statement. Thus a maximum reusability of the existing code will be achieved.

Example 3-8 Join of Relations in Different Branches of the Class Hierarchy.

In the expression,

[Name,Salary,Stud#] **in** Permanent [PermId **ujoin** StudId] Student

relations `Permanent` and `Student` belong to different branches of the same hierarchy. The attribute `Name` is not defined on any of the relations in the join. The minimum joins of the relations in the `Permanent` branch and the `Student` branch of the `Person` hierarchy are

- for `Permanent` Permanent [PermId **ijoin** PersId] Person
- for `Student` Student [StudId **ijoin** PersId] Person.

We replace the original join expression with the new one, which uses the minimum join expressions above

[Name,Salary,Stud#] **in** ((Permanent [PermId **ijoin** PersId] Person)
[PermId ujoin StudId] (Student [StudId **ijoin** PersId] Person));

■

3.8 Algorithm for Implementation of Join

Given a join expression,

[projection] **where** selection_condition **in** S [attr$_s$ join_symbol attr$_t$] T

determine if the input expression can be evaluated in the current implementation of Relix (i.e., all the attributes in the projection list and the selection-condition are defined on the relations S and T).

If the evaluation is not possible, then proceed as follows

1. Determine all the superclasses of the relations participating in the join on which the attributes in the projection list are defined. For each of these input relations generate a join expression which joins that relation with its superclasses.
2. Expand the original join in the ***in*** clause by putting together the join expressions of step 1 in the same order as their base relations appeared in the original join, and separating these expressions by the corresponding join symbol from the join_symbol table.

Section A.3 of Appendix A presents a more detailed description of this process.

3.9 Update

A subclass relation may override the attributes in its ancestor relations. There are two types of overrides: value and meaning.

1. Value override - when a system receives a request to update an inherited attribute, this attribute is updated in the relation in which it is defined, and not in the relation in which it is inherited.

Example 3-9. Value Override.

Consider three relations below which belong to the `Person` hierarchy (Example 3.7)

```
Person          Employee          Permanent
(Id, Name)      (Id, Position)    (Id, Salary)
 1   Mike        1   Manager       1    35
 2   Ann         2   Clerk         2    20
 3   Pete
```

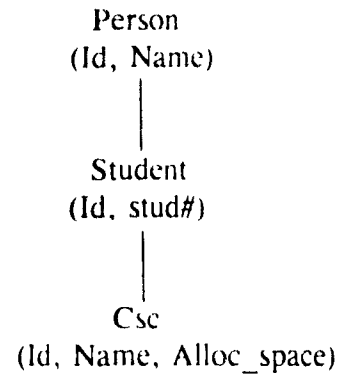
If the user wants to change the Name of a Permanent Employee who earns 20 to Kate, then the value of the attribute Name should be changed in its original relation Person. This ensures that the change is seen by the offspring of Person.

■

2. Meaning override - if an attribute defined in a superclass takes on a different meaning in a subclass, that attribute may be redefined in the subclass. When the system receives a request to update the redefined attribute, only the subclass is affected.

Example 3-10. Meaning Override

Consider the following hierarchy



where for a computer science student the Name attribute represents the user name on the computer and not the real name. Given: Pete's student# is 1234 and he has an account on "bart" machine

```
Person          Student          CSc
(Id, Name)      (Id, stud#)      (Id, Name,    Alloc_space)
 3   Pete        3   1234         3   Pete@bart   10Mb
```

If Pete's account gets transfered to "homer" machine, his login name can be changed without affecting his real name:

update CSc **change** Name ← "Pete@homer";

```
Person          Student          CSc
(Id, Name)      (Id, stud#)      (Id, Name,    Alloc_space)
 3   Pete        3   1234         3   Pete@homer  10Mb
```

■

3.9.1 Updating an ID field

Of special consideration is a question of updating an Id field. Since an ID field is a visible attribute and it plays a special role in linking classes, we are faced with a dilemma should the updates be allowed on it? Several options can be considered.

1. Allow updates of ID only in the following cases
   - addition of new IDs to the root of the hierarchy,
   - deletion of old IDs from the leafs of the hierarchy tree,
2. Allow updates everywhere, before a physical update takes place, verify the validity of it (make sure that no children are made orphans),
3. Allow unconditional additions and deletions of IDs anywhere in the hierarchy and fill in automatically the missing information with null values

Here we discuss each option in detail

1. Restrict updates of the ID field. Allow only additions of new IDs at the root and deletions of IDs at the leafs of the hierarchy tree.

Example 3-11.

Consider the following relations which belong to the hierarchy in the Example 3-7

Person (Id,Name)		Employee (Id,Emp#)		Permanent (Id, Salary)	
1	Mike	1	123	1	25
2	Ann	2	345		
3	Pete				

The rest of the relations are empty

The only operations allowed are to add tuples with new IDs to Person and to delete tuples from the leaf classes Field, Office, Temporary, and Student. In this case deletion of the employee with Id = 2 is not allowed, since Employee is not a leaf class. Logically, we should be allowed to delete it since it does not have children, i.e. there are no tuples in Permanent nor in Temporary with Id = 2. So in reality an instance of Employee with Id = 2 is the leaf of this branch.

Tuples with the new values of ID attribute can only be added to the root class. Below we show what might happen otherwise. If we add a tuple {4, 567} to the Employee relation, Employee 4 would not have the corresponding tuple in the Person

relation

Person	(Id, Name)	Employee	(Id, Emp#)	Permanent	(Id, Salary)
	1 Mike		1 123		1 25
	2 Ann		2 345		
	3 Pete		4 567		

■

2 Allow all updates as long as they make sense.

This approach does not put any restrictions on the syntax of the update, assignment and declaration statements But the statement should be semantically correct, i.e.

- in the case of a delete, the system should verify that there are no tuples in the subclass relation with the same ID (orphans are not allowed)
- in the case of an add, the system should make sure that a parent with the same ID already exists.

This approach gives the user a lot of freedom and flexibility in manipulating data

3 Allow all updates

Here the requirements of the previous option are completely relaxed. The additions and deletions of IDs are allowed at any level in the hierarchy. In the case of addition, the requested tuple is created, then the parent of the updated relation is found, and the existence of the added ID in the parent relation is verified If it is not found, a tuple with that ID is added to this parent relation and all other attributes are initialized to Null. These attributes can be updated at a later time

This is useful when a user is only interested in the attributes of the leaf classes and will (if necessary) input the missing information later. This might produce unwanted results when manipulating an incomplete relation (where Null values have not been changed to meaningful ones)

In the case of deletion, all the relations in the hierarchy under the class in which the ID is eliminated are cleared of the tuples with that ID.

This approach supports the concept of generalization, where we build the hierarchy from the most specific object to a more general one

Among the existing OODBs, POSTGRES supports the second approach.

To this point, we dealt with additions and deletions of tuples in a class. The next logical question is, what happens when an ID field is changed in the updating process?

A change of an attribute consists of two parts deleting the tuples with that attribute and inserting a tuple with the new value of that attribute. All the other attributes retain their original value

Example 3-12 Break of the generalization chain as a result of the change to the ID.

Given three relations that belong to the hierarchy in Example 3-7,

```
Person              Employee          Permanent
(PersId,Name)       (EmpId,Emp#)      (PermId,Salary)
   1     Pete          1    123          1      25
   2     Mike          2    345          2      35
   3     Kate          3    567
   4     Nick
```

update Permanent **change** Name ← "Ann", PermId ← 3,

If this statement is legal, then first, all the tuples in `Person` which have a corresponding tuple in `Permanent` will be identified These tuples will be updated and the name "Ann" will replace the previous names Then all the tuples in `Permanent` will have their `PermId` changed to 3 As a result we should see the following information

```
Person              Employee          Permanent
(PersId,Name)       (EmpId,Emp#)      (PermId,Salary)
   1     Ann           1    123          3      25
   2     Ann           2    345          3      35
   3     Kate          3    567
   4     Nick
```

Syntactically there is nothing wrong with this result since no orphans resulted from this operation Nevertheless the generalization chain has been changed dramatically. To avoid such a situation, a constraint might be enforced where the value of ID field can be changed only if its parent is specified

■

Advantages to treating an ID field in a special restricted fashion:

- ID plays a special role as a link between two relations in the hierarchy. Once that link is established, it should not be allowed to be changed, since any change to its value alters the established generalization path
- protects the user from mistakenly altering the generalization path.

Disadvantages to treating an ID field in a restricted fashion:

- the user is seriously limited in the desired manipulation of accessible fields in all relations,
- the user is not prevented from altering the generalization path, since the user can delete the leaf relations and recreate them with different ID values;
- the implementation of the update statement is complicated.

There are no mechanisms in Relix to protect fully any attribute from being changed. Even if strong protection is provided, the user can always go around it and achieve what he wants. This leads to the conclusion that there should be no restrictions on the use of ID field in the update statement. The user should carry full responsibility for the results of his update statements.

3.9.2 Implementation of the Update Statement

The update statement in Object-Oriented Relix may contain attributes from different levels of the inheritance hierarchy. As long as these attributes are found in the relations belonging to the same branch of the hierarchy tree, the statement is valid. In order to achieve maximum utilization of existing routines, the update statement (which contains attributes not found in the specified relation) will be transformed into two or more update statements, each operating only on the attributes in the specified relation.

Example 3-13. Treatment of the Update Statement with Inherited Attributes in the Change Clause.

The statement

update Permanent **change** Name ← "Ann", PermId ← 3,

will be transformed into two statements to give the above interpretation

update Person **change** Name ← "Ann" **using** ((Permanent [Permid **ijoin** EmpId] Employee) [EmpId **ijoin** PersId] Person),

update Permanent **change** Permid ← 3;

■

The previous example dealt with the ***change*** clause. Next we discuss the implementation of the ***add*** and ***delete*** clauses of update expressions. The operation of addition of tuples to a relation in the class hierarchy has two interpretations:

1. New tuples specialize the existing objects with the specified object ID because the new attributes are added to the classes lower down in the hierarchy,
2. New tuples define new object(s) which did not exist in the hierarchy by providing attributes in all the relations over which new objects are defined.

The first interpretation means that the relation in the ***add*** clause of the update expression has all the attributes of the relations in the hierarchy that need to be updated. The second interpretation means that the attributes of all the relations in the hierarchy, including the relation being updated and those above it, are supplied by the relation in the ***add*** clause. In both cases it is important that the set of attributes in the ***add*** clause should be a union of all the attributes in the relations affected by this update expression.

3.9.2.1 Adding New Objects to the Hierarchy

Example 3-14.

Given the following relations of the `Person` hierarchy of the Example 3-7,

```
Person              Employee          Temporary
(PersId,Name)       (EmpId,Emp#)      (TempId,Hours)
   1     Pete          1    123
   2     Mike          2    345
   3     Kate          3    567          3      27
   4     Nick
```

suppose a user wants to add the tuples of the relation NewEmployees

```
NewEmployees (TempId,Name, Emp#, Hours)
                5    Fred  987    45
                6    Ken   888    50
```

to the class Temporary. Note: NewEmployees contains all the attributes of the class Temporary(TempId,Hours), including the inherited ones (Name,Emp#). User's statement

update Temporary **add** NewEmployees;

will be transformed into three statements

update Person **add** ([TempId,Name] **in** NewEmployees);

update Employee **add** ([TempId, Emp#] **in** NewEmployees);

update Temporary **add** ([TempId, Hours] **in** NewEmployees);

After the execution of these three statements, the updated relations will have the following tuples (new tuples are in bold)

```
Person                Employee          Temporary
(PersId,Name)         (EmpId,Emp#)      (TempId,Hours)
   1    Pete             1   123
   2    Mike             2   345
   3    Kate             3   567          3     27
   4    Nick
   5    Fred             5   987          5     45
   6    Ken              6   888          6     50
```

■

There are still problems with this breakdown. The relations Person and Employee do not have the attribute PermId When climbing the hierarchy to find superclass relations to be updated, the ID of each superclass should be associated with the ID of the class in the update expression

3.9.2.2 Specializing Objects in the Hierarchy

Example 3-15

It is also possible to add a relation

```
FirstJob (TempId,Emp#,Hours)
              4   479   15
```

to the class Temporary The user's statement

update Temporary **add** FirstJob;

will be transformed into three statements

let EmpId **be** TempId,

update Employee **add** ([EmpId, Emp#] **in** FirstJob);

update Temporary **add** ([TempId, Hours] **in** FirstJob).

The changes are illustrated on the relations given at the beginning of Example 3-14.

Person (PersId, Name)		Employee (EmpId, Emp#)		Temporary (TempId, Hours)	
1	Pete	1	123		
2	Mike	2	345		
3	Kate	3	567	3	27
4	Nick	4	479	4	15

Adding `FirstJob` to the `Temporary` would have been invalid if the `Employee` relation was defined on three attributes (`Id,Emp#,Dept`), because the implementation of the ***add*** clause in Relix requires that the attributes in the add clause be the same as the attributes in the relation being updated

■

Before the update is completed, the system checks that no orphans are created and that all the attributes have values

A ***delete*** clause is interpreted in a similar way.

3.10 Algorithm for Implementation of Update Statement

The ***change*** clause of the Update statement is interpreted differently from the ***add/delete*** clause. Their implementation is described separately

1. *Change* clause.

Given a general format of an update statement with the ***change*** clause,

update R **change** $a_1 \leftarrow value_1$, $a_2 \leftarrow value_2$,...,$a_n \leftarrow value_n$ **using** Expr,

determine if the input expression can be evaluated in the current implementation of Relix (i.e., the fields $value_1$ through $value_n$ are defined on the Expr)

If the evaluation is not possible, then proceed as follows:

1. Find the superclasses of the relation in the ***using*** clause and choose those ones on which the attributes `value`$_1$ to `value`$_n$ as well as the attributes referenced in the Expr are defined. Construct a join expression from these classes.
2. Evaluate the join expression from step 1 and place the result in a relation `Temp_Rel`.
3. Determine which classes of the branch whose leaf is R are affected by the update statement, i.e. which classes define the attributes being updated.
4. For each class selected in step 3 generate the list of assignments which will make up the ***change*** clause
5. For each class selected in step 3 generate an executable update statement by concatenating "update ", Superclass, "change ", list of assignments, "using `Temp_Rel`". Execute the generated statements one after another in the same order as the attributes in the ***change*** clause of the input update statement.

2. ***Add*** and ***delete*** clauses.

Given the general format of the update statement with ***add*** or ***delete*** clause,

update R { **add** / **delete** } ([attr$_1$,...attr$_n$] **where** select_cond **in** S);

determine if the input expression can be evaluated in the current implementation of Relix (i.e., the fields `value`$_1$ through `value`$_n$ are defined on the Expr)

If the evaluation is not possible, then proceed as follows:

1. Find the superclasses of the relation being updated and among them choose those classes which define the attributes `attr`$_1$ to `attr`$_n$.
2. For each class selected in step 1, generate the projection list of attributes. This list will be included in the ***add/delete*** clause of the generated update statement for that class
3. For each class selected in step 1, generate an executable update statement by concatenating "update ", Superclass, type of clause "(", projection list, " in ", S. Execute these generated update statements.

Section A.4 of Appendix A presents a more detailed description of these processes.

Chapter 4

Method Inheritance

4.1 Functions and Procedures in the Object-Oriented Relational Database Language Relix

Functions and procedures (methods) provide a convenient way to encapsulate and to abstract computations. Presently Relix does not have an implementation of methods. Here we discuss how methods can be applied to relations and how to associate the method with a class.

The fact that a function returns only one value allows us to assign a function to a virtual attribute in a domain algebra statement. When this virtual attribute is projected from a relation, it is calculated for each tuple of that relation by the specified function. The arguments of a function can be scalars or attributes of the relation to which that function is applied.

Example 4-1. Function Declaration.

Given a relation Person and a function `Calc_Age`,

```
Person (Name YOB )
       Mike 1941
       Pete 1961
```

```
function Calc_Age (birth_year, curr_date);
    return (curr_year − birth_year);
```

calculate the age of each person using a function `Calc_Age`:

let curr_date **be** 1991;

let age **be** Calc_Age (YOB, curr_date);

R ← [Name, age] **in** Person;

```
R (Name age)
   Mike  50
   Pete  30
```

■

The difference between a procedure and a function is that a function returns one value, while a procedure returns several values. Therefore, when applied to a relation, a function returns one virtual attribute of that relation and a procedure returns several virtual attributes. Since more than one attribute is returned by a procedure, it cannot be assigned to one attribute in a domain algebra statement in the same way as a function is.

To overcome this problem, a procedure definition needs to have a set of input arguments and a set of output arguments

Example 4-2 Procedure Declaration

Procedure Adjust_Marks (**in:** midterm,final,adjustment; **out:** new_mid,new_fin);

new_mid ← midterm + adjustment;

new_fin ← final + adjustment;

■

The format of a procedure call in the domain algebra of Relix would be:

let Proc_name (**in:**I_1,I_2..,I_n; **out:**O_1,O_2..,O_m);

where I_1 I_n are the input attributes, and O_1 .O_m are the output virtual attributes.

This domain algebra statement defines the output attributes O_1 to O_m as coming from the procedure `Proc_name`. The procedure will only be invoked when at least one output attribute is specified in the projection or selection operation. As with the functions, the procedure `Proc_name` will be performed on every tuple of the invoking relation.

Example 4-3. Procedure Invocation.

Given a relation `Students` below and the procedure `Adjust_Marks` (Example 4-2),

```
Students (Name midterm final)
          Fred     86     80
          Jack     74     78
          Lynn     92     90
```

calculate adjusted midterm and final marks for each student:

let adjustment **be** 5;

let Adjust_Marks (**in:**midterm,final,adjustment; **out:**new_mid,new_fin);

Students_revised ← [Name, new_mid,new_fin] **in** Students;

```
Students_revised (Name new_mid new_fin)
                  Fred     91      85
                  Jack     79      83
                  Lynn     97      95
```

■

4.2 Generic and Class Associated Methods

The number of relations to which a method may be applied is controlled by the way that method is declared. There is an optional ***on*** clause which can be used for this purpose.

function Determine_nationality () **on** Person;

If the ***on*** clause is not used, then the method is generic (for example, function `Calc_age` in the Example 4-1). If the relation in the ***on*** clause is:

- not associated with any hierarchy, then the method can only be applied to that relation;
- associated with a hierarchy, then the method can be used with that relation and all its subclasses.

In other words, a method defined on a relation cannot be used with any other relation except with its subclasses. One of the advantages of using such a method is that no matter from where it is invoked, the attributes of the relation on which it is defined are always available. Methods associated with a relation can reference all the attributes of that relation without these attributes being passed in the parameter list. Since subclasses inherit all the attributes of its superclasses, a method defined on a superclass can be invoked on the subclass.

For a generic method, all the variables used in the body of the method should be passed as parameters. POSTGRES follows this approach.

4.3 Polymorphic Methods

A program may have several methods with the same name but declared on different relations. This can be useful when a different treatment is required for different relations, but the meaning of the result is the same. For example, to calculate the present value of a government bond we need to add the compounded interest to its base value, and to calculate the present value of a car we need to subtract the depreciation amount from its original price.

That is part of the concept of polymorphism. It also leads to late binding since at the compilation time the actual method name cannot be determined.

To implement this feature we need a system relation, `.Methods`, where this information will be stored. It has the following format.

```
.Methods ( .rel,          .meth          )
          Investment   Calc_Present_Value
          Vehicles     Calc_Present_Value
```

Every time a method is defined with an ***on*** clause, a tuple is added to this relation.

It is possible to override a superclass's method by defining a method with the same name on the subclass. A subclass can still access the superclass's method with the help of a keyword **super**. For example, a method `Training` defined for a `Person` class determines the number of years of postsecondary education. In the `Employee` class this method is redefined to determine the number of in-house training courses an employee has completed. To invoke the `Person`'s definition of `Training` on an `Employee`, we write,

```
let Education be super Training();
result ← [Education] in Employee;
```

When an invocation of a method M is requested by projecting virtual attributes defined by that method from a relation R, the system searches the `.Methods` relation for an indication that M is defined on R or one of the superclasses of R. If method M is invoked with the **super** keyword, then the search starts from an immediate superclass of R. If the

search is successful, it determines which definition of M to use. Otherwise the system assumes that the method is generic. An error occurs if this generic method is not defined.

Example 4-4. Mechanism to Determine the Applicability of Methods to Relations

Given the representation of a `Person` hierarchy and a function declared on a subclass of `Person`.

```
.Hierarchy (.Subclass,  .SubId,  .Superclass, .SuperId)
            Employee    EmpId    Person       PersId
            Permanent   PermId   Employee     EmpId
            Temporary   TempId   Employee     EmpId
```

function Calc_Seniority (curr_month) **on** Employee,

return (curr_month - start_month),

When the system receives the declaration of the `Calc Seniority` function, the following tuple is inserted into the `.Methods` relation

```
.Methods (.rel,       .meth          )
          Employee  Calc_Seniority
```

Apply `Calc_Seniority` method to the `Temporary` relation

let curr_month **be** 11;

let seniority **be** Calc_Seniority (curr_month),

Recent_Hires ← [Name, Emp#, seniority] **where** seniority < 2 **in** Temporary,

During the projection of the `seniority` attribute from the relation `Temporary`, the system realized that since this attribute is not defined on `Temporary` nor on its superclasses, it must be a virtual attribute returned by the function `Calc_Seniority`. The system checks the `.Methods` table to see if it can find the tuple {`Temporary`, `Calc_Seniority`}. Since this tuple is not found, the system consults the `.Hierarchy` table to locate the superclass of `Temporary`, and then looks for the tuple {`Employee`, `Calc_Seniority`}. The successful search provides the system with the information about which function to execute.

■

4.4 Algorithm for Implementation of Method Inheritance

If an input expression has method invocations, repeat the following steps for each independently evaluated subexpression that makes up the input expression:

- determine on which classes these invoked methods are defined;
- find all the superclasses of the relations on which these methods are invoked;
- verify that all the non-generic methods are defined either on the input relations, or on their superclasses. If a method is defined on both a subclass and a superclass, choose the subclass;
- determine if there are any ambiguous method invocations (i.e. some methods are defined on more than one branch of a class hierarchy).

Appendix B provides more details of this algorithm.

4.5 Method Invocation On Relational Expressions

Up to now we looked at simple expressions, like a projection from a relation, which did not involve any relational expression evaluation prior to invocation of the method. In general, a method defined on a relation can be invoked on any expression which contains that relation.

Example 4.5

Given two relations, `Employee(Id,Emp#)` and `Returned_Employees(Emp#, date_returned)`. `Employee(Id,Emp#)` belongs to the `Person` hierarchy in Example 4.4, and `Returned_Employees` is an independent relation.

The definition of the attribute `seniority` is found in the same example.

In the expression

[Id, seniority] **in** Employee **ijoin** Returned_Employees;

`calc_Seniority` is invoked on the intermediate relation that contains the result of the

evaluation of the join operation.

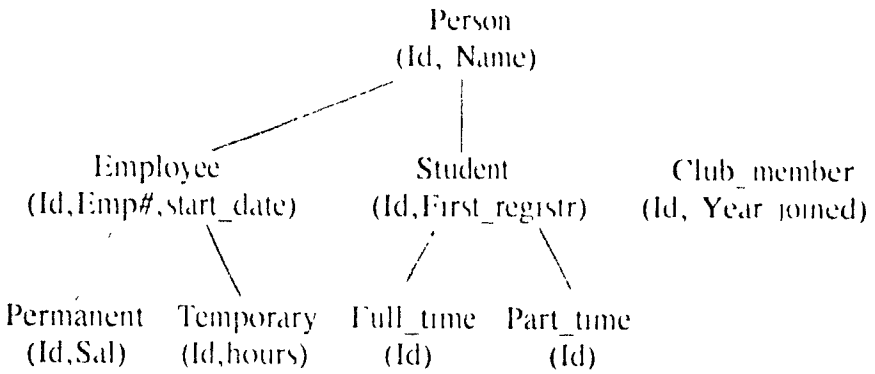
In determining the validity of this invocation, Calc_Seniority method is looked up in the .Methods relation. If it is found and its corresponding relation is Employee or Employee's superclass Person, or Returned_Employees, then the Calc_Seniority defined on that relation is invoked. In this example, the tuple {Employee,Calc_Seniority} of the .Methods relation tells the system to use the function Calc_Seniority defined on the Employee relation.

■

This approach suggests that if more than one relation in the relational expression has the same method defined on it or any of its superclasses, the ambiguity will arise. In order to avoid ambiguities, the method to be applied must be identified, for instance, by appropriate projection.

