ГОДИШНИК НА СОФИЙСКИЯ УНИВЕРСИТЕТ „СВ. КЛИМЕНТ ОХРИДСКИ“

ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
Книга 1 — Математика
Том 85, 1991

ANNUAIRE DE L'UNIVERSITE DE SOFIA „ST. KLIMENT OHRIDSKI“

FACULTE DE MATHEMATIQUES ET INFORMATIQUE
Livre 1 — Mathématiques
Tome 85, 1991

CLIP++: AN OBJECT-ORIENTED EXTENSION OF A RELATIONAL DBMS

PAVEL AZALOV, VENTSISLAV DIMITROV

Павел Азалов, Венцислав Димитров. CLIP^{++}: ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ РАСШИРЕНИЕ ОДНОЙ РЕЛЯЦИОННОЙ **СУБД**

В настоящей работе представлена система Clip^{++}, которая является объектно-ориентированным расширением реляционной СУБД Clipper 5.0. Clip^{++} дополняет одну из самых распространенных реляционных СУБД объектно-ориентированными средствами. Она обладает основными характеристиками объектно-ориентированных систем: можно определять классы и обекты; обеспечиваются йерархия классов, наследственность и полиморфизм. Реляционные возможности СУБД Clipper 5.0 используются чтобы поддерживать устойчивость объектов.

Pavel Azalov, Ventsislav Dimitrov. CLIP^{++}: AN OBJECT-ORIENTED EXTENSION OF A RELATIONAL **DBMS**

In this paper the Clip^{++} system is presented. It is an object-oriented extension of the Relational DBMS Clipper 5.0. Clip^{++} tries to upgrade one of the most widely spread Relational DBMS with some object-oriented features. It has the basic characteristics of an object-oriented system: classes and objects can be defined; class hierarchy, inheritance and polimorphism are supported. On the other hand, the relational features of the DBMS Clipper 5. 0 are used to achieve the object persistency.

1. INTRODUCTION

The wide usage of the relational DBMS in the development of computer application systems is due mainly to the simplicity of the relational model. But the relational DBMS cannot directly satisfy the requirements of the new applications in the fields like, for example, the office automation, computer-aided design, and knowledge based systems. These untraditional applications set some serious requirements:

— possibilities for direct modelling of the objects of the application area; the relations of the relational model are not always suitable for this;

— the uniform representation of all relations in the relational database suggests a single level hierarchy; the hierarchy is a typical feature of the modeled areas;

— the semantic modelling features and knowledge representation capabilities are insufficient.

Some approaches for solving these problems can be found in different publications [Gar 89], [Ong 84], [Obs 86], [Unl 90]. In one of the approaches the relation model is replaced by other models, like: entity-relationship model, semantic data models or object-oriented data models. In the first one the relational model is expanded introducing complex objects and abstract data types. One object-oriented extension of the relational database system Clipper 5.0 is represented in this paper.

2. PRELIMINARIES

2.1. OBJECT-ORIENTED CONCEPTS

The object is a basic notion in the object-oriented paradigm. The object unites both state and behavior of the modeled entities in one. The state of an object is characterized with the values of its attributes. The set of operations called *methods*, applicable to these attributes, characterizes the objects, behavior. Objects, sharing the same attributes and methods, are grouped into a class and are called *instances* of that class. The class describes the attributes of its instances and the operations (methods) applicable to them. The methods (their realization), as well as the attributes, however, are not visible from outside the object. The objects can communicate with one another only through messages. These messages are requests for an object to change its state, to return a value, or to perform some sequence of actions. The set of messages to which an object responds constitutes the public interface of an object. Each object responds to a received message by executing a method.

The notion of a class is different from that of a type. Its specification is the same as that of a type but it is more a runtime notion.

Inheritance and polymorphism are key concepts of an object-oriented system. The inheritance allows us to specify or to implement only the extensions between the existing classes and the new ones. The new class "inherits" all properties of the old one — its superclass. The polymorphism allows us to send the same message to different objects and have each object respond in a way appropriate to the kind of object.

2.2. THE SYSTEM CLIPPER 5.0

One of the most famous and worldwide spread relational systems is dBASE. On the basic concepts of dBASE a number of other systems appeared and developed successfully:

— interpreters (dBASE, FoxBase);

— compilers (Clipper);

— translators from the dBASE programming language to other programming languages (dbx3:translator to C);

— libraries for dBASE files manipulation (SoftC: C-library).

Additionally, almost all Relational DBMS support dBASE file compatibility and can manipulate dBASE file format.

The Clipper system takes a central place in the development of the dBASE-like systems. The latest release Clipper 5.0 has some important new features to which we should pay more attention:

— a preprocessor allowing command definition and redefinition;

— a new data type: code block — executable piece of data, which differs from the macros by its compile-time translation;

— four predefined classes: **ERROR, GET, TBROWSE, TBCOLUMN**.

In Clipper 5.0 the definition of new classes is not allowed. Therefore, there is no inheritance or polymorphism. The attributes of an object can be directly accessed and their values changed, so there is no encapsulation either.

A short description of an object-oriented extension of Clipper 5.0, called Clip^{++}, follows in this paper.

2.