Example 4-6 Ambiguity Resolution.

Given an inheritance hierarchy and a different implementation of the function Calc_Seniority on several subclasses.



function Calc_Seniority (curr_year) **on** Employee

  return (curr_year − start_date);

function Calc_Seniority (curr_year) **on** Student

  return (curr_year − First_registr);

function Calc_Seniority (curr_year) **on** Club_member
  return (curr_year − Year_joined);

The definition of a virtual attribute `seniority` is the same as in the Example 4-4, i.e ,

  let seniority **be** Calc_Seniority (1991),

To find the number of years of membership in the club of every temporary employee who is doing part-time studies, we have to invoke the `Calc_Seniority` function on the `Club_member` class in order to avoid ambiguities.

```
let seniority be Calc_Seniority (1991);
Club_member_info ← [Name,seniority] in
        (([Id, seniority] in Club_member) ijoin Temporary ijoin Part_time);
```

Here `seniority` will have the value returned by the function `Calc_Seniority` defined on the `Club_member` relation

Special attention should be paid to the fact that the result of the evaluation of the relational expression on which a method is invoked should have all the attributes referenced in that method In this example, the method `Calc_Seniority` uses the attribute `Year_joined` which is defined on the `Club_member` class

On the other hand, to find the number of years of experience of those temporary employees who are studying part-time and are members of the club, a different projection is executed. In this example, the method `Calc_Seniority`, defined on `Employee`, is invoked because the intermediate expression contains the attribute `start_date` and not `First_registr` or `Year_joined`.

```
Temp_employee_info ← [Name,seniority] in
                (([Id] in Club_member) ijoin Temporary ijoin ([Id] in Part_time));
```

■

Chapter 5

Collection Hierarchy and Subobjects

The notion of the inheritance of attributes and methods, although powerful, cannot represent the IS-PART-OF relationship between objects that epatures the notion of an object being part of another object. Along with the IS_A relationship, the IS_PART_OF relationship is one of the fundamental data modelling concepts.

Many applications require the ability to define and manipulate a set of objects as a single logical entity for the purposes of semantic integrity. We define a **collective object** as a collection of references to other objects (possibly of different classes), which make up that collective object. Collective objects are grouped together into **collective classes**. For example, the class `Room` holds objects of classes `Couch`, `Table`, and `Chair`.

The **collection hierarchy** expresses the IS_PART_OF relationship between classes by listing the classes of objects referenced in each collective class. A particularly useful characteristic of collective classes is the capacity to broadcast a message to every **subobject** (i.e., object that is part of the collective object) of a collective class.

Example 5-1

To find the area of a `Room` that is occupied by furniture (the sum of the sizes of different furniture items in the `Room`), a message `Calc_size` is sent to the `Room` collective class, which in turn forwards that message to all the subobjects found in that room. Since subobjects belong to different classes, the method corresponding to the message sent may have different implementations in every class. For example, to calculate the size of the `Couch`, multiply its `Length` by its `Width`, to determine the size of the `Table`, perform the calculation that is appropriate for its shape, to calculate the size of the `Chair`, square its base.

■

Thus when a message is sent to a collective class, the message is interpreted by the collective class and by the individual subobjects. The mechanism of collection works in a similar way to the inheritance mechanism, except that in the collection hierarchy the search is done in a downward direction, whereas in the inheritance hierarchy it is done in the upward direction

5.1 Declaration of Collection Hierarchy

A collective class is defined by a HASA expression in a way similar to the declaration of subclasses in the inheritance hierarchy.

Example 5.2 Declaration of Collection Hierarchy

The following Relix code is used to define a collective class `Room`, which contains subobjects of classes `Couch`, `Table`, and `Chair`.

Room **hasa** Couch,

Room **hasa** Table,

Room **hasa** Chair;

■

The collection hierarchy is represented by a system meta-relation `.Collection (.collectClass,.subobjClass)` whose attributes `.collectClass` and `.subobjClass` store the names of relations When the system receives a declaration of a collective class, as in Example 5-2, it inserts a tuple with the name of the collective class (the word on the left of the HASA keyword) and the name of the class of the subobjects (the word on the right of the HASA keyword) into the `.Collection` relation.

Example 5-3

The `Room` collective class is represented by three tuples in the `.Collection` relation.

```
.Collection (.collectClass,.subobjClass)
                Room            Couch
                Room            Table
                Room            Chair
```

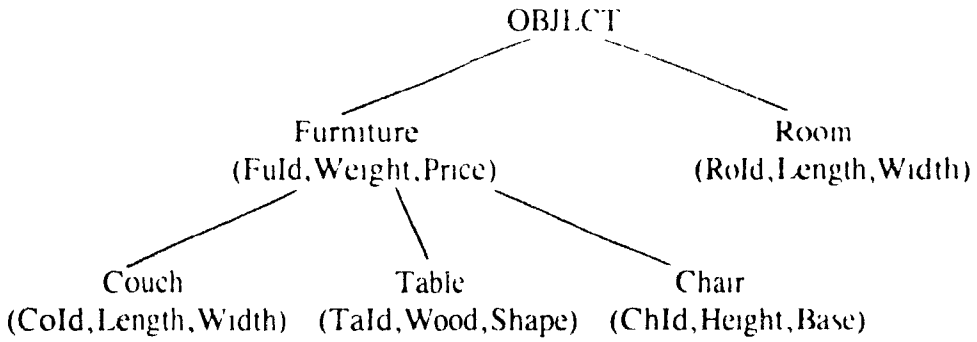
■

Another aspect of the representation of a collective class is how to include the actual subobjects in the collective object We use a relation `Subobjects (collectId,subobjId)`, which combines the IDs of the collective objects (`collectId`) and the IDs of the subobjects (`subobjId`) that make up that collective object Relation Subobjects is available to the user to manipulate - to add and delete subobject IDs to the collective object ID but the attribute names (collectId and subobjId) are not allowed to be changed by the user The user is responsible for updating and keeping up to date the Subobjects relation, but it cannot be deleted, and no changes to its structure (only two attributes) are allowed

Example 5-4.

The `Room` class has characteristics such as `Length` and `Width`, etc Each piece of furniture is described in a Furniture branch of the inheritance hierarchy The `Couch` class also has attributes `Length` and `Width` The `Table` class has information about the type of `Wood` and the `Shape` of the table The `Chair` class specifies the `Height` of the chair and the size of the `Base` of the seat, assuming that all chairs have square seats

Inheritance Hierarchy




```
Room (RoId,Length,Width)
       31     25     40
       32     45     20

Couch                    Table                    Chair
(CoId,Length,Width)      (TaId,Wood,Shape)        (ChId,Height,Base)
  1      15      5          11  Oak  round          21     7.5    3
  2      17      5          12  Pine square         22     7      3.5
  3      18      6                                  23     8      3
```

Suppose that we have two rooms. a wide room has one couch, an oak table and two chairs; a long room has two couches, oak and pine tables and a 8" chair The insertion of all the pairs of IDs into the `Subobjects` relation is shown below:

let collectId **be** RoId, **let** subobjId **be** CoId,

Subobjects < + [collectId] **where** Width = 40 **in** Room) **ijoin** ([subobjId] **in**
(([CoId] **where** Length = 15 **in** Couch) [CoId **ujoin** TaId]
([TaId] **where** Wood = "Oak " in Table) [TaId **ujoin** ChId]
([ChId] **where** Height < 8 **in** Chair)),

Subobjects < + [collectId] **where** Length = 45 **in** Room) **ijoin** ([subobjId] **in**
(([CoId] **where** Length > 15 **in** Couch) [CoId **ujoin** TaId]
([TaId] **where** Wood = "Oak " **or** Wood = "Pine" **in** Table)
[TaId **ujoin** ChId] ([ChId] **where** Height = 8 **in** Chair));

```
.Collection                          Subobjects (collectId,subobjId)
(.collectClass,.subobjClass)                31        1
      Room          Couch                   31        11
      Room          Table                   31        21
      Room          Chair                   31        22
                                            32        2
                                            32        3
                                            32        11
                                            32        12
                                            32        23
```

■

5.2 Functionality of Collection Hierarchy

The collective classes are stored in the inheritance hierarchy. To distinguish collective classes from non-collective classes and to identify the classes of objects held in each collective class, the collection hierarchy is used. Thus every collective class is found in both inheritance and collection hierarchies. The collection hierarchy refers to the classes found in the inheritance hierarchy.

In many ways the two hierarchies are similar. The difference is that the inheritance mechanism works in the upward direction, looking for superclasses, and the collection mechanism works in the downward direction, looking for subobjects.

In Section 4.3 we have shown how the keyword ***super*** is used to access the ***superclass's*** definition of the method that is defined in the current class. A ***sub*** keyword is similar to the ***super*** keyword, but it is used to access the ***subobject's*** definition of a method, even if that method is defined on the collective class.

let size **be sub** Calc_size();
Possible_furniture_sizes ← [RoId, size] **in** Room;

The subobject IDs can be projected together with the broadcasted methods. To avoid confusion (as subobject classes may have different ID fields), the IDs of the subobjects will be projected under the field name "subobjId". `SubobjId` will be an alias of the ID names, otherwise, any duplicate values returned by the methods will be ignored.

5.3 Interpretation of Messages Sent to a Collective Class

When a method is applied to a collective class, several possibilities arise:

1) the method is defined on the collective class: no broadcasting is performed whether or not the method is defined also on the subobjects;
2) the method is not defined on the collective class: broadcasting occurs, and the method is projected from those subobject classes for which it is defined;
3) the method name is prefixed with the keyword ***sub***: same as case 2) above, even if that method is defined for the collective class, the method is not invoked on it.

In summary, broadcasting is performed if among the methods applied to a class there is at least one method which either is called with the keyword ***sub*** or is not defined on the class to which it was applied.

Example 5-5

To illustrate the first and third case, find the free area of a room (room size: the sum of the sizes of the furniture in the room). The method `Calc_size` is defined on both the `Room` class and the subobject classes, `Couch`, `Table`, and `Chair`.

```
function Calc_size () on Room
  { return (Length * Width) }

function Calc_size () on Couch
  { return (Length * Width) }

function Calc_size () on Table
  { area ← if Shape = "round " then 100 else 150;
    return (area)  }
```

```
function Calc_size () on Chair
  { return (Base * Height) }

let sizeRoom be Calc_size();
let sizeSub be sub Calc_size();
```

Here `sizeRoom` is going to hold the result of the invocation of the `Calc_size` method on the `Room` class. `SizeSub` will hold the result of the invocation of the `Calc_size` method on the subobjects of the `Room` class. The only way to invoke `Calc_size` on the subobjects is with the help of the keyword ***sub***

To find the occupied area of the room, use the equivalence reduction that sums all the sizeSub attribute values for each room ID. We need to project the IDs of the subobjects so as not to loose any duplicate sizes. So we project the `subobjId` field. When method broadcasting occurs, the system makes this field available to be used as a reference to the IDs of subobjects (see Example 5-7 for details of its generation).

```
let free_area be sizeRoom − (equiv + of sizeSub by RoId),
Free_space ← [RoId,free_area] in ([RoId,sizeRoom,subobjId,sizeSub] in Room);
```

■

Example 5.6

To illustrate the second case, consider a method, `Heating_cost`, defined on the `Room` class, and a method, `Present_value`, defined on the `Couch` and `Table` classes.

```
let Heating be Heating_cost();
let Value be Present_value(),
Evaluation ← [RoId,Heating,subobjId,Value] in Room;
```

The method `Heating_cost` will not be broadcasted since it is defined on the collective class. The method `Present_value` is going to be broadcasted to all the classes of subobjects without the help of the **sub** keyword because `Present_value` is not defined on the collective class, but only on the subobject classes.

■

The following example demonstrates in general terms how a relational expression which contains a combination of all three types of method invocations is transformed into an expression which can be evaluated using the techniques discussed in the previous chapters

Example 5-7.

Given the following input expressions:

let m1 **be** method1 ();

let m2 **be** method2 ();

let m3 **be sub** method3 ();

R ← [Id,m1,subobjId,m2,m3] **in** collectiveClass;

where `method1` is defined on the collective class `collectiveClass`, `method2` is defined on the classes S1 and S2 of subobjects of `collectiveClass`, and `method3` is defined on both the `collectiveClass` and its subobject classes S1 and S2

Output expression:

R ← [Id,m1,subobjId,m2,m3] **in** (I-1)

([Id,m1] **in** collectiveClass) (I-2)

[Id **ijoin** collectId] Subobjects (I-3)

ijoin ([subobjId,m2,m3] **in** (I-4)

(([Id1,m2,m3] **in** S1) [Id1,m2,m3 **ujoin** Id2,m2,m3] ([Id2,m2,m3] **in** S2)), (I-5)

I-1 is the original projection

I-2 is a relational expression on the `collectiveClass`, similar to the input expression but containing only attributes and methods defined on that class, thus it can be evaluated as described in previous chapters

I-3 is needed to select only the IDs of the subobjects of `collectiveClass`.

I-4 gets the subobject IDs and the results of the invocation of methods m2 and m3 on all subobjects

I-5 invokes methods m2 and m3 on each subobject class, and then joins the results of these invocations

■

5.4 Algorithm for Implementation of Method Invocation on Collective Classes

Given an input expression which involves method invocation on the collective class `collectiveClass`,

R ← [Id,m_1, .,m_n] **in** collectiveClass;

where m_1 m_n represent methods some of which may be declared with the **sub** keyword.

Determine which methods are applicable to the collective class, and which ones need to be broadcasted to subobjects. If there are no methods to be broadcasted, then the input expression can be evaluated as described in the previous chapters.

Relational expressions which involve method broadcasting cannot be evaluated directly. They are replaced by a system generated output expression, which can be evaluated using techniques discussed in the previous chapters. There are three main parts to generating the output expression. Example 5.7 illustrates this algorithm.

1. Treatment of the collective class

   Find the attributes and methods defined on the collective class. Generate the expression for the collective class (1-2)

2. Treatment of the subobjects

   Find the methods to be broadcasted. Determine the subobject classes and their ID fields. For each subobject class generate a projection of its ID and the results of method invocations. Create a union of these projections (1-5)

3. Generation of the output expression

   Bring together method invocations on the collective class and on its subobjects by selecting the actual subobjects from the subobject classes (1-3).

For details of this algorithm, see Appendix C.

Chapter 6

Comparative Study of the Objective-C Implementation of Gedit and Its Proposed Implementation in Object-Oriented Relix

The purpose of this chapter is to illustrate how the object oriented features discussed in the previous chapters are used in an application program. This chapter gives examples of

- adding new objects to the hierarchy of classes (operation "creation of objects" on p 122);
- applying a method to the collective class (function `Display` (p 116) in the operation "creation of objects"),
- using generic methods (function `Draw_line` (p 117) in the operation "creation of objects");
- using methods which are defined on a subclass and which reference inherited attributes (functions `Overlap`, `Draw_frame` on p 117),
- updating an inherited attribute (operation "selection of several objects" on p 124),
- invoking a procedure (procedure `Move_obj` (p 117) in the operation "moving selected objects" (p 125)),
- calling a superclass procedure from a subclass procedure (procedure `Move_obj`);
- applying a procedure to a subset of a class, where the selection of affected objects is done outside the procedure (procedure `Move_obj` on the `Graphics` class);
- applying a function to a subset of a class, where the selection of affected objects is done inside that function (function `DrawSelf` on the `Constraints` class (p 117))

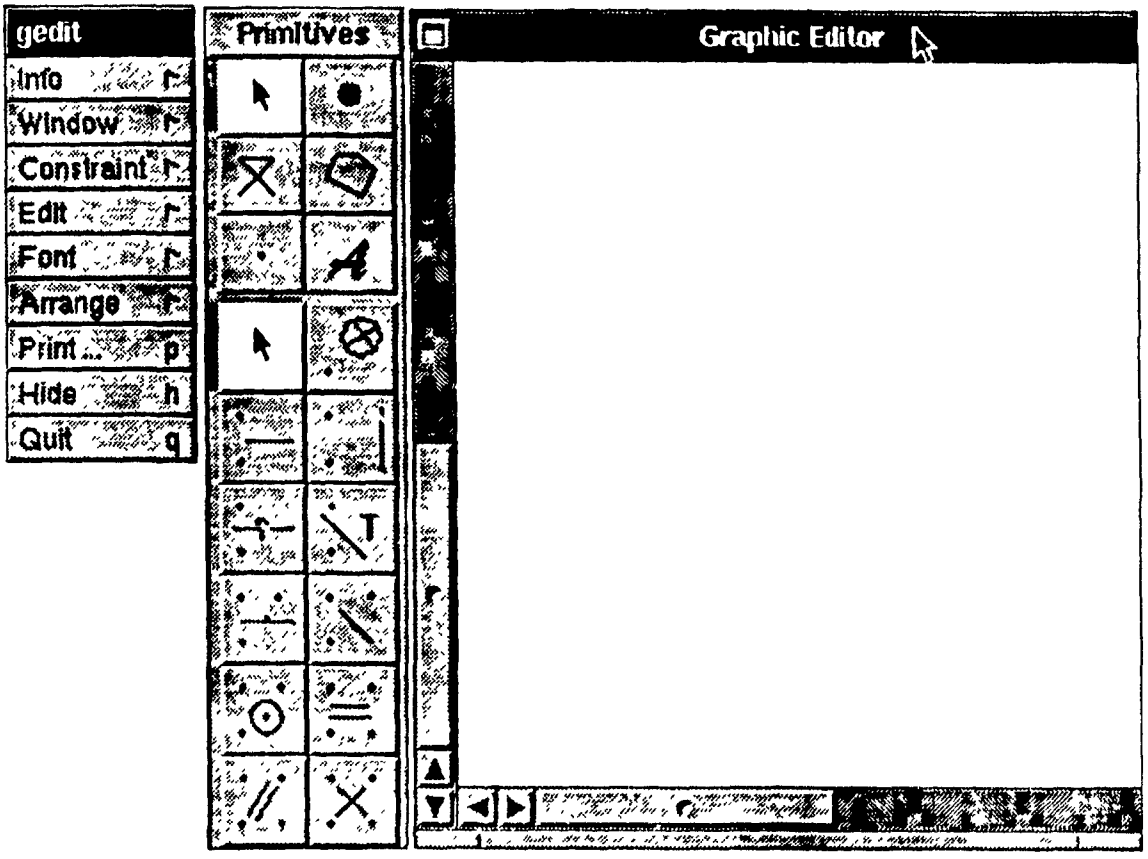
First, we present a short tutorial on the Graphics Editor (Gedit) to introduce the reader to the basic operations provided by the editor. Section 6.2 outlines the main characteristics and considerations required by the Objective-C implementation of the Gedit in the NeXT environment. Section 6.3 proposes how Gedit can be implemented in Object Oriented Relix.

6.1 Graphics Editor Implemented in Objective-C

The Graphics Editor (Gedit) is implemented on the NeXT machine [Huxx91]. It allows us to create simple objects like point, circle, polylines and polygons, and to apply constraints to them. In the following section we will demonstrate how to manipulate objects in Gedit.

6.1.1 Introduction to Gedit and a Short Tutorial

Gedit consists of three main components: the menu, the Object and the Constraint Primitives panel, and the drawing window. When a Gedit is first invoked, all its components appear as follows.

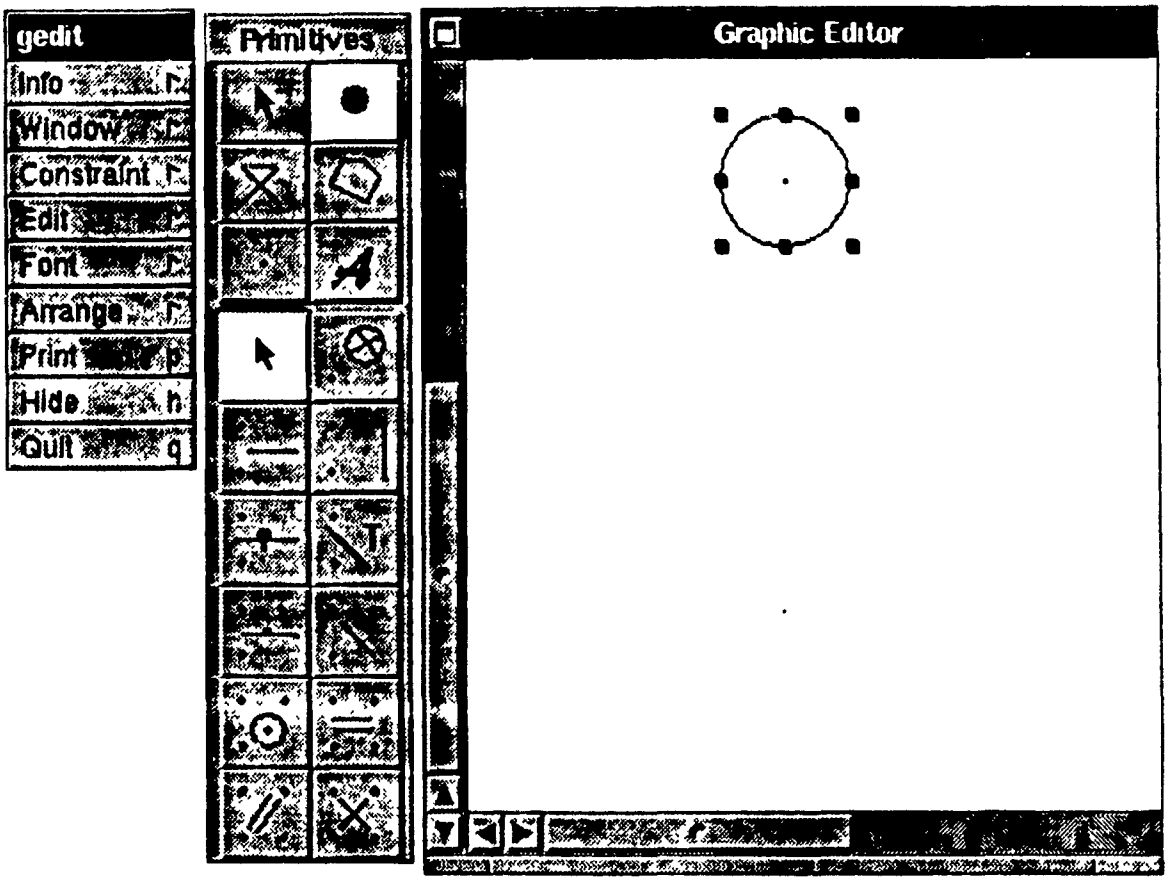


The top 6 icons of the Primitives panel constitute an object factory and allow the creation of objects like circle, polyline, polygon, point and text. The bottom 12 icons constitute the constraint factory and are used to impose different constraints on objects.

6.1.1.1 How to Create Objects

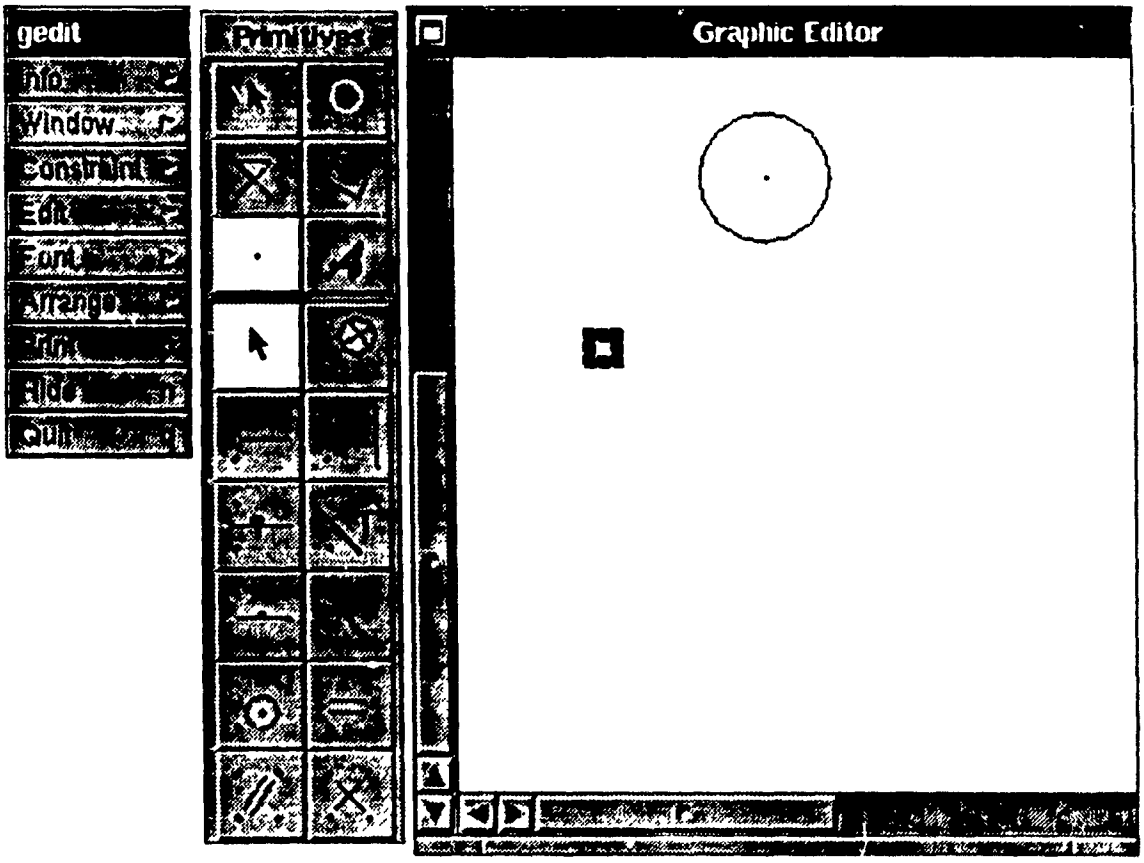
Creation of an object is done in two steps: a) selecting the corresponding icon from the Primitives panel by clicking the left mouse buton on it, and b) drawing the object in the window.

Let us create a circle. Click with the left mouse button on the circle icon. The icon becomes highlighted (its background is white). Each subpanel of primitives -- object factory and constraint factory -- can have only one selected icon at any time. Now move the cursor to the drawing window. Press the left mouse button down where you want the center of the circle to be, and drag the mouse until the desired radius is reached. At that point release the button.



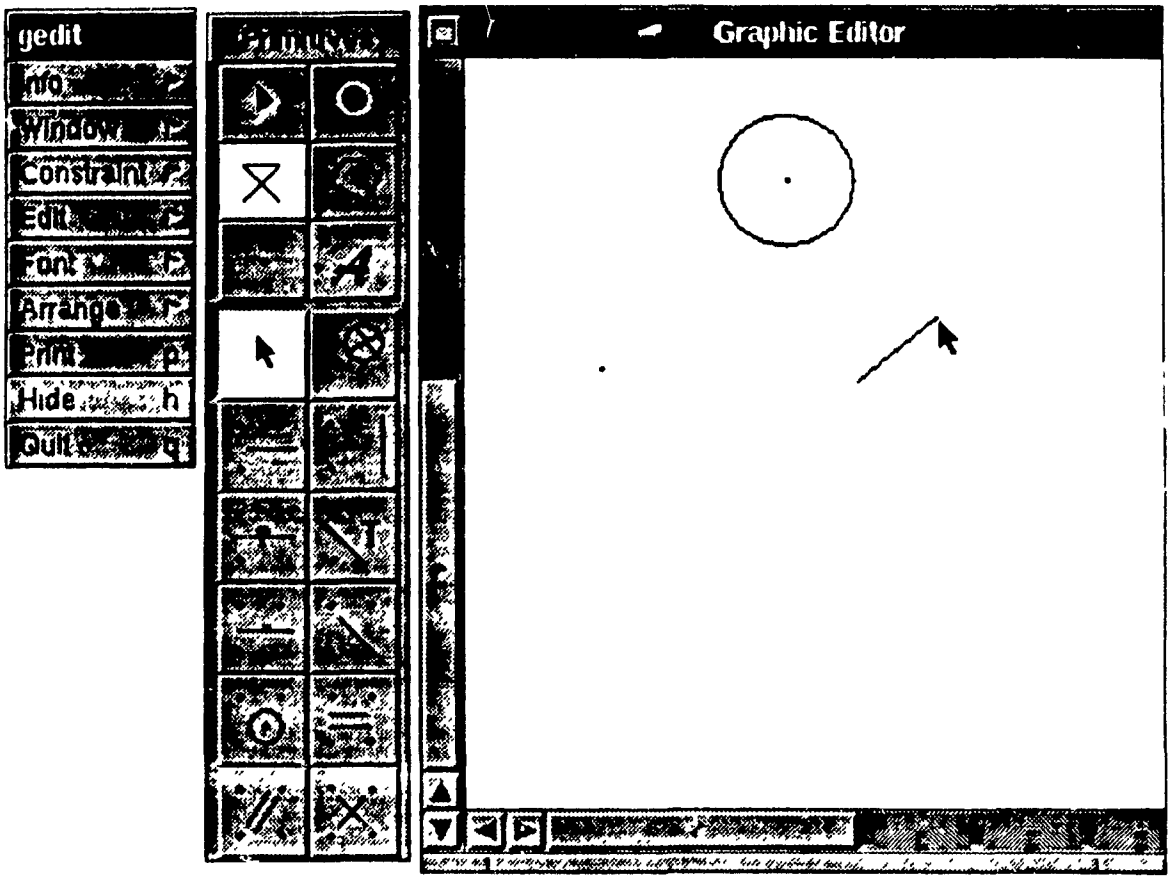
You see 8 dots around the circle. They show the frame of the circle. Every object in Gedit has a frame -- a rectangular region enclosing the object. A frame displayed around an object indicates that either the object has just been created, or it has been selected.

To create a point, use the mouse to select the point icon in the object factory. Position the mouse on the desired location of the drawing window. Then make a single click with the left button (press the button and release it immediately).

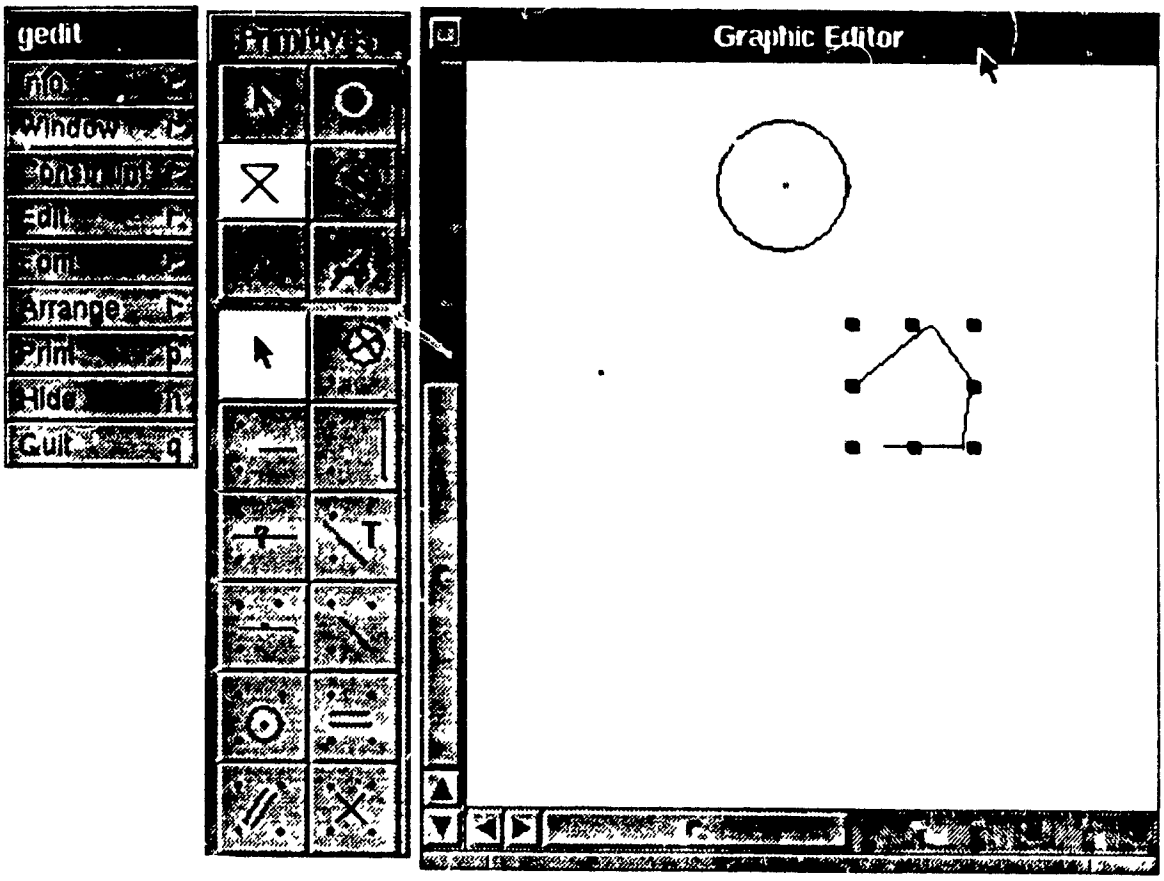


Notice that the frame around the circle has disappeared. Now the frame around the point is shown. We can draw several points by clicking in different places of the window.

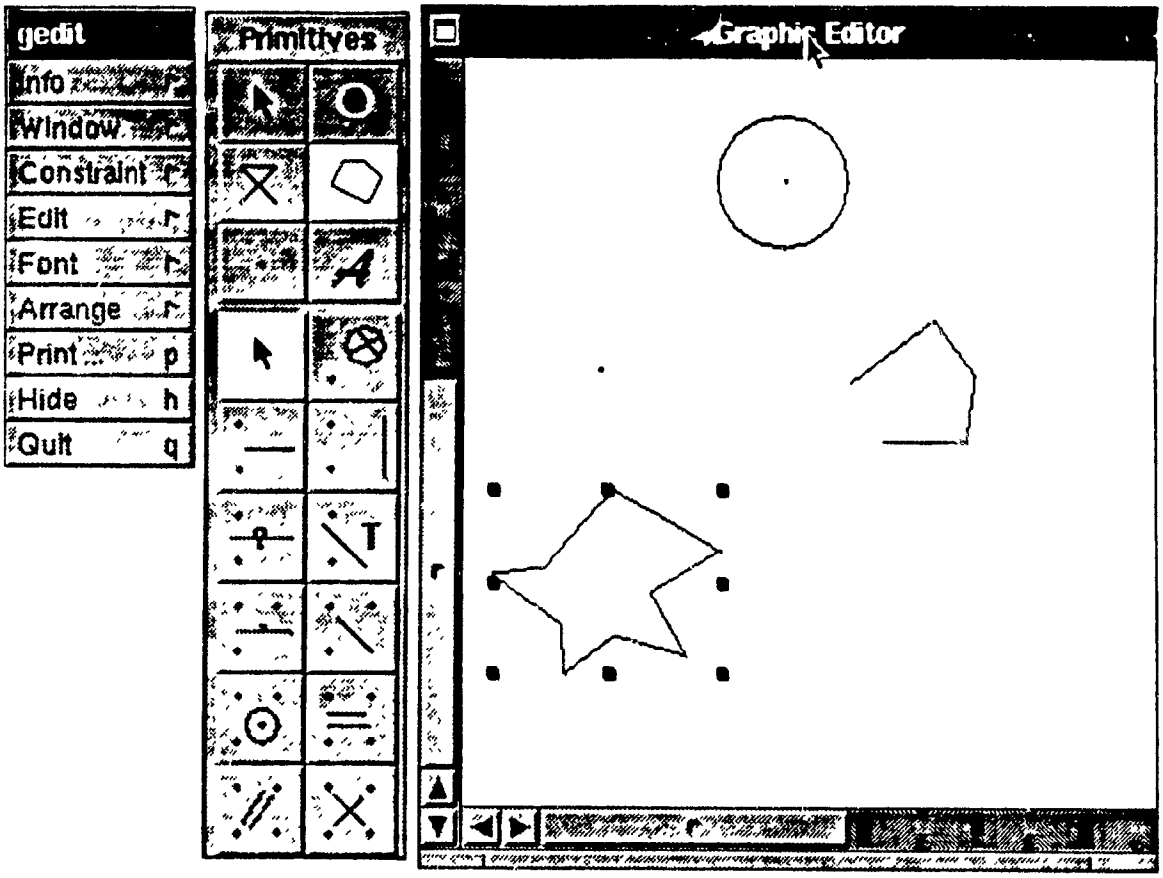
To draw a polyline, select a polyline icon with the left button. Position the mouse on place in the drawing window where you want the first vertex of the polyline to be. Click the left button once. Press the left button down, drag it to where the first line segment will end, and release the button. During the dragging process the line segment is drawn from the previous vertex to the current cursor position to aid the user in determining how the line would look if the button were released at that particular moment.



Repeat the process of pressing the button, dragging it and releasing it, until all the line segments of the polyline are created. The last vertex is signalled by a double click on the left button. When Gedit receives the double click, it completes the creation of the polyline and draws a frame around it.

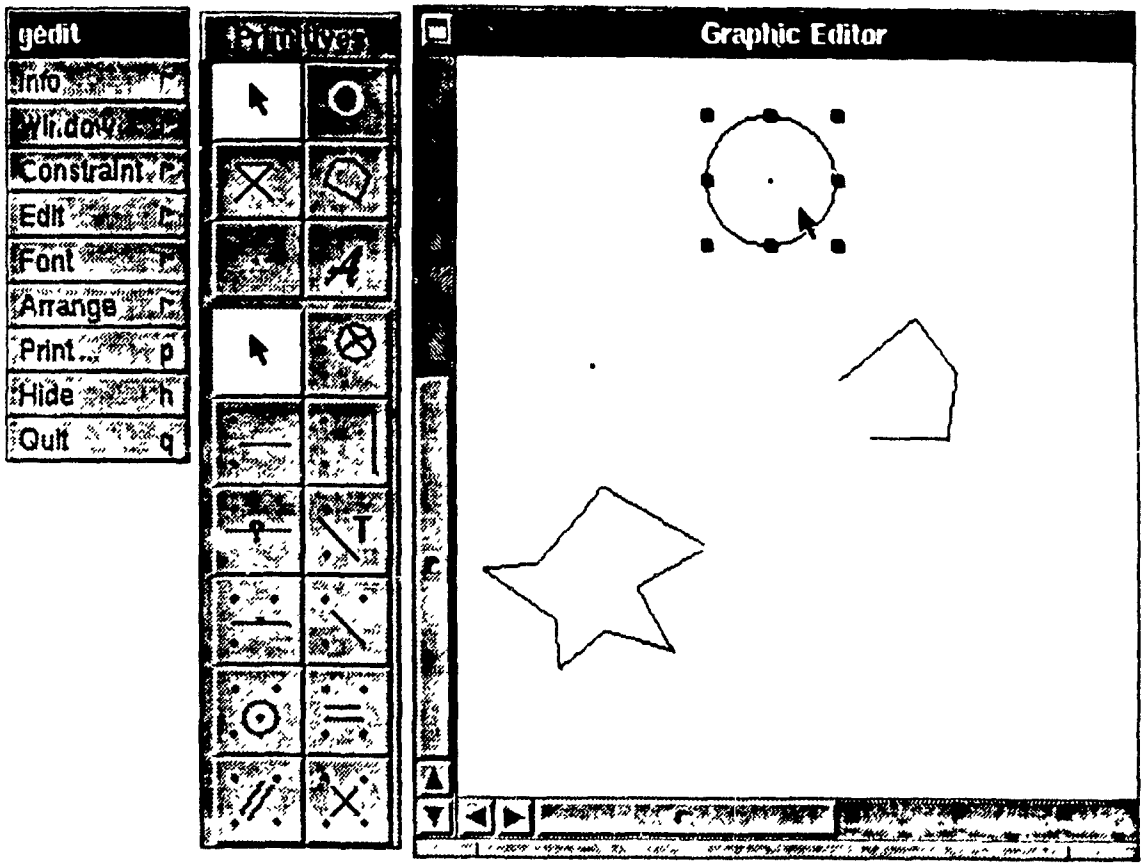


A very similar procedure is used for creating a polygon, with the difference that a double click at the end of creation connects the last vertex with the first vertex, thus closing the polygon.



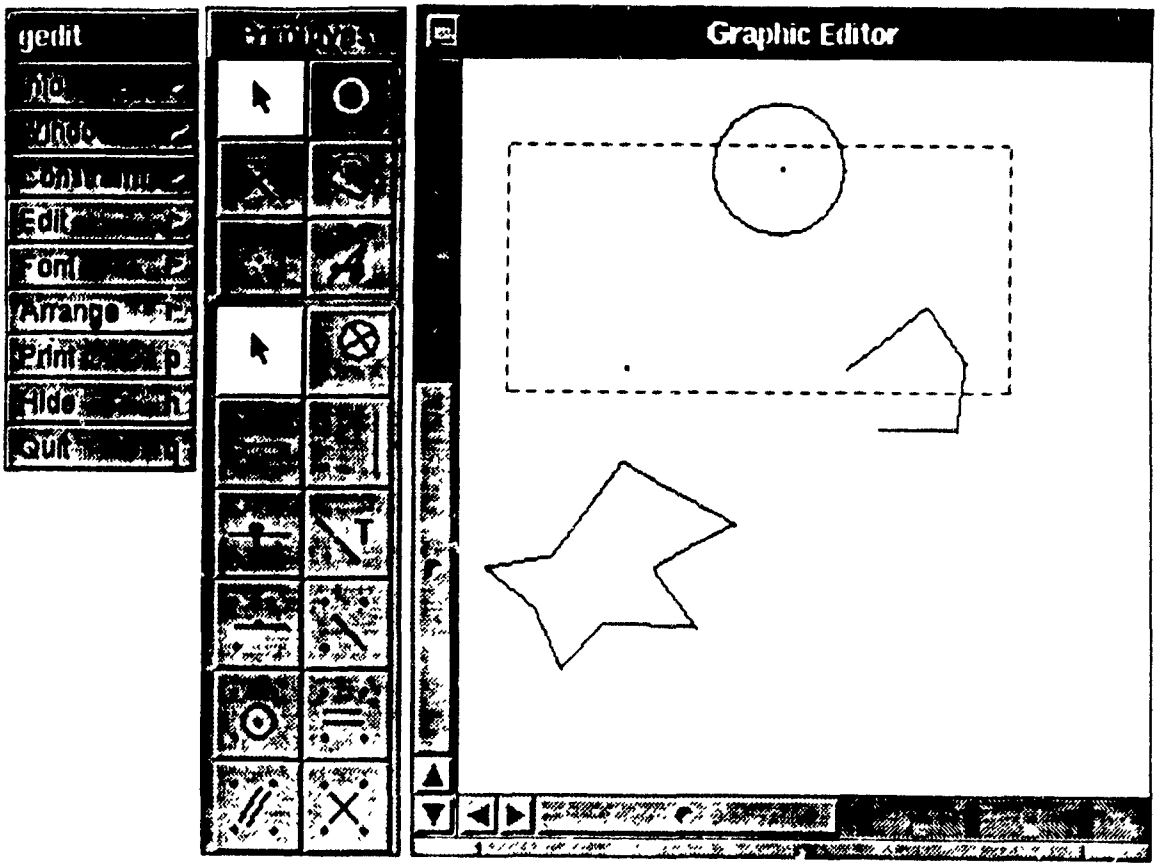
6.1.1.2 How to Select Objects

Object selection can be performed only when the arrow icon of the object factory is highlighted. Objects are selected for such operations as moving, zooming, resizing, removing, etc. To select an object, click the left button anywhere inside the object's frame. Below we show how the screen will look after the left button is released.

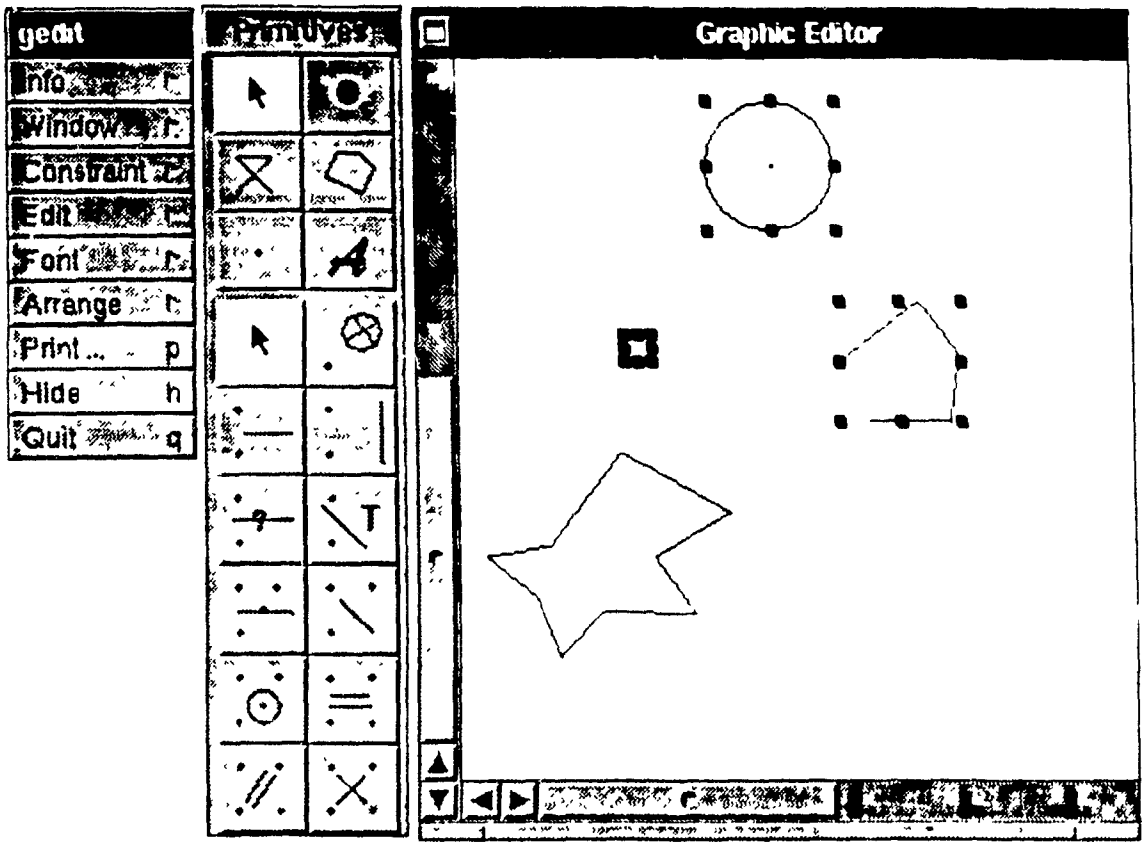


Several objects can be selected at the same time. These objects must be inside or intersecting a selected region in the window. To select the region, press the left button in one corner of the region and drag the mouse until the opposite corner is reached. Release the button.

Below we show the selected region; the left button is not released yet.

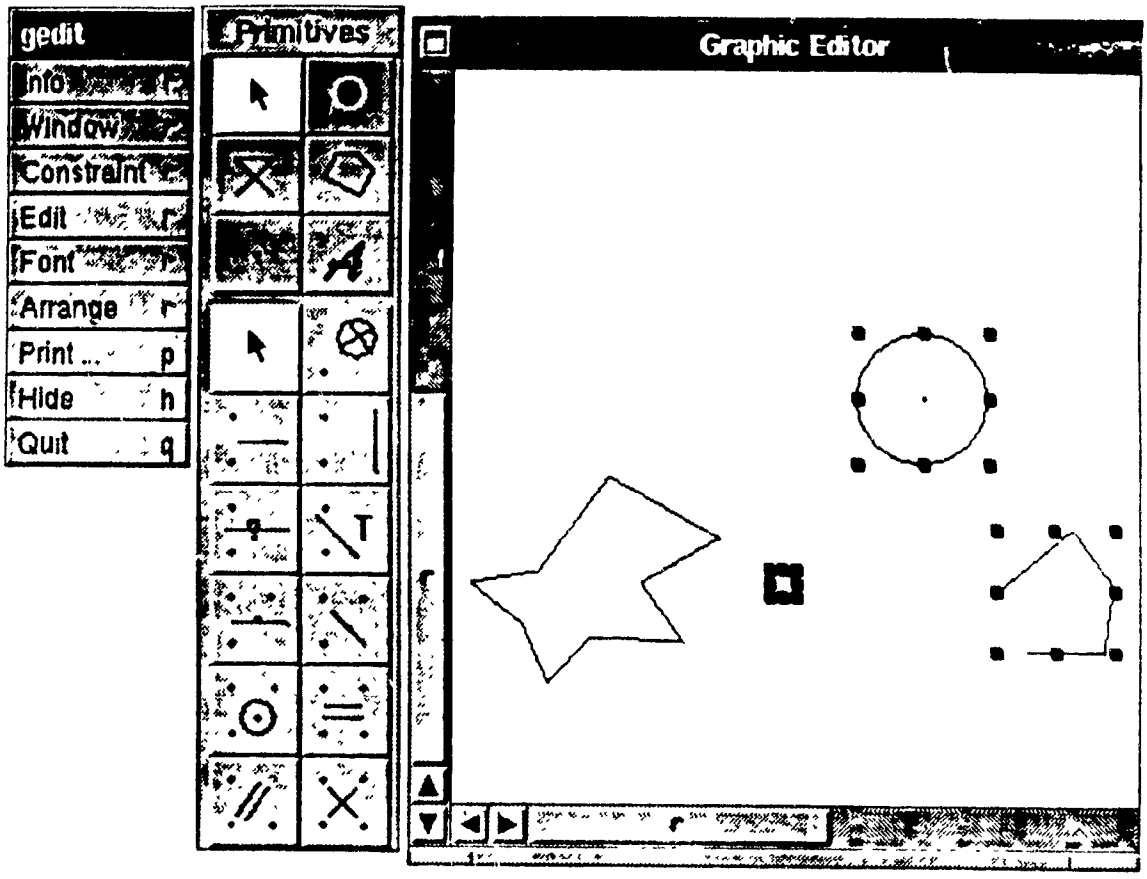


Now release the button and all the objects in the selected region are selected.



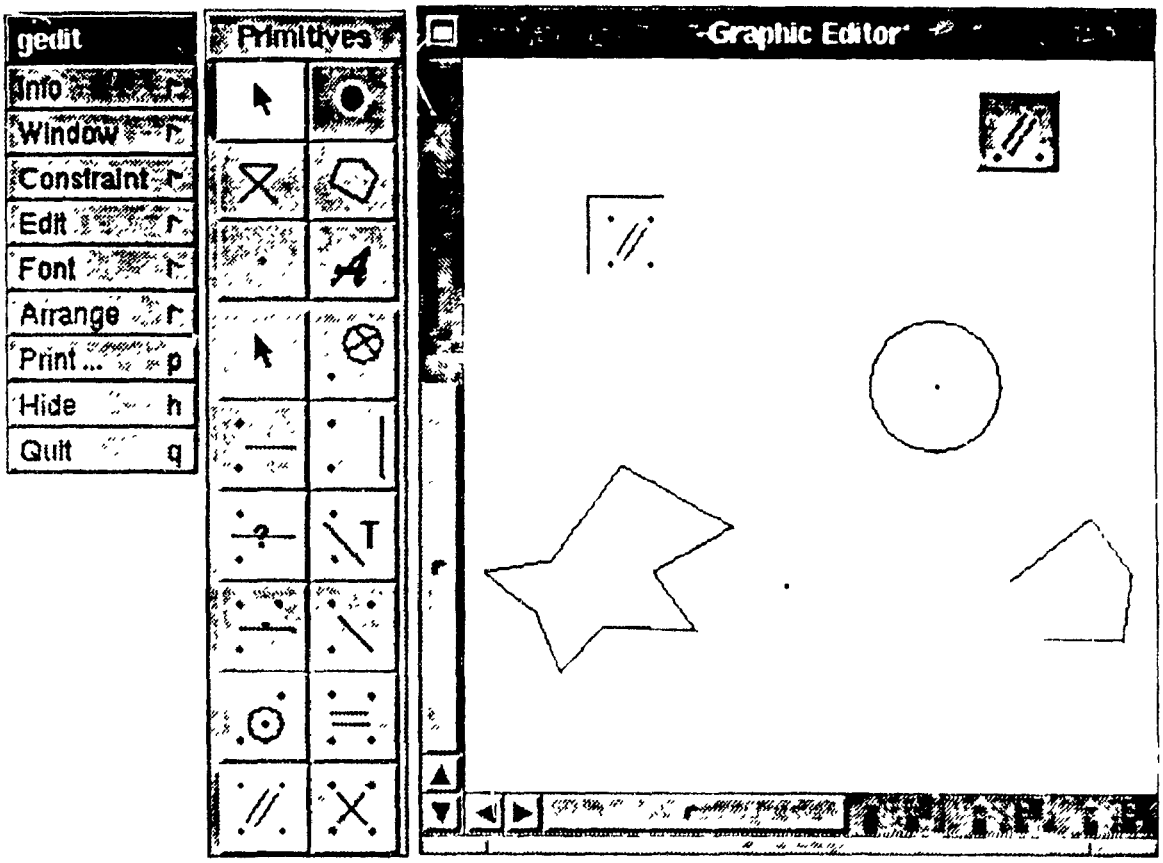
6.1.1.3 How to Move Objects

Moving an object or several objects is done in three steps: a) select the object(s), b) press the left button inside the object's frame, and c) drag the mouse in the desired direction As the cursor moves during the dragging event, the object moves with it. Releasing the button indicates the end of the moving process.



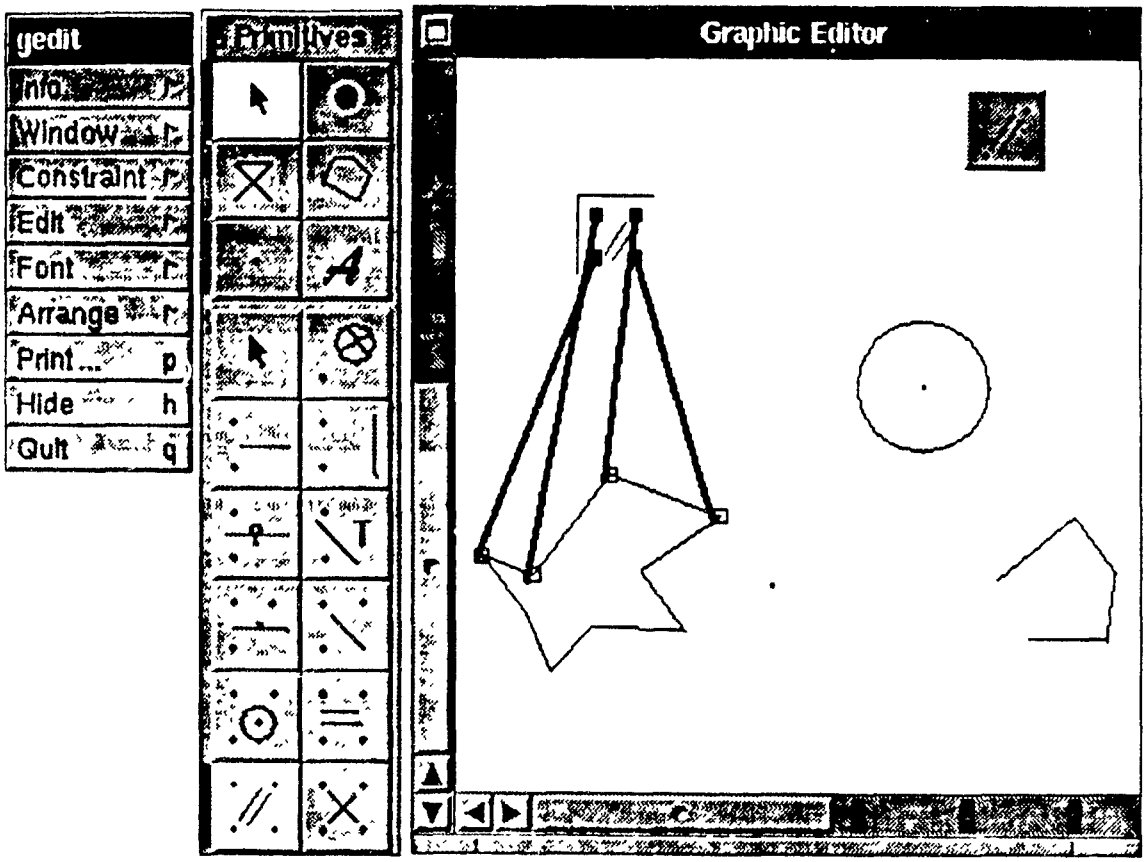
6.1.1.4 How to Use Constraints

We can impose some constraints on objects or parts of them. For example, it might be necessary to make two adjacent line segments of a polyline perpendicular to one other This is done with the help of the constraint icon of the specified type Constraint icons have from 1 to 4 arms, which are connected to the vertices of the objects to be constrained In the previous example, the "perpendicular" constraint has 4 arms the first two are for the end points of the first line segment, and the second two are for the end points of the second line segment. There are 11 different constraint icons in the constraint factory To apply a certain constraint to an object, select the corresponding constraint icon from the constraint factory (by clicking on it with the left button) and place that icon in the window (by clicking on the desired spot with the right button) Any subsequent right button clicks in the window will create new icons of that type. For example, in the picture below, the right button was clicked twice in the window after the parallel constraint was selected from the constraint factory

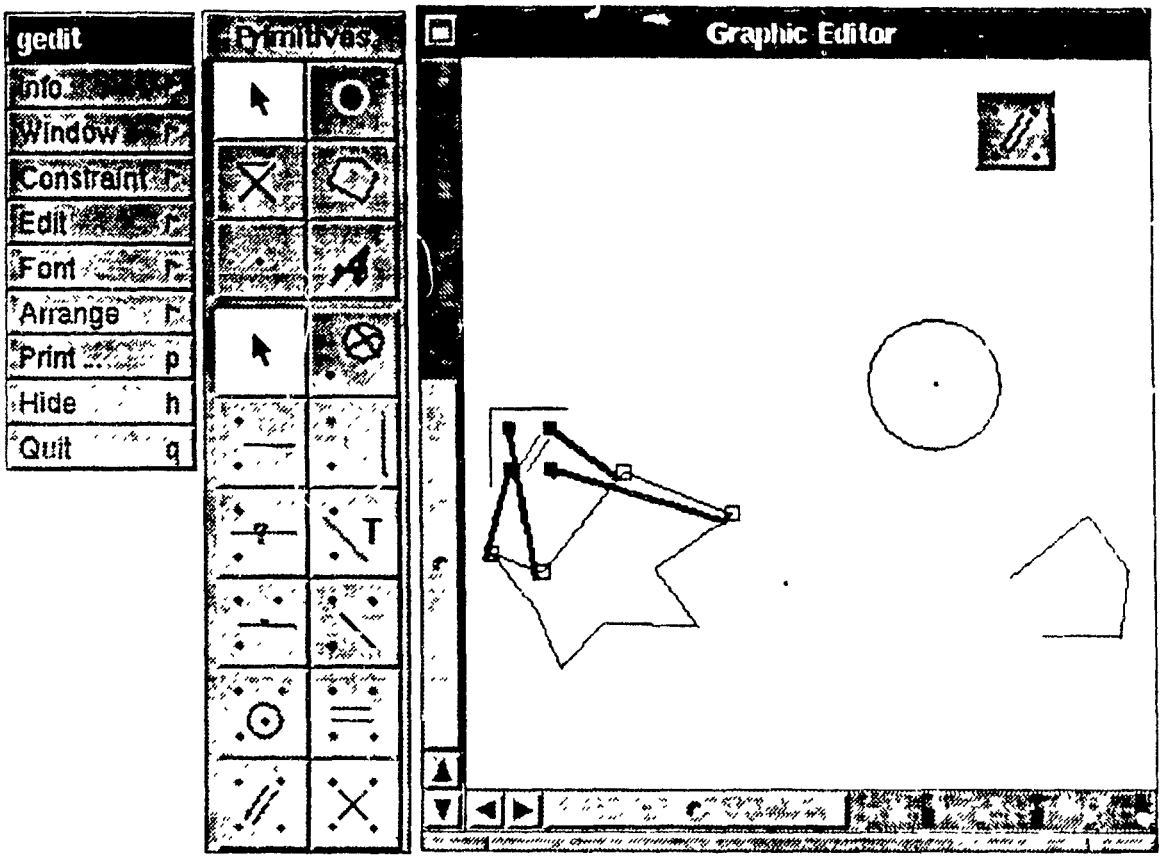


To enforce the constraint, its arms need to be connected to the vertices of the objects. The arms are located in the corners of the constraint icon.

Arm 1 is in the bottom left corner of the icon. Arms are numbered in a clockwise direction and are shown with a dot in the corner of the icon. To connect an arm of a constraint icon, press the right button down on or around the corresponding dot, and drag the mouse to the vertex of the object. When the vertex is reached, release the button. Repeat the process until all the arms have been connected. When all the arm connections have been established, the constraint is enforced.



To move a constraint, the arrow icon should be selected in the constraint factory. Follow the same procedure as that for moving objects As the constraint is moved, the arms remain attached to the vertices.

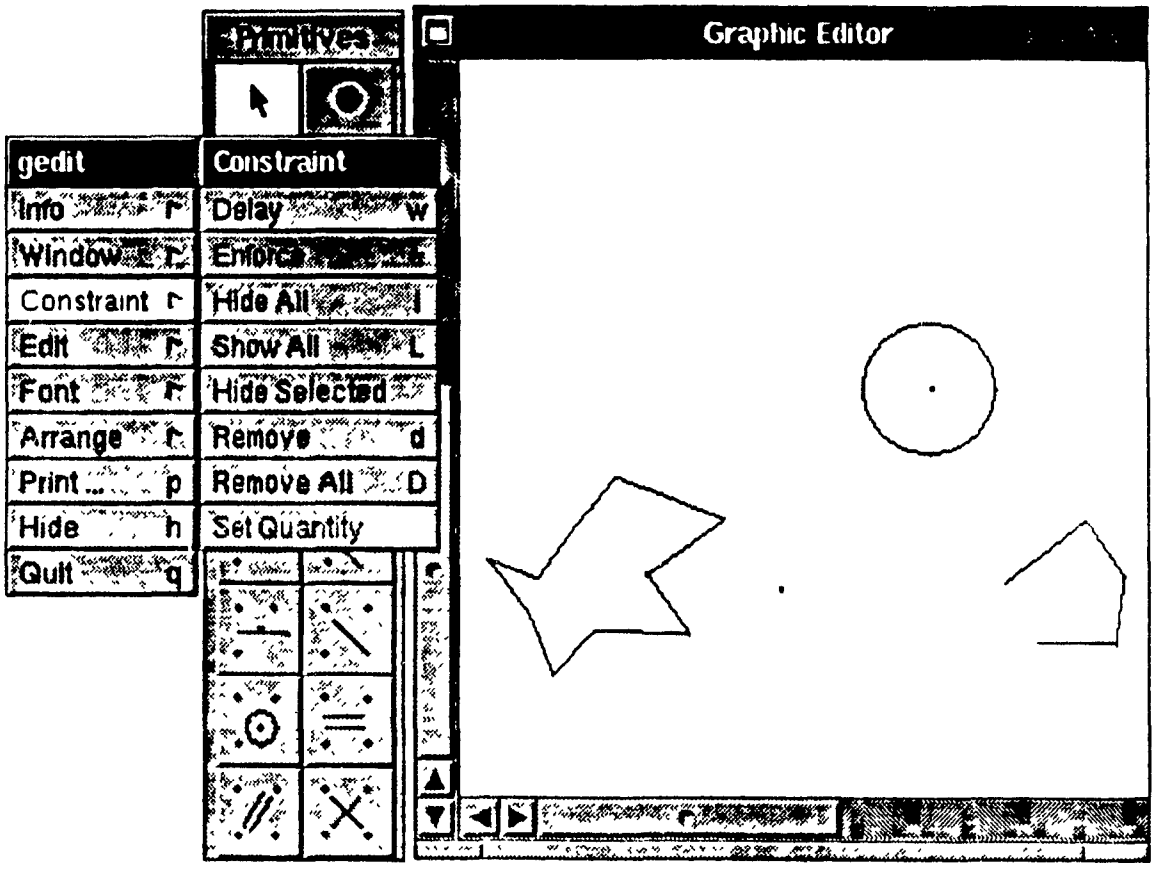


6.1.1.5 Invocation of Operations Through the Menu

Some operations can be invoked only through the menu. Here we discuss the "Hide All Constraints", "Show All Constraints", "Zoom out", and "Zoom in" operations.

How to Hide and Show Constraint Icons.

If we do not want to see the constraint icons with their arms in the drawing window, but we still want them to be enforced, we can hide them temporarily. To do that, select a Constraint option in the main menu by clicking once on it with the left button. From a Constraint submenu select the Hide All option. The result of that operation is shown below.



The hidden icons can be redisplayed by selecting the "Show All" option from the same menu.

How to Zoom out and Zoom in Objects

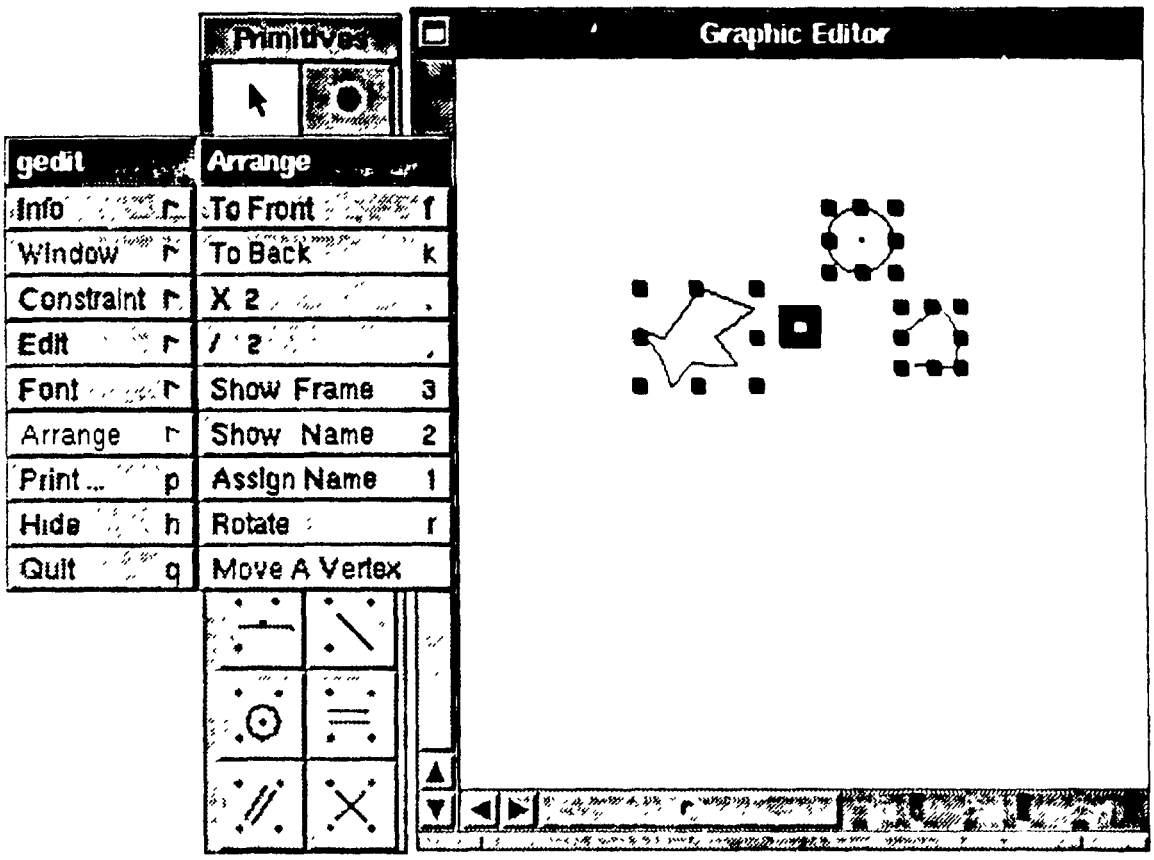
Graphical objects can be zoomed out. This operation takes selected object(s) and halves them in size. If more than one object is selected, the distance between the vertices of the selected objects and the midpoint of the rectangle encompassing them is also doubled. To zoom out selected objects, follow these steps:

a) select one or more objects in the drawing window,

b) call up the Arrange submenu by clicking once on the Arrange option in the main menu; and

c) select the [/ 2] option from the Arrange submenu.

As a result we see that all the objects in the selected region become smaller in size.



To achieve an enlarging effect, zoom the selected objects in by selecting the [X 2] option in the same menu.

6.1.2 Summary of the Basic Gedit Operations Described in the Tutorial

Below we give an outline of the sequence of mouse clicks and drags necessary to complete each of the operations described in the above tutorial.

Creation of an Object

a) Select the desired icon from the object factory by clicking on the icon with the left button.
b) Create an object in the drawing window by a series of left button clicks and drags. Each type of object requires a different sequence of mouse movements and/or clicks, summarized below
   - point. click once,
   - circle. create the center by pressing the left button down; drag the mouse till the desired radius is reached; then release;
   - polyline and polygon place the first vertex by clicking once; press the left button down to prepare to place the next vertex; drag the mouse until the desired position is reached; release the button to create the vertex; repeat the process of creating vertices until the last vertex is created; double click to finish the object creation.

Creation of a Constraint

a) Select the desired icon from the constraint factory by clicking on the icon with the left button.
b) Create a constraint in the drawing window by a single right button click on the desired location.

Connection of the Arms of the Constraint

Repeat the following steps for each arm of the constraint icon that is to be connected to the objects in the drawing window:

a) Position the cursor on the dot corresponding to the arm to be connected and press down the right button;

b) Drag the mouse to the vertex of the object to which the arm will be connected;

c) Release the button when the vertex is reached.

Selection of an Object

a) Click the left button on the arrow icon in the object factory.

b) Click the left button inside the object to be selected

Selection of Several Objects

a) Click the left button on the arrow icon in the object factory.

b) With the cursor located in a corner of the rectangular region containing the objects to be selected, press the left button down. Drag the mouse until the opposite corner of that region is reached, and release the button. At this point all the objects whose frames overlap the selected region are selected.

Selection of One Constraint

a) Click the left button on the arrow icon in the constraint factory.

b) Click the left button inside the constraint icon in the drawing window.

Moving Selected Objects

a) Click the left button on the arrow icon in the object factory.

b) Press the left button down inside the frame of one of the selected objects, drag the mouse to the new location, and release the button. All the selected objects will be moved at the same time

Moving Selected Constraint

Press the left button down on the selected constraint icon in the drawing window. Drag the mouse to the new location and release the button.

Hiding Constraints / Showing Constraints

a) Click the left button on the Constraint option of the main menu.

b) Click the left button on the appropriate option of the Constraint submenu.

Zooming Selected Objects

a) Click the left button on the Arrange option of the main menu.

b) Click the left button on the appropriate option in the Arrange submenu.

6.2 Objective-C Environment

The implementation of Gedit is simplified by the fact that it is written in the NeXT environment. The NeXT environment supplies some common classes. The classes are linked together in a hierarchical tree with Object class at its root. Each class inherits both instance variables and methods from its superclasses. Any new user defined class must be linked to the hierarchy by declaring its superclass.

The inheritance of instance variables means that the subclasses do not have to declare those variable themselves -- their declaration is implicit. Since class instances inherit variables and not values, every instance of the same class has its own copy of all the instance variables declared for the class, thus each object controls its own data. A subclass cannot override an inherited instance variable. This means that a subclass cannot declare a new variable with the same name.

Internally, an object ID is a pointer to the data structure that contains all the instance variables of that object.

The NeXT environment supplies some important functions. When a "mouse down" event occurs, the NeXT identifies the object on which the cursor is positioned and sends the "mouse down" message to that object. For example, when the right mouse is pressed on the constraint icon, the "mouse down" message is sent directly to that constraint object.

In Objective-C a message is sent to the individual objects. To send a message to some or all objects of a class, the IDs of those objects are added to an instance of a list class and the method is performed on all the elements of that list object. Gedit uses three lists: ***Glist,*** to hold the IDs and frames of all the graphical objects defined in the window; ***Clist,*** to hold the IDs of all the constraints defined in the system; and ***Slist,*** to hold the IDs of all selected objects.

6.2.1 Method Inheritance

An object can inherit behaviour defined for its class and for its superclasses. In the inheritance of behaviour (unlike the inheritance of instance variables), an object does not

keep a copy of methods that were defined for its class or for its superclasses. The object just accesses (or calls) them when it receives a message requesting that method. When an object receives a message, the system searches for the requested method in the class's table. If the method is not found, the system searches for it in the object superclass's table, and so on, up to the root, Object class, table. Once the method is found, the system passes the object's instance variables to the method and invokes the method.

There are three ways to inherit superclass's methods:

- ***Simple inheritance*** subclass uses all the methods defined in its superclass.
- ***Partial inheritance*** a method defined in the superclass may call another method for which the search will start from the subclass and, if it is not defined there, its definition in the superclass will be used
- ***Selective inheritance***: a method is defined both in the superclass and the subclass; the subclass's version of the method overrides the superclass's method.

An object may use the method defined in its own class by sending the message to ***self***

[self method]

Alternatively, it may use the method defined in its superclass by sending the message to ***super***

[super method]

6.2.2 Inheritance Hierarchy

The partial inheritance hierarchy of Gedit is shown in Figure 6.1. We only show classes and instance variables that are going to be used in our discussion below. The hierarchy contains both the NeXT supplied classes (`Object, View, List, Button`) and the classes defined by Gedit (`Graphics, Point, Circle, Poly, Constraint, Glist, Clist, Slist`). Instance variables are listed in curly brackets.

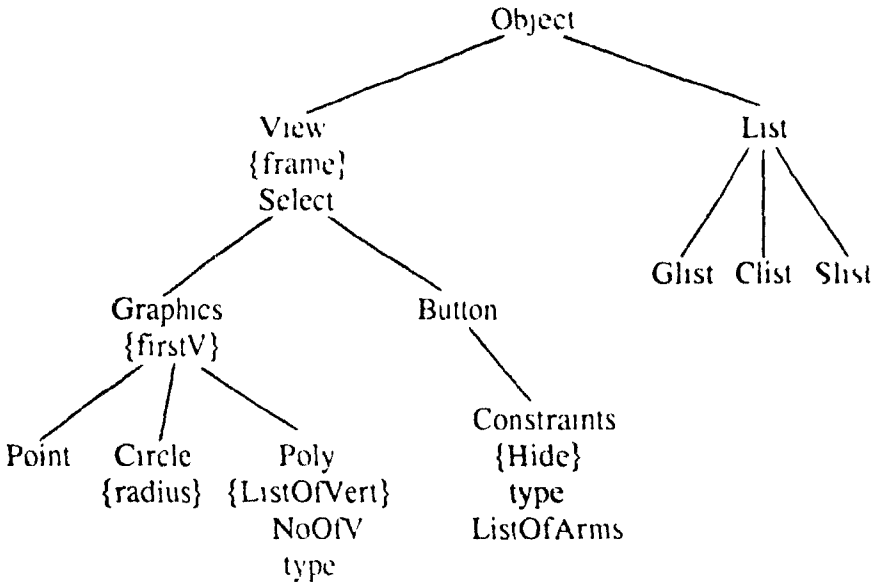


Figure 6.1 Inheritance Hierarchy of Gedit

The meaning of instance variables declared in this hierarchy are explained below. Note that the types of instance variables can be simple (Boolean, char, int), structures, or linked lists

frame - Every graphical object has a frame, which is a rectangular region that completely encloses all the vertices and sides of the object. In Objective-C, a frame is represented by its bottom left corner (origin), and its size: width and height. Frame plays an important role in the way objects are manipulated. Its primary purpose is to identify the object being selected. Also, selected objects have their frame displayed to distinguish them from non-selected objects. The frame display consists of 9 small squares along the perimeter of the rectangular region that encompasses that object.

Select - The variable `Select` is set to True if an object is selected and to False otherwise.

firstV - Every object has one or more vertices. The first vertex is of particular importance for the `Polygon` class since, when displaying the object, its first and last vertices are connected automatically by a line to close the polygon

radius - Objects of class `Circle` can be either circles or ellipses. To allow for both shapes, the radius has two components: horizontal and vertical radii.

ListOfVert - For each vertex of a poly-figure, the `ListOfVert` contains the ID of the vertex, the sequence number of that vertex in the poly-figure, and the x and y coordinates of that vertex

NoOfV - This variable specifies the number of vertices in a poly-figure.

type - In a `Poly` class, type has two possible values: G for polygon, and L for polyline. In a `Constraints` class, type indicates the constraint name: P for parallel, p for perpendicular, etc.

ListOfArms - This list contains an entry for each connected arm of the constraint. The arm information includes the arm#, the ID, and the sequence number of the vertex to which that arm is connected.

A general idea of how the inheritance of attributes and methods is used in Objective-C can be illustrated by showing the implementation of a sample of the basic operations. For this purpose, four of those operations are selected in Gedit: creation of an object; selection of several objects; moving selected objects; and hiding and showing constraints. These sample operations, selected from the operations described in the tutorial at the beginning of this chapter, demonstrate different aspects of inheritance. In Section 6.3.4 we will show how Object-Oriented Relix can achieve the same results. For this purpose in both this section and Section 6.3.2 we provide an outline of the functionality of each method used in the implementation of those operations.

In Objective-C, methods associated with a class are defined inside the class declaration. Therefore, it is logical to list all of the methods that are of interest to us under their corresponding class heading. The `View` and `Button` classes are defined by the NeXT environment. We restrict the following discussion to classes created specifically for Gedit.

Graphics Class

DrawSelf

self Draw;

Move :mouse_down :mouse_up

Calculate the distance moved by subtracting `mouse_down` from `mouse_up`;

Add that distance to the frame origin and to the address of the first vertex;

Perform Display on the whole drawing window;

Select_obj

Set `Select` flag to True;

Postscript routine to display 9 small squares around the perimeter of the frame.

Point Class

Draw

Postscript routine to draw a point;

Circle Class

Draw

Postscript routine to draw a circle;

Poly class

Draw

Postscript routine to draw lines between adjacent vertices in the `ListOfVert`.

If `type` = "G" then draw a line between the last vertex in the list and the first vertex;

Move :mouse_down :mouse_up

Calculate the distance moved by subtracting `mouse_down` from `mouse_up`;

For all the vertices except the first one, add that distance to the address of the vertex;

[**super** move :mouse_down :mouse_up];

Constraints class

DrawSelf

if HIDE = 0 then [**super** drawSelf]

Hide_All

set Hide = True

perform Display on the whole drawing window;

Hide_Selected

set Hide = True

perform Display on the whole drawing window;

Show_All

set Hide = False

perform Display on the whole drawing window;

Generic function

Overlap :RegionOrigin :RegionSize :FrameOrign :FrameSize

If the frame of an object overlaps the region's boundaries,

then return (True)

else return (False)

6.2.3 View Hierarchy

All the graphical objects that are created in the drawing window are linked together in a **View hierarchy**. The View hierarchy is not the same as the inheritance hierarchy. The inheritance hierarchy is an arrangement of classes; the View hierarchy is an ordered arrangement of objects. Objects that are referenced more recently are placed ahead of the objects that are referenced earlier. The View hierarchy also defines a message-passing chain. For example, sending a `Display` message to the root of an aggregation hierarchy will cause every object in the hierarchy (in the drawing window) to be displayed.

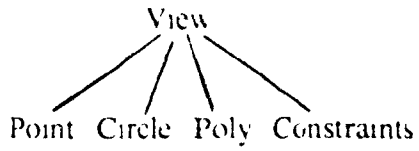


Figure 6.2 The View Hierarchy of Gedit

6.2.4 Objective-C Implementation of the Sample Operations

Now we describe the sequence of commands that execute the operations mentioned in the previous section

6.2.4.1 Creation of an Object

A graphical object of the class selected in the object factory is created when the drawing window receives a "mouse down" event.

Point.

A `firstV` variable is initialized with the coordinates of the "mouse down" event A point is drawn.

Circle

A `firstV` variable is initialized with the coordinates of the "mouse down" event The "mouse up" event determines the radius of the circle. Both the center and circumference of the circle are drawn.

Poly.

Every time a mouse button is clicked, the coordinates of the cursor are inserted into the `ListofVert` list The line between the current and previous vertex is drawn. When the double click event is received, if the object factory selection is a "polygon", the line is drawn between the last and the first vertex in the `ListofVert` list.

When the creation of an object is complete (double click for poly-figures and "mouse up" event for points and circles), the frame of that graphical object is calculated and displayed The object ID and the frame are added to the `Glist`.

6.2.4.2 Selection of Several Objects

The addresses of the left-button "mouse down" and "mouse up" events are temporarily saved. A generic function `Overlap::::` is applied to every element of the `Glist` to identify all the objects whose frame overlaps with the selected region's boundary. `Slist` is cleared and all the objects identified by the `Overlap` function are inserted into the `Slist`. The message `Select_obj` is sent to each object in the `Slist`.

6.2.4.3 Moving Selected Objects

The `move::` message is sent to all the objects in the `Slist`. As a result, the coordinates of all the vertices in those objects are changed, as is the frame origin of those objects.

6.2.4.4 Hiding / Showing Constraints

Execute the procedure corresponding to the Constraint submenu selection.

"Hide All"

Send a `Hide_All` message to every element of the `Clist`. This method sets the `Hide` variable to True for those constraints. Also, the `Display` message is sent to the whole window. `Display` routine does not display icons whose `Hide` variable is True.

"Hide Selected".

Set `Hide` variable to True for the constraint whose `Select` flag is set to True. Send a `Display` message to that constraint.

"Show All"

Send a `Show_All` message to every element of the `Clist`. This method sets the `Hide` flag to False for every constraint. Also the `Display` message is sent to the whole `View`.

6.3 Relix Environment

In Object-Oriented Relix, subclasses inherit both attributes and methods from their superclasses. Methods are inherited in the same way as in Objective-C. On the other hand, attribute inheritance is different: the subclasses do not contain the inherited attributes, but can access them through the link (the object ID field) connecting superclasses with their subclasses. This approach is significantly different from the Objective-C inheritance mechanism, because in Object-Oriented Relix an object which belongs to a leaf class of an inheritance hierarchy spans all the classes above it in the hierarchy. That is, its attributes are distributed among all its superclass relations.

In Object-Oriented Relix, graphical objects are represented by their vertices. So a poly-figure having 4 vertices will have 4 tuples in the `Graphics` class. Thus for constraint satisfaction routines all the vertices are stored in one central place - `Graphics` class.

6.3.1 "Class-Oriented" Relix vs Objective-C

In Relix, methods are invoked on classes rather than on the class instances. Thus a method is applied either to all of the instances of a class or to a selection of class instances. There is no need to deal with lists of objects, as is necessary in Gedit. If a user wants to apply a method only to instances satisfying some conditions, the selection may be performed in two ways:

- by the relational expression invoking the method; or
- by that method itself, which can be written to ignore all the instances that do not satisfy those conditions

Because of the nature of Relix, it is appropriate to execute the same procedures as Gedit but on a log of the Gedit session (see Section 6.3.4). Our task then would be to combine all similar information in corresponding relations and to apply the same method to all tuples in such a relation.

6.3.2 Inheritance Hierarchy

There are two major differences between the class hierarchies in Gedit and Relix. The first is that Relix does not have List classes (`Clist,Slist,Glist`). The second is that Poly-figures in Relix do not have an attribute which represents a list of vertices -- all the vertices with their corresponding sequence numbers are stored in the `Graphics` class. Also a collective class `Figure` was added to the class hierarchy in Relix.

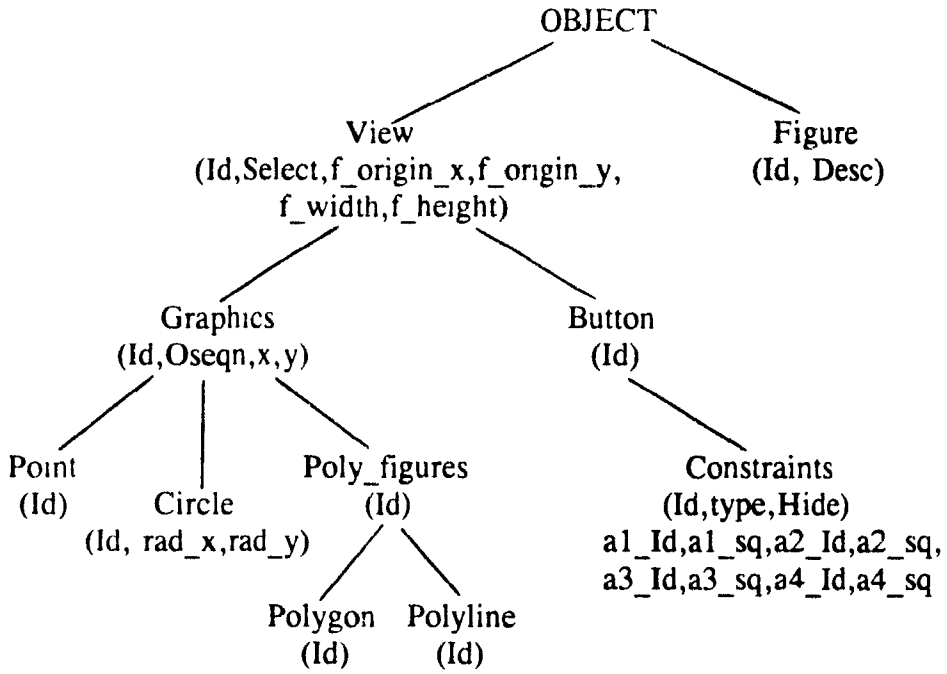


Figure 6.3 Inheritance Hierarchy in the Object-Oriented Relix Version of Gedit.

The meaning of instance variables declared in this hierarchy and different from the Gedit hierarchy are explained as follows:

f_origin_x, f_origin_y - the x and y coordinates of the frame origin;

f_width, f_height - the width and height of the frame;

Oseqn - the sequence number of the vertex within a graphical object;

addr_x, addr_y - the coordinates of the vertex;

rad_x, rad_y - horizontal and vertical radius of a cirle (if the two radii are not the same, it is an elipse);

a1_Id, a1_sq, ... - for each arm of a constraint we store the ID of the graphical object

that contains the vertex to which that arm is connected; the sequence number uniquely identifies the vertex among all the vertices of that object.

The `View` class has one tuple per object. The `Graphics` class has attributes x and y which are the coordinates of a point, and which are interpreted differently in each of the subclasses of `Graphics`. For the `Point` it is the point itself, for the `Circle` it is the center, and for the `Poly-figures` it is their vertex. The `Poly-figures` have two or more vertices. It is important to preserve the sequence in which these vertices were originally created. The `Oseqn` attribute in the `Graphics` class serves that purpose. Thus a `Graphics` class has one tuple for each `Point` or `Circle`, and two or more tuples for each `Poly-figure`.

Below we show a pseudo code for the methods that are going to be used in Section 6.3.5 to perform the four operations. The actual Relix code for these methods and for the operations described in Section 6.3.5 can be found in Appendix D.

In Object-Oriented Relix, the association of methods with classes is achieved by declaring the class in the ***on*** clause of the method declaration. For the better illustration of method inheritance we group methods by their name.

function Display () **on** OBJECT;
    call DrawSelf();

function DrawSelf() **on** Graphics;
    call Draw();

function Draw () **on** Point;
    Postscript routine to draw a Point

function Draw () **on** Circle;
    Postscript routine to draw a Circle

function DrawSelf () **on** Constraints;

if Hide = False **then**

Postscript routine to draw each icon

else

erase icon and arms

function Overlap (orig_x, orig_y, width, height) **on** Graphics;

determine if graphics frame overlaps the rectangle specified in the parameter list.

function Draw_frame () **on** Graphics;

if Select = True **then**

Postscript routine to display 9 small squares around the perimeter of the rectangular frame

else

erase the 9 small squares around the perimeter if they are there.

procedure Move (**in:** dist_x, dist_y; **out:** orig_x, orig_y) **on** View;

Move the frame of an object or a constraint. Since the frame is represented by its origin and its size, it is only necessary to change its origin by the given distance.

procedure Move_obj (**in:** dist_x, dist_y; **out:** new_x, new_y, orig_x, orig_y) **on** Graphics;

Calculate the new address of each vertex by adding the given distance to the coordinates of each vertex.

Call the procedure Move() defined on the superclass to move the frame.

Generic function

function Draw_line (from_x, from_y, to_x, to_y)

Postscript routine to draw a line from point (from_x,from_y) to point (to_x,to_y);

6.3.3 Collection Hierarchy

The collection hierarchy of Object-Oriented Relix shown in Figure 6.4 differs from the collection hierarchy of Gedit (Figure 6.2). The main difference is that it does not keep track of the order in which objects were referenced. It serves uniquely as a distributor of messages. A message sent to a collective class is distributed to every subobject class of that collective class. For example, a `Move_obj` message sent to the `Figure` class will be applied to the subobject classes of `Figure`, namely `Point`, `Circle`, `Polygon`, and `Polyline` classes

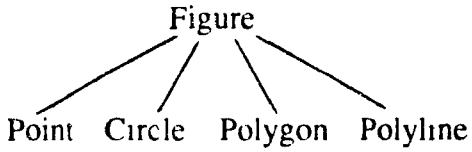


Figure 6.4 Collection Hierarchy in Object-Oriented Relix Implementation of Gedit.

Also, this hierarchy does not include the `Constraints` class. The reason for that is that the functionality of the `Constraints` class is very different from the subclasses of the `Graphics` class. For the four operations discussed in this chapter, we need only to include the subclasses of the `Graphics` class in the collection hierarchy.

6.3.4 Representation of the Gedit Session in Relational Format

In a distributed database there might be a situation in which two users are working on the same database of graphical objects. Suppose that one user in Montreal is using Gedit to create and manipulate graphical objects, and another user in Vancouver needs to know the changes to the screen. The information can be transmitted along the telephone lines in the relational format. Gedit has a logging capability which records every detail about each mouse click such as its location, which button was used (left or right), the type of the mouse click (single or double), and if any dragging has occurred between the "mouse down" and "mouse up" events. Since the order of the mouse events is important in the process of

manipulating graphical objects, the log also generates the relative sequence number of each click. The log contains independent actions so that the order in which these actions are performed is irrelevant.

We illustrate how a Gedit session (consisting of creating objects, selecting objects, moving selected objects, hiding or showing constraints) can be used as a good example of how an object-oriented Relix can achieve the same results as Gedit using the log of this session.

A Graphics editor session can be recorded as a sequence of mouse clicks. The following assumptions simplify the task of correctly identifying the purpose of each mouse click:

- the Primitives panel, drawing window, menu and all the submenus do not overlap;
- the Primitives panel, drawing window, and menu remain immobile during the complete session, so that the location of a mouse click can be unambiguously determined;
- all the mouse clicks are either on the Primitives panel, on the drawing window, or on the menu;
- text creation and manipulation is omitted.

Let us examine the actions performed during the Gedit session described above. All the events are generated by mouse clicks. There are two types of mouse clicks: single and double. A single mouse click consists of consecutive "mouse down" and "mouse up" events. Between those two events a mouse may be dragged. A double mouse click consists of two single mouse clicks made within a short time interval of each other. The left and right button clicks serve different purposes. The right button is used only for creating constraints inside the drawing window and connecting the arms of the constraints. The left button is used for everything else

Different sequences of mouse clicks and mouse drags lead to different results. The address of a mouse click determines whether the event happened on the Primitives panel, on the menu, or on the drawing window.

The log of the Gedit session is stored in the SESSION relation.

SESSION (seqn,x,y,click_type,mouse,drag)

where seqn - the sequence number of the mouse click;

x, y - the coordinates of the cursor at the time of the mouse click;

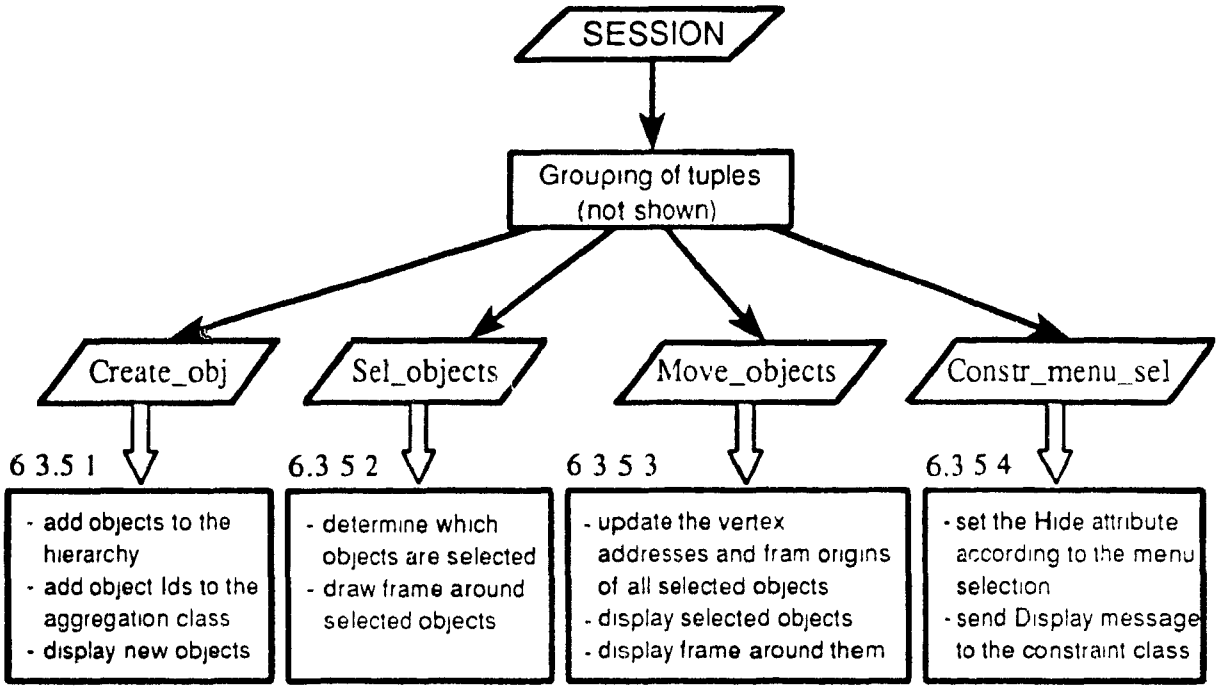
click_type - s for single
d for double;

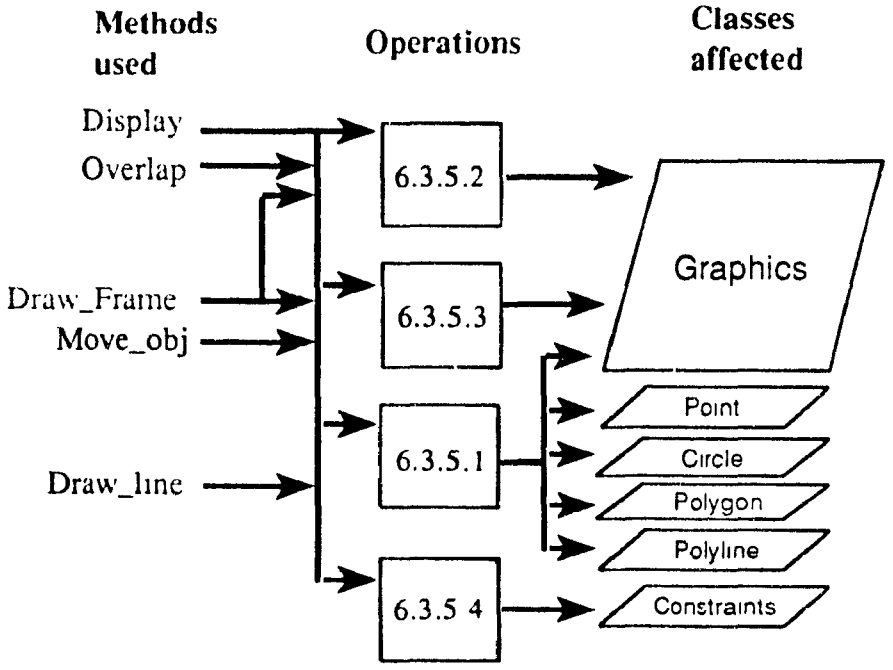
mouse - l for left mouse click
r for right mouse click;

drag - n for no dragging between the mouse clicks
d for mouse down followed by a dragging event
u for the release of mouse after the dragging event.

From the SESSION relation we can obtain the relations `Create_obj`, `Sel_objects`, `Move_objects`, and `Constr_menu_sel`, by grouping the tuples representing operations of creating graphical objects, selecting objects, moving objects and hiding/showing of constraints respectively. This grouping can be done in the current implementation of Relix Since it does not require the object oriented features presented in this thesis, this procedure is not discussed here.

The complete process of using the log of the Gedit session is illustrated by the flowchart below.





Below we describe the four relations that are extracted from the SESSION relation

1. Creation of graphical objects.

Create_obj (Id,Oseqn,type,x,y,f_orig_x,f_orig_y,f_width,f_height)

where Oseqn - sequence number of the vertex within an object;

type - c for circle
p for point
g for polygon
l for polyline;

x, y - the coordinates of the vertex if the figure is a point, polygon or a polyline. If the figure is the `Circle`, for `Oseqn` = 1 it is the center, and for `Oseqn` = 2 it is the radius in each direction;

f_orig_x - the coordinates of the left bottom corner,

f_orig_y of the rectangular box (frame) that surrounds each object,

f_width - the width of the frame;

f_height the height of the frame.

In this relation `Points` are represented by one tuple. `Circles` are represented by two tuples. The tuple with `Oseqn` = 1 contains the coordinates of the center, and the tuple with `Oseqn` = 2 contains the radius of the circle in both x and y direction. `Poly-figures` are represented by N tuples where N is the number of line segments that make up that poly-figure.

2. Selection of several objects

Sel_Objects (r_orig_x,r_orig_y, r_width,r_height)

where r_orig_x, r_orig_y - coordinates of the bottom left corner of the selected region;

r_width, r_height - width and height of the rectangular region.

3. Moving selected objects

Move_objects (dist_x, dist_y)

where dist_x - the absolute horizontal distance travelled by the mouse;

dist_y - the absolute vertical distance travelled by the mouse.

4 Hiding/showing of constraint(s)

Constr_menu_sel (menu_sel)

where `menu_sel` is the option of the Constraint submenu chosen.

6.3.5 Object-Oriented Relix Implementation of the Sample Operations

In this section we will describe the procedure for executing the four Gedit operations in object-oriented Relix. Each procedure consists of relational and domain algebra statements which use the relations extracted from the SESSION, classes of the inheritance hierarchy, and the methods defined on those classes.

6.3.5.1 Creation of Objects

Given: Create_obj (Id,Oseqn,type,x,y,f_orig_x,f_orig_y,f_width,f_height)

The process of creating graphical objects consists of placing them in the appropriate classes of the inheritance hierarchy and displaying them. Each class should be treated differently. Creation of objects is the very first operation of the session. At this point the class hierarchy is empty

When adding an object to the leaf class of the inheritance hierarchy, we should provide the values of all the attributes (both inherited and defined in the leaf class itself). This can be achieved by an update statement.

Point

Add to the `Point` class all the relevant information from those tuples of the `Create_obj` relation where `type` = "P".

Circle.

First, add to the `Circle` class coordinates of the center and frame specifications from those tuples of the `Create_obj` relation where `type` = "C" and `Oseqn` = 1. Second, record the radius in the `Circle` class by changing the value of the `rad_x` and `rad_y`

attributes to x and y attributes from the tuples of the Create_obj relation where type = "C" and Oseqn = 2.

Poly-figures.

Add to the Polygon and Polyline classes all the relevant information from those tuples of Create_obj relation where type = "G" or "L" respectively.

At this point the Graphics branch of the inheritance hierarchy is complete. Now we need to place all the objects into the collection class. It is done by adding all the object IDs in the Create_obj relation to the Figure collective class.

Send a Display message to the Figure class to display new objects. That takes care of displaying points and circles. To display poly-figures, invoke a Draw_line procedure for every pair of adjacent vertices.

6.3.5.2 Selection of Several Objects

Given: Sel_objects (r_orig_x,r_orig_y,r_width,r_height)

Selection of objects is always associated with another operation such as moving or zooming. It is essential to preserve the order in which selection, moving and zooming operations occur. The same sequence number will be assigned to the selection and to the operation that immediately follows the selection and operates on the selected objects. The selection of all the objects in the region consists of these procedures:

1. In the Graphics class, set the Select attribute to the value returned by the function Overlaps, defined on the Graphics class. The Overlaps function determines if the frame of the object overlaps the frame of the selected region,
2. Invoke the Draw_frame method in the Graphics relation to display the frame around the selected objects; when this method is applied to objects with the Select attribute set to False, nothing happens; thus by sending the Draw_frame message to all the objects in the Figure, the desired effect is achieved.

6.3.5.3 Moving Selected Objects

Given: Move_objects (dist_x, dist_y)

Selected objects are moved by invoking method `Move_obj` on the `Graphics` class. Method `Move_obj`, defined on the `Graphics` class, changes the addresses of the vertices of those objects whose `Select` attribute is set to True. Then the frame's origin is moved by invoking method `Move` defined on the `View` class. When both methods are applied to objects with the `Select` attribute set to False, nothing happens. Thus by sending the `Move` message to all objects in the `Graphics` class, only selected objects are moved.

6.3.5.4 Hiding Constraints / Showing Constraints

Given: Constr_menu_sel (menu_sel)

The `menu_sel` attribute has one of the following values: "Hide All", "Show All", "Hide Selected".

Hiding and showing of constraints is achieved by setting the `Hide` attribute of the Constraints class to True or False respectively, and then sending a `DrawSelf` message to the `Constraints` class. The `DrawSelf` method is coded in such a way that it does not display the constraints whose `Hide` attribute is True.

"Hide All":

Set the `Hide` attribute to True for every constraint;

"Hide Selected".

Set the `Hide` attribute to True for the constraint whose `Select` attribute is set to True;

"Show All":

Set the `Hide` attribute to False for every constraint; determine the the coordinates of end points of all connected arms for all constraints; call a `Draw_line` method to draw these arms.

Chapter 7

Conclusion

In this thesis we have shown that the relational database model has the power to support the most important object-oriented features, namely, inheritance of attributes and methods, and method polymorphism. We have derived these features from the data structure and operations of the classical relational model. This work is based on the assumption that a relation in a relational DBMS is equivalent to a class in an object oriented DBMS.

We formalized the definition of these object oriented features. This enables a database programmer to understand them operationally, in terms of the well known relational algebra.

7.1 Summary

The class hierarchy is represented by a relation where each superclass-subclass pair is represented by a tuple. Attribute inheritance is implemented by projecting the superclass's attributes from the join of the subclass with its superclass on the ID field. Objects, which span one or more tuples, are uniquely identified by their ID field.

Methods can be either generic (applicable to any relation) or specific to a class, i.e., applicable only to that class or its subclasses. The information about the association of methods and classes is stored in a relational format. The mechanism of method inheritance is implemented by combining the information on method association with the inheritance hierarchy to determine which subclasses are eligible to use those methods.

The collection hierarchy determines if a message sent to a class should be passed on to the classes that make up that collective class.

As a result of this design, the following new syntax was added to the set of operations available to the user in Object-Oriented Relix:

New Syntax	Page
`Class2 [Id2 isa Id1] Class1`	50
`Class3 hasa Class4`	80
`function func_name (input_attr_list) on Class1`	72
`let virtual_attr be super Method();`	74
`let virtual_attr be sub Method();`	83

Chapter 6 illustrates how the features presented in this thesis can be used in an object-oriented application. The first part of that chapter familiarizes the reader with an object-oriented application, Gedit, which is written in Objective-C. The second part of that chapter proposes how some of the features of Gedit may be implemented in Object-Oriented Relix. It is important to keep in mind that this thesis is not based on an actual implementation, but is of a pure analytical nature. The implementation which is proposed in the appendices has not been tested.

7.2 Further Work

1. Chapter 3 mentioned that the ID fields are generated by the system, but did not specify how this was accomplished. The ID generation procedure should keep track of the existing ID values so that newly-generated values are unique. This procedure should make extensive use of the `.Hierarchy` system meta-relation to guide it in determining when to generate new ID values and when to assign the corresponding values of the parent's ID. This is significant for consistency checks, and for verification of the existence of orphans.

2. Chapter 3 concentrated on simple inheritance. Multiple inheritance is more powerful than simple inheritance. To implement multiple inheritance, some algorithms (like method broadcasting) would have to be redesigned.

3. Can the design described in Chapter 3, which represents the ISA relationships between classes in the inheritance hierarchy, be adopted for the representation of nested relations?

4. Chapter 4 discussed polymorphic functions and procedures which can be declared on specific relations. Before this feature can work, it is necessary to implement functions and procedures in Relix. This would require further research on type theory.

5. Chapter 4 also described how methods are invoked on relations and produce one or more values for each tuple of that relation. It is desirable to have another type of method which would work with relations as a whole, like the procedure `Join_Hierarchy` in Appendix A.

6. In this thesis we concentrated on attribute and method inheritance. No object-oriented database can be complete without other important features, like abstract data types (ADT), encapsulation, and information hiding.

Appendix A

Attribute Inheritance

The appendices of this thesis use a system relation `.rd (.rel_name,.dom_name)`. This relation contains information about all the relations in the database. It is maintained by the system and is updated every time a relation is created or deleted. For each relation the system table `.rd` contains as many tuples as there are attributes in that relation. So for example, the `.Hierarchy` relation is represented by four tuples: {`.Hierarchy,.Subclass`}, {`.Hierarchy,.SubId`}, {`.Hierarchy,.SuperId`}, {`.Hierarchy,.Superclass`}.

The actualization of a virtual attribute in an operand relation is determined by the following conditions:

- If the requested attribute is found in the definition of the operand relation (there is a tuple for that attribute in the .rd system relation), then that attribute is used;
- if the operand relation does not have that attribute, the attribute is calculated (actualized) according to its domain algebra definition.

The Null values are represented in Relix as Don't Care (DC).

The following system relations are used to illustrate the Relix code presented in this appendix:

.Hierarchy

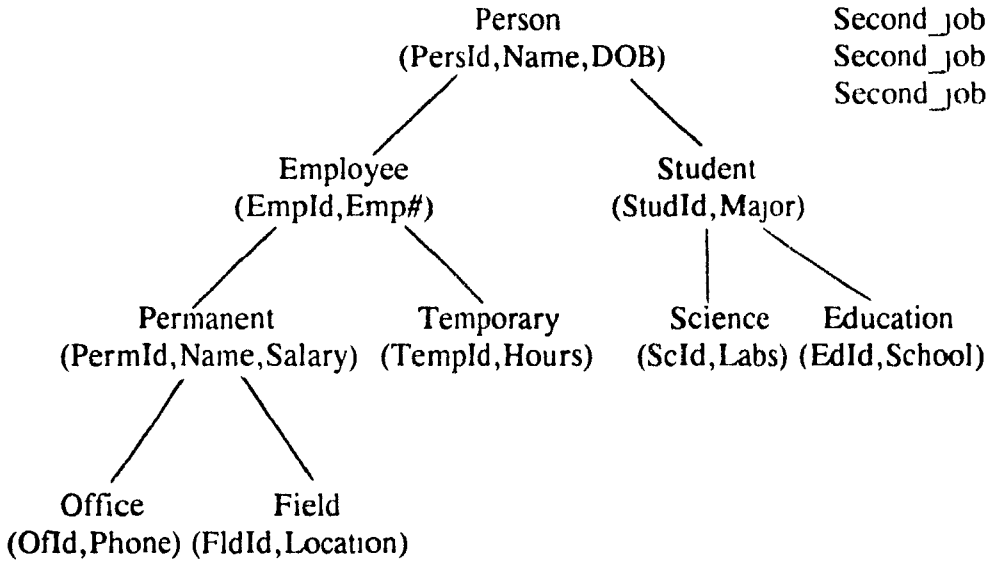
(.Subclass,	.SubId,	.Superclass,	.SuperId)
Employee	EmpId	Person	PersId
Permanent	PermId	Employee	EmpId
Office	OfId	Permanent	PermId
Field	FldId	Permanent	PermId
Temporary	TempId	Employee	EmpId
Student	StudId	Person	PersId
Science	ScId	Student	StudId
Education	EdId	Student	StudId
Pay_errors	TempId	Corrections	CorrId

.rd

(.rel_name,	.dom_name)
Person	PersId
Person	Name
Person	DOB
Employee	EmpId
Employee	Emp#
Permanent	PermId
Permanent	Salary
Permanent	Name
Office	OfId
Office	Phone
Field	FldId
Field	Location
Temporary	TempId
Temporary	Hours
Student	StudId
Student	Major
Science	ScId
Science	Labs
Education	EdId
Education	School
Corrections	CorrId
Corrections	Amount
Corrections	Date
Pay_errors	TempId
Pay_errors	Pay#
Second_job	TempId
Second_job	Name
Second_job	DOB
Second_job	Emp#
Second_job	Hours

Corrections
(CorrId,Amount,Date)

Pay_errors
(TempId,Pay#)



A.1 Inherit Statement

This detailed explanation corresponds to the algorithm in section 3.4.

The inherit statement is parsed and four scalar variables are initialized: `Child`, `CId`, `PId`, and `Parent`.

In order to insert a tuple with the above values into the `.Hierarchy` relation, we need to use an ADD clause of the update statement. The ADD clause requires that the relation being added has the same attributes as the relation being updated. In other words, since `.Hierarchy`relation has attributes `.Subclass`, `.SubId`, `.SuperId`, and `.Superclass`, we need to find a way of referring to the four input scalars under the names of the `.Hierarchy` attributes. This can be achieved by renaming `.Hierarchy` attributes to their corresponding input scalars and projecting these virtual attributes from any relation that does not have any of the `.Hierarchy` attributes, say `.rd`.

if Parent = "ROOT" **then**
    update .Hierarchy **delete** (**where** .Subclass = Child);
else
    if Child relation is empty **then**
        let .Superclass **be** Parent;
        let .Subclass **be** Child;
        let .SubId **be** Cld;
        let .SuperId **be** Pld;
        update .Hierarchy **add** ([.Subclass,.SubId,.SuperId,.Superclass] **in** .rd);
    else
        error message "Cannot insert a non-empty relation into a class hierarchy".

A.2 Projection and Selection

This detailed description corresponds to the algorithm in section 3.6.

Illustration.

Evaluate an expression

New_rel ← [Name,DOB,Emp#] **where** Salary = 25 **in** Office;

■

The input expression is parsed and its elements are stored in several relations that are going to be used by the system to generate an executable expression:

- all the attributes between square brackets and in the ***where*** clause are stored in a relation `Proj_List (attr)`;
- the name of the relation (R) in the ***in*** clause is stored in the relation `Join_rels (In_rel,Rel_ordr)` with `Rel_ordr` = 1;
- the part of the input expression excluding the contents of the ***in*** clause is stored in a scalar variable `non_relational_part`.

Illustration.

Proj_List (attr)	Join_rels (In_rel, Rel_ordr)	
Name	Office	1
DOB		
Emp#		
Salary		

`non_relational_part` = "New_rel ← [Name,DOB,Emp#] where Salary = 25 in "

■

We have broken this process into three procedures (`Join_Hierarchy`, `Minimum_Join`, and `Find_Classes_in_Branch`) that call each other. This break down enables other processes in this appendix to use one or more of these procedures. The calling sequence of these procedures is shown below:

`Join_Hierarchy` calls `Minimum_Join` calls `Find_Classes_in_Branch`

The procedures `Join_Hierarchy`, `Minimum_Join`, and `Find_Classes_in_Branch` are shown below. They are illustrated for the input expression of section A.2.

The complete process of projecting and selecting inherited attributes is presented on page 139.

procedure Find_Classes_in_Branch()

≪ This procedure finds all the superclasses of the relations in the `Join_rels` table. The
≪ relation `Classes_in_Branch` contains the following information:
≪ .Superclass - the name of the superclass relation
≪ .SuperId - the ID field of the superclass relation
≪ In_rel - relation for which the superclasses are determined
≪ RelId - the ID field of the `In_rel` relation
≪ Rel_ordr - order in which current `In_rel` appears in the input expression
≪ ordr - order in which superclasses are found. The lower the order, the more
≪ closely related is the superclass to the `In_rel` relation. The highest order
≪ indicates the root of the hierarchy to which `In_rel` belongs.

{

≪ To insert the first set of tuples (one for each relation in the `Join_rels` table) into
≪ the `Classes_in_Branch` relation, we redefine `.Superclass` as `In_rel` and `ordr` as
≪ `Rel_ordr`. The other attributes are unknown at this time, so they will have the
≪ value DC.

let .Superclass **be** In_rel; **let** .SuperId **be** DC; **let** RelId **be** DC;
let ordr **be** Rel_ordr;
Classes_in_Branch ← [.Superclass,.SuperId,In_rel,RelId,Rel_ordr,ordr] **in** Join_rels;

Illustration.

Classes_in_Branch	(.Superclass,	.SuperId,	In_rel,	RelId,	Rel_ordr,	ordr)
	Office	DC	Office	DC	1	1

■

≪ To find the immediate superclass of each input relation, we join `Join_rels` with the
≪ `.Hierarchy` system relation. The `ordr` attribute is calculated by incrementing the
≪ total count of tuples in the intermediate relation.

let RelId **be** .SubId;
let ordr **be** (**red** + **of** 1) + 1;
Classes_in_Branch < + [.Superclass,.SuperId,In_rel,RelId,Rel_ordr,ordr] **in**
Join_rels [In_rel **ijoin** .Subclass] .Hierarchy;

Illustration.

Classes_in_Branch (.Superclass,	.SuperId,	In_rel,	RelId,	Rel_ordr,	ordr)
Office	DC	Office	DC	1	1
Permanent	PermId	Office	OfId	1	2

■

≪ Recursively find the other superclasses. Notice that the attribute `RelId` exists in the
≪ `Classes_in_Branch` relation, and therefore that its value will be taken from the
≪ `Classes_in_Branch` relation rather than from `.Hierarchy`'s `.SubId`, as it was in
≪ the previous step.

Classes_in_Branch **is** Classes_in_Branch **ujoin**
([.Superclass,.SuperId,In_rel,RelId,Rel_ordr,ordr]**in**Classes_in_Branch
[.Superclass,.SuperId **icomp** .Subclass,.SubId] .Hierarchy);

Illustration.

Classes_in_Branch (.Superclass,	.SuperId,	In_rel,	RelId,	Rel_ordr,	ordr)
Office	DC	Office	DC	1	1
Permanent	PermId	Office	OfId	1	2
Employee	EmpId	Office	OfId	1	3
Person	PersId	Office	OfId	1	4

■

}

procedure Minimum_Join ()

≪ This procedure finds all the superclasses of the relations in the input expression and
≪ selects only those classes which contain the attributes of the `Proj_List` table.

{

Find_Classes_in_Branch();

Classes_with_attr ← [.Superclass,.SuperId,In_rel,RelId,Rel_ordr,ordr,attr] **in**

.rd [.rel_name,.dom_name **ijoin** .Superclass,attr] (Classes_in_Branch **ujoin** Proj_List);

Illustration.

Classes_with_attr (.Superclass, .SuperId, In_rel, RelId, Rel_ordr, ordr, attr)

Permanent	PermId	Office	OfId	1	2	Name
Permanent	PermId	Office	OfId	1	2	Salary
Employee	EmpId	Office	OfId	1	3	Emp#
Person	PersId	Office	OfId	1	4	Name
Person	PersId	Office	OfId	1	4	DOB

■

≪ Ensure that if there are any renamed attributes in the specialized classes, those
≪ attributes are projected instead of the attributes with the same name in the more
≪ general classes. This involves determining from the `Classes_with_attr` relation
≪ if any attribute is defined in more than one class. We want to refer only to the
≪ more specialized attributes, so we select those tuples which have the lowest `ordr`.

let attr_count **be equiv** + **of** 1 **by** In_rel,attr;
let lowest_rel **be equiv min of** ordr **by** In_rel,attr;
let conflict_attr **be if** (attr_count > 1 **and** ordr > lowest_rel) **then** 1
else 0;

≪ Special treatment is required when the input expression contains references to
≪ *several* attributes of a general class (`Name,DOB` in `Person`), some of which are also
≪ defined in a more specialized class (`Name` in `Permanent`). The special treatment
≪ involves including in the join expression, generated by the system, the projection of

```
≪ non-renamed attributes from the general class, in order to allow unambiguous
≪ projection of the renamed attributes from the more specialized class. The attribute
≪ proj_attr_list is generated from the Classes_with_attr relation and takes care
≪ of necessary projections from general classes.

let list_of_attr be equiv max of (par cat of ("," cat attr) order ordr
                    by.Superclass,In_rel,conflict_attr)by.Superclass,In_rel,conflict_attr;
let conflict_rel be equiv max of conflict_attr by .Superclass,In_rel;
let proj_attr_list be if conflict_rel = 1 then
                    " ([" cat .SuperId cat list_of_attr cat "] in " cat .Superclass cat ")) "
      else .Superclass cat ") ";
Necessary_Classes ← [.Superclass,.SuperId,In_rel,RelId,Rel_ordr,ordr,proj_attr_list]
                    where (attr_count = 1 or ordr = lowest_rel) in Classes_with_attr;
```

Illustration.

Classes_with_attr
(.Superclass, .SuperId,In_rel, RelId,Rel_ordr,ordr,attr) attr count | lowest rel | conflict attr

Permanent	PermId	Office	OfId	1	2	Name	2	2	0
Permanent	PermId	Office	OfId	1	2	Salary	1	2	0
Employee	EmpId	Office	OfId	1	3	Emp#	1	3	0
Person	PersId	Office	OfId	1	4	Name	2	2	1
Person	PersId	Office	OfId	1	4	DOB	1	4	0

Necessary_Classes
(.Superclass, .SuperId,In_rel, RelId,Rel_ordr,ordr, proj_attr_list)

Permanent	PermId	Office	OfId	1	2	"Permanent) "
Employee	EmpId	Office	OfId	1	3	"Employee) "
Person	PersId	Office	OfId	1	4	"([PersId,DOB] in Person) "

■

```
≪ We will not need the tuples with the input relation names in the .Superclass
≪ attribute.

update Necessary_Classes delete (where .SuperId = DC in Necessary_Classes);
}
```

```
procedure Join_Hierarchy ()

≪ Build a Necessary_Classes relation which contains only those superclasses of the
≪ relation(s) participating in the input expression, which define the attributes from the
≪ Proj_List. Use the Necessary_Classes and the Proj_List tables to generate join
≪ expression(s) which will be used to replace the contents of the in clause. Store the
≪ generated expression(s) in the relation Expanded_hierarchy.
{
  Minimum_Join ();

  ≪ To build a join expression we work with the Necessary_Classes relation. Each
  ≪ tuple is used to provide information of what class is joined on what ID field. The
  ≪ intermediate attributes (before and after) are designed in such a way that, when
  ≪ concatenated together, they produce the required expression.

  let min_index be equiv min of ordr by In_rel;
  let before be
   if ordr = min_index then
     In_rel cat " [" cat RelId cat " ijoin " cat .SuperId cat "] " cat .Superclass cat ") "
   else    .SuperId cat "] " cat .Superclass cat ") ";
  let max_index be equiv max of ordr by In_rel;
  let after be if ordr = max_index then " "
                 else                    "[" cat .SuperId cat " ijoin ";
  let int_expr be before cat proj_attr_list cat after;
  let brackets be equiv max of (fun cat of "(" order ordr) by In_rel;
```

Illustration.

Necessary_Classes

(.Superclass,	.SuperId,	In_rel,	RelId,	Rel_ordr,	ordr,	proj_attr_list)
Permanent	PermId	Office	OfId	1	2	"Permanent) "
Employee	EmpId	Office	OfId	1	3	"Employee) "
Person	PersId	Office	OfId	1	4	"([PersId,DOB] in Person) "

.. continued from the above relation.

.Superclass	min_index	before	max_index	after
Permanent	2	"Office [OffId ijoin PermId] "	4	"[PermId ijoin "
Employee	2	"EmpId] "	4	"[EmpId ijoin "
Person	2	"PersId] "	4	" "

.. continued from the above relation.

.Superclass	brackets	int_expr
Permanent	"((("	"Office [OfId ijoin PermId] Permanent) [PermId ijoin "
Employee	"((("	"EmpId] Employee) [EmpId ijoin "
Person	"((("	"PersId] ([PersId,DOB] in Person)) "

■

≪ The last step of joining the hierarchy involves generating the final join of all the

≪ necessary classes that will replace the contents of the ***in*** clause of the input

≪ expression.

let expression **be equiv max of** (**par cat of** int_expr **order** ordr **by** In_rel) **by** In_rel;

let string_stmt **be** brackets **cat** expression;

Expanded_hierarchy ← [In_rel,Rel_ordr,string_stmt] **in** Necessary_Classes;

Illustration.

Expanded_hierarchy
(In_rel, Rel_ordr, string_stmt)

Office	1	"(((Office [OfId ijoin PermId] Permanent) [PermId ijoin EmpId] Employee) [EmpId ijoin PersId] ([PersId,DOB] in Person)) "

■

}

Process of projecting and selecting inherited attributes

1. Determine all the superclasses of the relation R and select among them only those superclasses which define the attributes referenced in the input expression. Generate a join expression from these selected classes and store it in the relation `Expanded_hierarchy`. These operations are performed in the procedure `Join_Hierarchy` which is described on page 137.

```
Join_Hierarchy ();
```

2. In the input expression replace R with the join expression generated in the previous step. Evaluate the input expression.

```
let exec_stmt be (stmt (non_relational_part cat string_stmt cat ";"));
S ← [exec_stmt] in Expanded_hierarchy;
S;
```

Illustration.

The expression in the relation S is intended to be as follows:

```
New_rel ← [Name,DOB,Emp#] where Salary = 25 in
            (((Office [OfId ijoin PermId] Permanent) [PermId ijoin EmpId]
             Employee) [EmpId ijoin PersId] ([PersId,DOB] in Person)) ;
```

■

A.3 Join

This detailed description corresponds to the algorithm in section 3.8.

Illustration.

Evaluate an expression

New_rel ← [Name,Emp#,School] **where** Hours > 25 **in**

Temporary [TempId **ijoin** EdId] Education;

■

The input expression is parsed and its elements are stored in several relations that are going to be used by the system to generate an executable expression:

- all the attributes between square brackets (includes projected and join attributes) and in the ***where*** clause are stored in a relation `Proj_List (attr)`;
- the names of the relations participating in the join are stored in `Join_rels (In_rel,Rel_ordr)`;
- the part of the input expression excluding the ***in*** clause is stored in a scalar variable `non_relational_part`;
- the information about the type of join following each relation in the ***in*** clause (which might include join attributes) is stored in the relation `Join_symbol (symbol,Rel_ordr)`, where `Rel_ordr` is the order of the relation in the ***in*** clause after which the `symbol` is found.

Illustration.

Proj_List (attr)	Join_rel (In_rel,	Rel_ordr)	Join_symbol (symbol,	Rel_ordr)
Name	Temporary	1	" [TempId ijoin EdId] "	1
Emp#	Education	2		
TempId				
EdId				
Hours				
School				

non_relational_part = "New_rel ← [Name,Emp#,School] where Hours > 25 in "

■

1. Build a `Necessary_Classes` relation which contains only those superclasses of the relations in the `Join_rels` table in which all of the attributes from the `Proj_List` are defined. Construct a join expression from the classes listed in the `Necessary_Classes` and store it in the relation `Expanded_hierarchy`. These operations are performed in the procedure `Join_Hierarchy` on page 137.

```
Join_Hierarchy ();
```

Illustration.

Classes_in_Branch (.Superclass, .SuperId,In_rel, RelId, Rel_ordr,ordr)

Temporary	DC	Temporary	DC	1	1
Employee	EmpId	Temporary	TempId	1	2
Person	PersId	Temporary	TempId	1	3
Education	DC	Education	DC	2	2
Student	StudId	Education	EdId	2	3
Person	PersId	Education	EdId	2	4

Classes_with_attr
(.Superclass, .SuperId,In_rel, RelId, Rel_ordr,ordr,attr) attr count, lowest rel, conflict attr

Temporary	DC	Temporary	DC	1	1	TempId	1	1	0
Temporary	DC	Temporary	DC	1	1	Hours	1	1	0
Employee	EmpId	Temporary	TempId	1	2	Emp#	1	2	0
Person	PersId	Temporary	TempId	1	3	Name	1	3	0
Education	DC	Education	DC	2	2	EdId	1	2	0
Education	DC	Education	DC	2	2	School	1	2	0
Person	PersId	Education	EdId	2	4	Name	1	4	0

Necessary_Classes
(.Superclass, .SuperId,In_rel, RelId, Rel_ordr,ordr)

Employee	EmpId	Temporary	TempId	1	2
Person	PersId	Temporary	TempId	1	3
Person	PersId	Education	EdId	2	4

Expanded_hierarchy
(In_rel, Rel_ordr, string_stmt)

Temporary	1	"((Temporary [TempId ijoin EmpId] Employee) [EmpId ijoin PersId] Person) "
Education	2	"(Education [EdId ijoin PersId] Person) "

■

≪ 2. Insert the join symbols between generated expressions.

```
let complete_stmt be if symbol ~= DC then string_stmt cat " " cat symbol cat " "
                        else string_stmt;
let final_stmt be red max of (fun cat of complete_stmt order Rel_ordr);
let exec_stmt be (stmt (no_relational_part cat final_stmt cat ";"));
S ← [exec_stmt] in Expanded_hierarchy ujoin Join_symbol;
S;
```

Illustration.

The following statement is executed:

```
New_rel ← [Name,Emp#,School] where Hours > 25 in
        ((Temporary [TempId ijoin EmpId] Employee) [EmpId ijoin PersId] Person)
          [TempId ijoin EdId] (Education [EdId ijoin PersId] Person);
```

■

A.4 Update

This detailed description corresponds to section 3.10.

A.4.1 Update Operation With Change Clause

Illustration.

Evaluate expression

```
update Temporary change Hours ← Amount, Emp# ← Pay#, DOB ← Date
                        using Pay_errors;
```

■

The input expression is parsed and its elements are stored in several relations that are going to be used by the system to generate an executable expression:

- the name of the relation being updated and the name of the relation in the ***using*** clause are stored in the relation `Join_rels (In_Rel,Rel_ordr)`;
- all the attributes referenced in the ***using*** clause and the attributes on the right hand side of the "←" symbol in the ***change*** clause are stored in the relation `Proj_List (attr)`;
- the attributes being updated (on the left hand side of the "←" symbol) and the attributes from which they are assigned values (on the right hand side of the "←" symbol) are stored in a relation `Change_update_list (attr,value,attr_ordr)`, where `attr_ordr` indicates the order of attr-value pairs in the update expression.

Illustration.

Proj_List (attr)	Join_rels (In_rel,	Rel_ordr)	Change_update_list (attr,	value,	attr_ordr)
Amount	Temporary	1	Hours	Amount	1
Pay#	Pay_errors	2	Emp#	Pay#	2
Date			DOB	Date	3

■

≪ 1. Find the superclasses of all the relations in the input statement and store them in the
≪ `Classes_in_Branch` relation. Choose the relations in the `Classes_in_Branch` on
≪ which the attributes in the `Proj_List` are defined, and store those relations in the
≪ `Necessary_Classes` relation. Notice that `Necessary_Classes` does not contain the
≪ relation being updated or any of its superclasses since the attributes in the `Proj_List`
≪ are from the relation in the ***using*** clause. Construct a join expression from the
≪ classes listed in the `Necessary_Classes` and store it in the relation
≪ `Expanded_hierarchy`. These operations are performed in the procedure
≪ `Join_Hierarchy` presented on page 137.

Join_Hierarchy ();

Illustration.

Classes_in_Branch (.Superclass, .SuperId,In_rel, RelId, Rel_ordr,ordr)

Temporary	DC	Temporary	DC	1	1
Employee	EmpId	Temporary	TempId	1	2
Person	PersId	Temporary	TempId	1	3
Pay_errors	DC	Pay_errors	DC	2	2
Corrections	CorrId	Pay_errors	TempId	2	3

Classes_with_attr

(.Superclass, .SuperId,In_rel, RelId, Rel_ordr,ordr,attr) attr count, lowest rel, conflict attr

Pay_errors	DC	Pay_errors	DC	2	2	Pay#	1	2	0
Corrections	CorrId	Pay_errors	TempId	2	3	Amount	1	3	0
Corrections	CorrId	Pay_errors	TempId	2	3	Date	1	3	0

Necessary_Classes

(.Superclass, .SuperId,In_rel, RelId, Rel_ordr,ordr, proj_attr_list)

Corrections	CorrId	Pay_errors	TempId	2	3	"Corrections) "

Expanded_hierarchy

(In_rel, Rel_ordr, string_stmt)

Pay_errors	2	"(Pay_errors [TempId ijoin CorrId] Corrections) "

■

≪ 2. Generate and evaluate a join of the necessary superclasses of the relation in the
≪ ***using*** clause. It is important to include in that join the relation being updated (R)
≪ so that later when the superclasses of R are updated, only tuples with offsprings in
≪ R are updated. The join expression is assigned to a relation `Temp_Rel`, which will be
≪ used in the ***using*** clauses of all the generated ***update*** statements. This increases
≪ the efficiency of the evaluation of those statements because this join expression will
≪ be evaluated only once.

```
let final_expr be "Temp_Rel ← " cat string_stmt cat " ijoin " cat In_rel cat ";";
let Stmt_final be (stmt final_expr);
S ← [Stmt_final] in (([string_stmt] in Expanded_Hierarchy) ujoin
                    ([In_rel] where Rel_ordr = 1 in Join_rels);
S;
```

Illustration.

Temp_rel ← (Pay_errors [TempId **ijoin** CorrId] Corrections) **ijoin** Temporary;

■

≪ 3. Determine which classes are affected by the input update expression (i.e. among the
≪ class being updated and its superclasses find the ones on which the attributes in the
≪ `Change_update_list` are defined). The superclasses of the relation being updated
≪ have been determined in the `Join_Hierarchy` procedure and are stored in the
≪ `Classes_in_Branch` relation.

Update_Classes ← [.Superclass,.SuperId,RelId,attr,value,attr_ordr] **in**
rd [rel_name,.dom_name **ijoin** .Superclass,attr]
((**where** Rel_ordr = 1 **in** Classes_in_Branch) **ujoin** Change_update_list);

Illustration.

Classes_in_Branch (.Superclass, .SuperId,In_rel, RelId, Rel_ordr,ordr)

Temporary	DC	Temporary	DC	1	1
Employee	EmpId	Temporary	TempId	1	2
Person	PersId	Temporary	TempId	1	3
Pay_errors	DC	Pay_errors	DC	2	2
Corrections	CorrId	Pay_errors	TempId	2	3

Update_Classes

(.Superclass,	.SuperId,	RelId,	attr,	value,	attr_ordr)
Temporary	DC	TempId	Hours	Amount	1
Employee	EmpId	TempId	Emp#	Pay#	2
Person	PersId	TempId	DOB	Date	3

■

≪ 4. For each class, determine the list of assignments to be placed in the ***change*** clause.
≪ This is done by concatenating all the attr-value pairs defined on the same relation.

let comma **be if** attr_ordr = (**equiv max of** attr_ordr **by** .Superclass) **then** " " **else** ",";
let list_of_updates **be equiv max of** (**par cat of** (attr **cat** " ← " value **cat** comma)
order attr_ordr **by** .Superclass) **by** .Superclass;

Illustration.

Update_Classes

(.Superclass,	.SuperId,	RelId,	attr,	value,	attr_ordr)	comma	list_of_updates
Temporary	TempId	TempId	Hours	Amount	1	" "	"Hours ← Amount "
Employee	EmpId	TempId	Emp#	Pay#	2	" "	"Emp# ← Pay# "
Person	PersId	TempId	DOB	Date	3	" "	"DOB ← Date "

■

≪ 5. Generate an executable update expression for each relation in the `Update_classes`.

≪ Concatenate generated statements one after another in the same order as their

≪ attributes appear in the ***change*** clause of the input update statement.

```
let USING_join_attr be if SuperId = DC or .SuperId = RelId then " "
                       else " [" cat .SuperId cat " join " cat RelId cat "] ";
let update_expr be "update " cat .Sup cat " change " cat list_of_updates cat " using "
                       cat USING_join_attr cat Temp_Rel cat ",";
let update_ordr be equiv min of attr_ordr by .Superclass;
let ready_update_expr be (stmt (red max of (fun cat of update_expr order update_ordr)));
S ← [ready_update_expr] in Update_Classes;
S;
```

Illustration.

The following statements are executed:

update Temporary **change** Hours ← Amount **using** Temp_Rel;

update Employee **change** Emp# ← Pay# **using** [EmpId **ijoin** TempId] Temp_Rel;

update Person **change** DOB ← Date **using** [PersId **ijoin** TempId] Temp_Rel;

■

A.4.2 Update Operation With Add or Delete Clause

Illustration.

Given a relation `Second_job (TempId,Name,DOB,Emp#,Hours)`. Evaluate the following:

update Temporary **add** Second_job;

■

The input expression is parsed and its elements are stored in several relations that are later used by the system to generate an executable expression:

- the name of the relation being updated is stored in the relation `Join_rels (In_rel,Rel_ordr)` with `Rel_ordr = 1`;
- attributes listed in the ***add/delete*** clause are stored in the `Proj_List (attr)` relation; if there is no explicit projection in the ***add/delete*** clause, then the .rd relation is used to find all the attributes of the relation in the ***add/delete*** clause;
- initialize a scalar variable `var_clause` to "add " or to "delete " depending on the clause used;
- initialize a scalar variable `Add_Delete_rel` to the name of the relation in the ***add/delete*** clause.

Illustration.

```
Proj_List (attr )   Join_rels (In_rel,       Rel_ordr)
          TempId              Temporary      1
          Name
          DOB
          Emp#          var_clause = "add "
          Hours        Add_Delete_rel = "Second_job"
```

■

≪ 1. Find which classes among the superclasses of the relation being updated define the
≪ attributes of the relation in the ***add/delete*** clause. Use the procedure `Minimum_Join`
≪ on page 135.

Minimum_Join ();

Illustration.

Classes_in_Branch (.Superclass, .SuperId, In_rel, RelId, Rel_ordr, ordr)

Temporary	DC	Temporary	DC	1	1
Employee	EmpId	Temporary	TempId	1	2
Person	PersId	Temporary	TempId	1	3

Classes_with_attr

(.Superclass, .SuperId, In_rel, RelId, Rel_ordr, ordr, attr)

Temporary	DC	Temporary	DC	1	1	TempId
Temporary	DC	Temporary	DC	1	1	Hours
Employee	EmpId	Temporary	TempId	1	2	Emp#
Person	PersId	Temporary	TempId	1	3	Name
Person	PersId	Temporary	TempId	1	3	DOB

■

≪ 2. For each class in the `Classes_with_attr` table, generate the projection list of
≪ attributes. The attributes in this projection list will be projected from the relation in
≪ the ***add/delete*** clause and this will ensure that for each update statement the
≪ attributes of the ***add/delete*** clause are the same as those of the relation being
≪ updated. If the name of the ID attribute of some superclasses is not the same as in
≪ the relation in the ***add/delete*** clause, then, to project the ID attribute under the
≪ appropriate name, it is necessary to rename it to its corresponding name in the
≪ relation of the ***add/delete*** clause.

```
let elem_of_attr_list be
  if attr = (equiv min of attr by .Superclass) then
      if .SuperId = DC then " [" cat attr
      else " [" cat .SuperId cat "," cat attr;
  else "," cat attr,
let part_of_proj_list be
  if attr = (equiv max of attr by .Superclass) then elem_of_attr_list cat "] "
  else  elem_of_attr_list;
let list_of_attr be equiv max of
            (par cat of part_of_proj_list order attr by .Superclass) by .Superclass;
```

```
let Id_rename be if .SuperId = DC then ""
                 else  "let " cat .SuperId cat " be " cat RelId cat ";";
```

Illustration.

```
Classes_with_attr
(.Superclass, .SuperId,In_rel,      RelId,  Rel_ordr,ordr,attr      )  part_of_proj_list
  Temporary   DC      Temporary     DC      1        1    TempId     ",TempId] "
  Temporary   DC      Temporary     DC      1        1    Hours      " [Hours"
  Employee    EmpId   Temporary     TempId  1        2    Emp#       " [EmpId,Emp#] "
  Person      PersId  Temporary     TempId  1        3    Name       ",Name] "
  Person      PersId  Temporary     TempId  1        3    DOB        "[PersId,DOB,"
```

```
.. continued from the above table
.Superclass    list_of_attr                Id_rename
Temporary      " [Hours,TempId] "          ""
Employee       " [EmpId,Emp#] "            "let EmpId be TempId;"
Person         " [PersId,DOB,Name] "       "let PersId be TempId;"
```

■

◄ 3. Generate an update statement for each relation in the `classes_with_attr`.

```
let update_expr be "update " cat .Superclass cat " " cat var_clause cat " (" cat list_of_attr
                    cat " in " cat Add_Delete_rel cat ");";
let executable_expr be (stmt (fun cat of (Id_rename cat update_expr) order ordr));
S ← [executable_expr] in Classes_with_attr;
S;
```

Illustration.

The following statements will be executed as a result of execution of S:

```
update Temporary add ([TempId,Hours] in Second_job);
let EmpId be TempId;
update Employee add ([EmpId,Emp#] in Second_job);
let PersId be TempId;
update Person add ([PersId,Emp#] in Second_job);
```

■

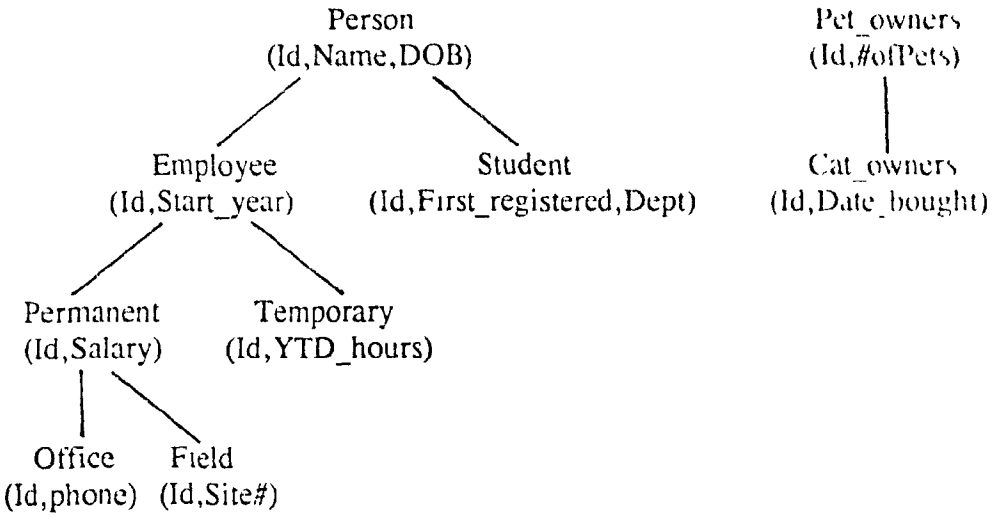
Appendix B

Method Inheritance

This detailed description corresponds to the Section 4.4.

Method inheritance will be illustrated by the following example.

Example B-1. Consider two hierarchies: Person and Pet_owners.



After the declaration of several methods below, the user issues a request to evaluate a relational expression which involves method invocations.

```
function Years_spent (curr_year) on Person;
        return (curr_year − DOB);

function Years_spent (curr_year) on Employee;
        return (curr_year − Start_year);
```

```
function Years_spent (curr_year) on Student;
        return (curr_year − First_registered);

function Years_spent (curr_year) on Cat_owners;
        return (curr_year − Date_bought);

function Which_Floor () on Office;
        Floor ← if phone > 2000 then 2
                else                 1;
        return (Floor);

function Find_status (Income);
        status ← if Income > 25 then "rich"
                else                 "poor";
        return (status);

let curr_year be 91;
let experience be Years_spent(curr_year);
let floor# be Which_Floor();
let status_in_society be Find_status (Salary);
Complete_info ← [Id,experience,floor#,status_in_society] in Office ijoin Cat_owners;
```

During the parsing of the input expression, independently evaluated subexpressions are identified. For each subexpression found, create the relations `Join_rels` and `Method_calls`.

Join_rels	
(In_Rel,	Rel_ordr)
Office	1
Cat_owners	2

Method_calls
(call)
Years_spent
Which_Floor
Find_status

For each method which is declared on a specific class there is a tuple in the

`.Methods` system relation which was introduced in Section 4.3. The information about classes and their superclasses is stored in the `.Hierarchy` system meta-relation discussed in Section 3.2.

.Methods (.rel,	.meth)
Person	Years_spent
Employee	Years_spent
Student	Years_spent
Office	Which_Floor
Cat_owners	Years_spent

.Hierarchy (.Subclass,	.SubId,	.Superclass,	.SuperId)
Employee	Id	Person	Id
Student	Id	Person	Id
Permanent	Id	Employee	Id
Temporary	Id	Employee	Id
Office	Id	Permanent	Id
Field	Id	Permanent	Id
Cat_owners	Id	Pet_owners	Id

■

≪ 1. Find all the classes on which the methods in the `Method_calls` table are defined.

Invoked_methods ← [.rel,.meth] **in** .Methods [.meth **ijoin** call] Method_calls;

Illustration.

Invoked_methods (.rel,	.meth)
Person	Years_spent
Employee	Years_spent
Student	Years_spent
Office	Which_Floor
Cat_owners	Years_spent

■

≪ 2. Find all the ancestors of the relations in the `Join_rels` table and keep the order in
≪ the hierarchy. We use the procedure `Find_Classes_in_Branch`, defined in Appendix
≪ A on page 133 which uses the `Join_rels` table and generates the `Classes_in_Branch`
≪ relation.

Find_Classes_in_Branch ();

Illustration.

Classes_in_branch

(.Superclass,	.SuperId,	In_rel,	RelId,	Rel_ordr,	ordr)
Office	DC	Office	DC	1	1
Permanent	Id	Office	Id	1	2
Employee	Id	Office	Id	1	3
Person	Id	Office	Id	1	4
Cat_owners	Id	Cat_owners	Id	2	1
Pet_owners	Id	Cat_owners	Id	2	2

■

≪ 3. Determine if all the methods in the `Invoked_methods` are defined on the relations in
≪ the `Join_rel` table or their superclasses. If a method is defined on both a subclass
≪ and its superclass, choose the subclass. If the relational expression contains more
≪ than one relation (`Join_rel` table has more than one tuple), verify that each method
≪ is defined on only one of those relations to avoid any ambiguity.

```
let min_index be equiv min of ordr by In_rel,.meth;
let count be equiv max of (par + of 1 order In_rel by .meth) by .meth;
Association <- [.Superclass,.meth,count] where ordr = min_index in
                                Classes_in_branch [.Superclass ijoin .rel] Invoked_methods;
```

Illustration.

Classes_in_branch [.Superclass **ijoin** .rel] Invoked_methods

(.Superclass,	In_rel,	ordr,	.meth	)	min_index	count
Office	Office	1	Which_Floor		1	1
Employee	Office	3	Years_spent		3	2
Person	Office	4	Years_spent		3	2
Cat_owners	Cat_owners	1	Years_spent		1	2

Association

(.Superclass,	.meth,	count)
Office	Which_Floor	1
Employee	Years_spent	2
Cat_owners	Years_spent	2

■

≪ 4. The `Association` table provides two types of information: presence of ambiguously
≪ referenced methods, and an indication to the system of which definition of each
≪ method should be used during the evaluation of the current relational expression. If
≪ the count attribute for any method is greater than one, then that method is defined on
≪ several unrelated classes. If some methods from the `Method_calls` list are not found
≪ in the `Association` table, then these methods are potentially generic.

Illustration.

The `Association` table above indicates that the method `Years_spent` is defined on two unrelated classes and thus introduces ambiguity. The evaluation cannot proceed, and an error message is issued. Also the method `Find_status` is not listed in this table. This tells the system that `Find_status` is either a generic method or it is not defined. ■

The ambiguity can be avoided by rewriting the expression:

Complete_info ← [Id,experience,floor#,status_in_society] **in**
Office **ijoin** ([Id] **in** Cat_owners);

As a result the system considers two expressions for determining which method to apply:

a) [Id,experience,floor#,status_in_society] **in** Office

b) [Id] **in** Cat_owners

Illustration.

The interpretation of the expression a) follows the same steps which produce the intermediate relations listed below.

Classes_in_branch
(.Superclass,.SuperId,In_rel,RelId,Rel_ordr,ordr)

Office	DC	Office	DC	1	1
Permanent	Id	Office	Id	1	2
Employee	Id	Office	Id	1	3
Person	Id	Office	Id	1	4

Classes_in_branch [.Superclass **ijoin** .rel] Invoked_methods

(.Superclass,	In_rel,	ordr,	.meth)	min_index	count
Office	Office	1	Which_Floor	1	1
Employee	Office	3	Years_spent	3	1
Person	Office	4	Years_spent	3	1

Association

(.Superclass,	.meth,	count)
Office	Which_Floor	1
Employee	Years_spent	1

Here the `Association` table unambiguously determines that the system should invoke method `Which_Floor`, defined on the `Office` class, and should invoke method `Years_spent` defined on the `Employee` class.

The interpretation of expression b) does not follow these steps because there are no methods invoked in this expression.

■

Appendix C

Method Broadcasting

This detailed description corresponds to Section 5.1.

To illustrate the operations presented in this appendix, the following example will be used.

Given the OBJECT hierarchy in the Example 5-4 and the following methods,

function Calc_heating_cost () **on** Room;

function Calc_size () **on** Room;

function Calc_size () **on** Couch;

function Calc_size () **on** Table;

function Calc_size () **on** Chair;

function Calc_Depreciation () **on** Furniture;

and, after the four domain algebra statements below, the system encounters an expression which cannot be directly evaluated:

let Heating **be** Calc_heating_cost ();

let Depr **be** Calc_depreciation ();

let RoomSize **be** Calc_size ();

let Size **be sub** Calc_size ();

Costs ← [RoId,Width,RoomSize,Heating,subobjId,Depr,Size]

where Length = 25 **in** Room;

The above statement has to be transformed into the following format in order to be executable:

Costs ← [RoId,Width,RoomSize,Heating, subobjId,Depr,Size] **in**
([RoId,Width,RoomSize,Heating] **where** Length = 25 **in** Room)
[RoId **ijoin** aggregId] Subobjects **ijoin** ([subobjId,Depr,Size] **in**
(Chair [ChId,Depr,Size **ujoin** CoId,Depr,Size] Couch
[CoId,Depr,Size **ujoin** TaId,Depr,Size] Table));

The following scalars are initialized as a result of the parsing of the input expression:

`condition` - the selection criterion in the ***where*** clause;

`requested_class` - the name of the relation in the ***in*** clause;

`final_projection` - contains the assignment of the projected attributes to a new relation.

The following relations are initialized:

Collect_methods_list (C_invok_attr)
- attributes resulting from methods invoked by the input expression defined on the `requested_class`;

Subobj_methods_list (S_invok_attr)
- attributes resulting from methods invoked by the input expression either not defined on the `requested_class` or called with the **sub** keyword;

Collect_attributes_list (C_attr,ordr)
- attributes in the projection list of the input expression that are defined on the `requested_class` with the order in which they are projected.

The other relations used here and described in Sections 5.1 and 3.2:

.Collection (.collectClass,.subobjClass)
- pairs of coleletive class names (`.collectClass`) and their corresponding subobject class names (`.subobjClass`);

Subobjects (collectId,subobjId)

- lists the IDs of the collective objects (`collectId`) and the IDs of the subobjects (`subobjId`) that are part of the collective object;

.Hierarchy (.Subclass,.SubId,.Superclass, SuperId)

- lists the name of the superclass (`.Superclass`) for each subclass (`.Subclass`) and the names of the ID fields in each class

Illustration.

requested_class : Room

condition : "Length = 25"

final_projection : "Cost ← [RoId,Width,RoomSize,Heating, subobjId,Depr,Size] "

Collect_methods_list (C_invok_attr)	Subobj_methods_list (S_invok_attr)	Collect_attributes_list (C_attr,ordr)	
Heating	Depr	RoId	1
RoomSize	Size	Width	2

.Collection (.collectClass,.subobjClass)		Subobjects (collectId,subobjId)		.Hierarchy (.Subclass,.SubId,.Superclass,.SuperId)			
Room	Couch	31	1	Furniture	FuId	OBJECT	Id
Room	Table	31	11	Couch	CoId	Furniture	FuId
Room	Chair	31	21	Table	TaId	Furniture	FuId
		31	22	Chair	ChId	Furniture	FuId
		32	2	Room	RoId	OBJECT	Id
		32	3				

■

≪ 1. Treatment of the collective class.

≪ a) find the name of the ID field of the `requested_class` and include it in the

≪ `Collect_attributes` list.

C_Id ← [.SubId] **where** .Subclass = requested_class **in** .Hierarchy;

Collect_attributes_list ← Collect_attributes_list [C_attr,ordr **ujoin** .SubId,1] C_Id,

Illustration.

C_Id (.SubId)	Collect_attributes_list (C_attr,ordr)	
RoId	RoId	1
	Width	2

■

≪ b) generate a projection expression using the information in the `requested_class` table.

```
let C_attr_temp be if ordr = 1 then C_attr
                  else "," cat C_attr;
let Co_attributes be red max of (fun cat C_attr_temp order ordr);
let Co_invok_attrs be red max of (fun cat ("," cat C_invok_attr) order C_invok_attr);
let collect_projection be "[" cat Co_attributes cat Co_invok_attrs cat "] ";
Collect_proj_rel ← [collect_projection] in (([Co_attributes] in Collect_attributes_list)
                                    ujoin ([Co_invok_attrs] in Collect_methods_list));
let C_projection be "(" cat collect_projection cat " where " cat condition cat " in " cat
    requested_class cat ")[" cat C_Id cat " ijoin collectId] Subobjects ijoin ([subobjId,";
```

Illustration.

Collect_attributes_list				Collect_methods_list	
(C_attr, ordr)		C_attr_temp	Co_attributes	(C_invok_attr)	Co_invok_attrs
RoId	1	"RoId"	"RoId,Width"	Heating	",Heating,RoomSize"
Width	2	",Width"	"RoId,Width"	RoomSize	",Heating,RoomSize"

```
Collect_proj_rel (collect_projection                    )
                 "[RoId,Width,Heating,RoomSize]"
```

When the `c_projection` attribute is projected from the `collect_proj_rel`, it will have the value:

"([RoId,Width,Heating,RoomSize] where Length = 25 in Room)[RoId ijoin collectId] Subobjects ijoin ([subobjId,"

■

≪ 2. Treatment of subobject classes.

≪ a) find the classes of subobjects and the names of their ID fields.

```
Subobj_class_id_rel ← [ subobjClass,.SubId] in ([.subobjClass]
                        where .collectClass = requested_class
                        in .Collection) [.subobjClass ijoin .Subclass] .Hierarchy;
```

Illustration.

Subobj_class_id_rel (.subobjClass,.SubId)

Chair	ChId
Couch	CoId
Table	TaId

■

≪ b) project from each subobject class the ID and methods.

```
let sub_invok_attrs be red max of (fun cat ("," cat S_invok_attr) order S_invok_attr);
let first_tuple be red min of .subobjClass;
let last_tuple be red max of  subobjClass;
let before be if  subobjClass = first_tuple then sub_invok_attrs cat "] in ("
              else  SubId cat sub_invok_attrs cat "] ";
let after be if  subobjClass = last_tuple then ")),"
             else  "[" cat .SubId cat sub_invok_attrs cat " ujoin ",
let sub_interm_proj be before cat  subobjClass cat after,
let sub_projection be red max of (fun cat of sub_interm_proj order  subobjClass),
Sub_projection_rel ← [sub_projection] in (Subobj_class_id_rel ujoin
                                        ([sub_invok_attrs] in Subobj_methods_list)),
```

Illustration.

Subobj_methods_list

(S_invok_attr)	sub_invok_attrs
Depr	",Depr,Size"
Size	",Depr,Size"

#	Subobj_class_id_rel (.subobjClass, SubId)		first tuple	last tuple	before	after
1	Chair	ChId	Chair	Table	"Depr,Size] in ("	"[ChId,Depr,Size ujoin "
2	Couch	CoId	Chair	Table	"CoId,Depr,Size]"	"[CoId,Depr,Size ujoin "
3	Table	TaId	Chair	Table	"TaId,Depr,Size]"	")),""

#	sub_interm_proj
1	"Depr,Size] in (Chair [ChId,Depr,Size ujoin "
2	"CoId,Depr,Size] Couch [CoId,Depr,Size ujoin "
3	"TaId,Depr,Size] Table)),""

Sub_projection_rel
(sub_projection)
"Depr,Size] in (Chair [ChId,Depr,Size ujoin CoId,Depr,Size] Couch [CoId,Depr,Size ujoin TaId,Depr,Size] Table));"

■

≪ 3. Generate and evaluate the final join.

let subobjId_declaration **be stmt** ("let subobjId be " **cat** .SubId **cat** ";");

letStmt ← [subobjId_declaration] **where** .subobjClass = first_tuple **in** Subobj_class_id_rel;

letStmt,

let final_join_expr **be stmt** (final_projection **cat** "in " **cat** C_projection **cat** sub_projection);

Final_join_rel ← [final_join_expr] **in**

([C_projection] **in** Collect_proj_rel) **ujoin** Sub_projection_rel;

Final_join_rel;

Illustration.

The execution of letStmt results in the evaluation of the expression

let subobjId **be** chId;

The execution of Final_join_rel results in the following expression being evaluated:

Cost ← [RoId,Width,RoomSize,Heating, subobjId,Depr,Size] **in**

([RoId,Width,Heating,RoomSize] **where** Length = 25 **in** Room)

[RoId **ijoin** collectId] Subobjects **ijoin** ([subobjId,Depr,Size] **in**

(Chair [ChId,Depr,Size **ujoin** CoId,Depr,Size] Couch

[CoId,Depr,Size **ujoin** TaId,Depr,Size] Table));

■

Appendix D

Implementation of Gedit in Object-Oriented Relix

The Relix code of this appendix provides the details of the Sections 6.3.2 and 6.3.4. The declaration of relations in the inheritance hierarchy in Figure 6.3 is shown below.

relation View (Id,Select,f_origin_x,f_origin_y,f_width,f_height),
relation Graphics (Id,Oseqn,x,y)
relation Point (Id),
relation Circle (Id,rad_x,rad_y);
relation Poly_figures (Id);
relation Polygon (Id),
relation Polyline (Id),
relation Figure (Id, Desc),
relation Button (Id);
relation Constraints (Id,type,Hide,a1_Id,a1_sq,a2_Id,a2_sq,a3_Id,a3_sq,a4_Id,a4_sq);

Graphics **isa** View; Button **isa** View, Figure **isa** OBJECT,
Point **isa** Graphics; Circle **isa** Graphics; Poly_figure **isa** Graphics;
Polygon **isa** Poly_figure, Polyline **isa** Poly_figure; Constraints **isa** Button;
Figure **hasa** Point, Figure **hasa** Circle; Figure **hasa** Polygon;
Figure **hasa** Polyline;

The following functions are declared.

```
function Display () on OBJECT;
   return (DrawSelf());

function DrawSelf() on Graphics;
   return (Draw());

function Draw () on Point;
   Postscript routine to draw a Point
   return (the completion status of the postscript routine);

function Draw () on Circle,
   Postscript routine to draw a Circle
   return (the completion status of the postscript routine);

function DrawSelf () on Constraints;
   if Hide = False then
        Postscript routine to draw the icon
   else  erase icon and arms
   return (the completion status of the postscript routine);

function Overlap (orig_x,orig_y,width,height) on Graphics;
   y_cond ← if (f_orig_y >= orig_y and f_orig_y <= orig_y + height)  or
             (f_orig_y + f_height >= orig_y and
              f_orig_y + f_height <= orig_y + height)
                 then True
           else   False;
   res ← if y_cond = True then
             if (f_orig_x >= orig_x and f_orig_x < orig_x + width) or
               (f_orig_x + f_width >= orig_x  and
                f_orig_x + f_width <= orig_x + width)
                  then  True
             else  False;
   return (res);
```

function Draw_frame () **on** Graphics,

if Select = True **then**

Postscript routine to display small square around point (f_orig_x,f_orig_y)

Postscript routine to display small square around point

(f_orig_x + f_width / 2,f_orig_y)

Postscript routine to display small square around point (f_orig_x + f_width,f_orig_y)

Postscript routine to display small square around point

(f_orig_x + f_width,f_orig_y + f_height / 2)

Postscript routine to display small square around point

(f_orig_x + f_width,f_orig_y + f_height)

Postscript routine to display small square around point

(f_orig_x + f_width / 2,f_orig_y + f_height)

Postscript routine to display small square around point (f_orig_x,f_orig_y + f_height)

Postscript routine to display small square around point

(f_orig_x,f_orig_y + f_height / 2)

else

erase the 9 small squares around the perimeter if they are there.

return (the completion status of the postscript routine),

procedure Move (**in**:dist_x,dist_y; **out**:orig_x,orig_y) **on** View;

orig_x ← f_orig_x + dist_x;

orig_y ← f_orig_y + dist_y;

procedure Move_obj (**in**:dist_x,dist_y; **out**: new_x,new_y,orig_x,orig_y) **on** Graphics;

new_x ← x + dist_x;

new_y ← y + dist_y;

Move (**in**:dist_x,dist_y; **out**:orig_x,orig_y);

Generic function

function Draw_line (from_x,from_y,to_x,to_y)

Postscript routine to draw a line from the point (from_x,from_y) to the point (to_x,to_y).

The following sections which provide the Relix code for the operations described in Chapter 6 are numbered in the same way as their corresponding sections in Chapter 6.

D.3.5.1 Creation of Objects

Given: Create_obj (Id, Oseqn, type, x, y, f_orig_x, f_orig_y, f_width, f_height)

```
let Select be False;
update Graphics add ([ID,Oseqn,x,y,Select,f_orig_x,f_orig_y,f_width,f_height]
                     in Create_obj);
update Point    add ([Id] where type = "P" in Create_obj);
let rad_x be x;  let rad_y be y,
update Circle   add ([Id,rad_x,rad_y] where type = "C" and Oseqn = 2 in Create_obj);
update Circle   delete (where Oseqn = 2 in Circle);
update Polygon  add ([Id] where type = "G" in Create_obj);
update Polyline add ([Id] where type = "L" in Create_obj);
```

```
≪ This completes the graphical hierarchy.  Now place all the objects into the collective
≪ class.  Suppose that the current Figure has its ID value stored in a scalar variable
≪ Curr_Figure_Id.  As we discussed in Section 5.1, the information about which
≪ subobjects are associated with a collective object is stored in the relation Subobjects.
```

```
let subobjId be Id;
let collectId be Curr_Figure_Id;
update Subobjects add ([collectId,subobjId] in Create_obj);
```

```
≪ The Display method can be applied to both Point and Circle classes by invoking that
≪ method on the Figure class.
```

```
let result be Display();
display_result ← [Id, result] in Figure;
```

≪ Since `Polygon` and `Polyline` classes do not have a `DrawSelf` method, they will be
≪ displayed with the help of the `Draw_line` generic method. This method takes as
≪ parameters the end points of a line. Poly-figures are similar in that they consist of
≪ lines. The end points of each line of `Polygon` and `Polyline` objects are recorded in one
≪ relation `ready_to_draw` and then the method `Draw_line` is invoked on that relation

≪ To specify the lines of polygons:

let next_x **be par succ of** x **order** Oseqn **by** Id;
let next_y **be par succ of** y **order** Oseqn **by** Id;
ready_to_draw ← [Id,x,y,next_x,next_y] **in** Polygon;

≪ To specify the lines of polylines:

let last_vertex **be equiv max of** Oseqn **by** Id;
ready_to_draw <+ [Id,x,y,next_x,next_y] **where** Oseqn ~= last_vertex **in** Polyline;
let status **be** Draw_line (x,y,next_x,next_y);
display_result ← [Id,status] **in** ready_to_draw;

D.3.5.2 Selection of Several Objects

Given: Sel_objects (r_orig_x,r_orig_y,r_width,r_height)

let overlapping be Overlap (r_orig_x,r_orig_y,r_width,r_height);
update Graphics **change** Select ← overlapping **using ijoin on** Sel_objects;
let res **be** Draw_frame ();
display_result ← [Id,res] **in** Graphics;

D.3.5.3 Moving Selected Objects

Given: Move_objects (dist_x, dist_y)

```
let  Move_obj (in:dist_x,dist_y; out:new_x,new_y,orig_x,orig_y);
update Graphics change x ← new_x, y ← new_y, f_orig_x ← orig_x, f_orig_y ← orig_y
                    using ijoin on Move_objects;
let res be Draw_frame ();
let result be Display ();
display_result ← [Id,result,res] in Graphics;
```

D.3.5.4 Hiding Constraints / Showing Constraints

Given: Constr_menu_sel (menu_sel)

```
let hide_or_not be if menu_sel = "Hide All"   then True
                        else if menu_sel = "Hide Selected"  then Select
                             else if menu_sel = "Show All"  then False;
update Constraints change Hide ← hide_or_not using ijoin on Constr_menu_sel;
let result be Display ();
display_result ← [Id,result] in Constraints;

≪   Draw the connected arms of the constraints that are not hidden.

let from_x be f_orig_x;          let from_y be f_orig_y;
Arms ← [Id, from_x,from_y,x,y] in ([Id,Oseqn,x,y] in Graphics)
                    [Id,Oseqn ijoin a1_Id,a1_sq]
                    ([a1_Id,a1_sq, from_x,from_y] where Hide = False in Constraints);
```

```
let from_x be f_orig_x;          let from_y be f_orig_y + f_height;
Arms <+ [Id, from_x,from_y,x,y] in ([Id,Oseqn,x,y] in Graphics)
                [Id,Oseqn ijoin a2_Id,a2_sq]
                ([a2_Id,a2_sq, from_x,from_y] where Hide = False in Constraints);

let from_x be f_orig_x + f_width;    let from_y be f_orig_y + f_height,
Arms <+ [Id, from_x,from_y,x,y] in ([Id,Oseqn,x,y] in Graphics)
                [Id,Oseqn ijoin a3_Id,a3_sq]
                ([a3_Id,a3_sq, from_x,from_y] where Hide = False in Constraints);

let from_x be f_orig_x + f_width;    let from_y be f_orig_y;
Arms <+ [Id, from_x,from_y,x,y] in ([Id,Oseqn,x,y] in Graphics)
                [Id,Oseqn ijoin a4_Id,a4_sq]
                ([a4_Id,a4_sq, from_x,from_y] where Hide = False in Constraints);

let status be Draw_line (from_x,from_y,x,y);
display_arms ← [Id, status] in Arms;
```

References

[Bane87] Banerjee J., Kim W., Kim H.J., Korth H., *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*, Proceeding of ACM Special Interest Group on Management of Data 1987 Annual Conference, pages 311-322, San Fransisco, May 1987

[Brac90] Bracha G., Cook W , *Mixin-based Inheritance*, Proceedings of the conference on Object-Oriented Programming: Systems, Languages and Applications, pages 303-311, Ottawa, October 1990

[Bret89] Bretl R., Maier D , et al, *The GemStone Data Management System*, in Kim W. and Lochovsky F , editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283-308, ACM Press, New York, Reading, Mass., Addison-Wesley, 1989

[Butt91] Butterworth P., Otis A., Stein J., *The GemStone Object Database Management System*, Communications of the ACM, 34(10):65-77, October 1991

[Catt91] Cattell R.G.G., *What are next-generation database systems?*, Communications of the ACM, 34(10):31-33, October 1991

[CaWe85] Cardelli L., Wegner P., *On Understanding Types, Data Abstraction, and Polymorphism*, ACM Computing Surveys, 17(4):471-523, December 1985

[Clo87] Clouatre A., *Implementation and applications of recursively defined relations*, Ph.D. thesis, School of Computer Science, McGill University, 1987

[Clo89] Cloutier F., *Object-Oriented Prolog: Design and Implementation Issues*, Master's thesis, School of Computer Science, McGill University, 1989

[Codd70] Codd E.F., *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, 3(6), June 1970, pages 377-387

[Codd72] Codd E.F., *A Data Base Sublanguage Founded on the Relational Calculus*, Proceedings of 1971 ACM SIGFIDET Workshop on Data Description, Access and Control

[CoMa84] Copeland G. and Maier D., *Making Smalltalk a Database System*, in Yormark, editor, Proceedings of ACM SIGMOD International Conference on Management of Data, pages 316-325, Boston, June 1984

[Day85] Dayal U., Buchmann A., Goldhirsch D., Heiler S., Manola F., Orenstein J. and Rosenthal A., *PROBE - A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis*, CCA-85-02 Computer Corporation of America, Cambridge, Massachusetts, July 1985

[Deux90] Deux O., *The Story of O_2*, IEEE Transactions on Knowledge and Data Engineering, 2(1):91-108, March 1990

[Deux91] Deux O., *The O_2 system*, Communications of the ACM, 34(10):35-48, October 1991

[Gold83] Goldberg A., Robson D., *Smalltalk-80. The language and its implementation*, Addison-Wesley, Reading, Mass., 1983

[Huxx91] Hu X X , *Implementation of a Constraint-Based Graphics Editor for Relix*, Master's thesis, School of Computer Science, McGill University, Montreal, February 1992

[Jell90] Jellinghaus R., *Eiffel Linda: an Object-Oriented Linda Dialect*, SIGPLAN NOTICES 25(12):70-84, December 1990

[Kent78] Kent W , *Data and Reality*, North-Holland Publishing Co , New York, 1978

[Khos86] Khoshafian S , Copeland G., *Object Identity*, Proceedings of the conference on Object-Oriented Programming, Systems, Languages and Applications, pages 406-416, Portland, Oregon, September 1986

[KimB88] Kim W, Ballou N, Chou H-T , Garza J , Woelk D., *Integrating an Object-Oriented System With a Database System*, Proceedings of the conference on Object-Oriented Programming: Systems, Languages and Applications, pages 142-152, San Diego, California, September 1988

[Kim91] Kim W , *Object-Oriented Database Systems: Strengths and Weaknesses*, Journal of Object-Oriented Programming, 4(4):21-29, July/August 1991

[Lamb91] Lamb C , Landis G., Orenstein J., Weinreb D , *The ObjectStore database system*, Communications of the ACM, 34(10):51-63, October 1991

[Merr77] Merrett T.H., *Relations as Programming Language Elements*, Information Processing Letters, 6(1):29-33, February 1977

[Merr84] Merrett T H., *Relational Information Systems*, Reston 1984

[Meye88a] Meyer B., *Object-oriented Software Construction*, Prentice Hall, New York 1988

[Meye88b] Meyer B., *Eiffel: A Language and Environment for Software Engineering*, The Journal of Systems and Software, 8(3), 199-246, June 1988

[Moon89] Moon D., *The Common Lisp Object Oriented Programming Language Standard*, in Kim W. and Lochovsky F., editors, *Object Oriented Concepts, Databases, and Applications*, pages 49-79, New York, ACM Press, Reading Mass., Addison-Wesley, 1989

[Mull89] Mullin M., *Object-Oriented Program Design with Examples in C++*, Addison Wesley, 1989

[NgRi89] Nguyen G., Rieu D., *Schema Evolution in Object Oriented Database Systems*, Data & Knowledge Engineering, 4(1), North Holland, July 1989

[Nier89] Nierstrasz O., *A Survey of Object Oriented Concepts*, in Kim W. and Lochovsky F., editors, *Object-Oriented Concepts, Databases, and Applications*, pages 3-22, New York, ACM Press, Reading, Mass., Addison Wesley, 1989

[Penn87] Penney D., Stein J., *Class Modifications in the GemStone Object Oriented DBMS*, Proceedings of the conference on Object Oriented Programming Systems, Languages and Applications, pages 111-117, Orlando, Florida, October 1987

[Rowe87] Rowe L., Stonebraker M., *The POSTGRES Data Model*, Proceedings of the 13th International Conference on Very Large Data Bases, pages 83-95, Brighton, England, September 1987

[Shaf91] Shafer D., Ritz D., *Practical Smalltalk*, String-Verlag, 1991

[Shaf86] Shaffert C., Cooper T., Bullis B., Kilian M., Wilpolt C., *An Introduction to Trellis/Owl* Proceedings of the conference on Object-Oriented Programming Systems, Languages and Applications, pages 9-16, Portland, Oregon, September 1986

[She88] O'Shea T., *Panel: The Learnability of Object-Oriented Programming Systems*, Proceedings of the conference on Object-Oriented Programming Systems, Languages and Applications, pages 502-504, San Diego, California, September 1988

[Ston86] Stonebraker M., Rowe L., *The Design of POSTGRES*, Proceedings of International Conference on Management of Data, Washington, D.C., May 1986, published in SIGMOD RECORD, 15(2):340-355, June 1986

[Ston91] Stonebraker M., Kemnitz G., *The Postgres Next Generation Database Management System*, Communications of the ACM, 34(10):79-92, October 1991

[Stro86] Stroustrup B., *The C++ Programming Language*, Addison-Wesley, 1986

[UnSc90] Unland R., Schlageter G., *Object-Oriented Database Systems: Concepts and Perspectives*, in Blaser A., editor, *Database Systems of the 90s*, LNCS 466, pages 154-197, Spring-Verlag, New York, 1990

[Weg87] Wegner P., *Dimensions of Object Based Language Design*, Proceedings of the conference on Object-Oriented Programming: Systems, Languages and Applications, pages 168-182, Orlando, Florida, October 1987

[Yon87] Yonezawa A., Tokoro M., editors, Object-Oriented Concurrent Programming, The MIT Press, 1987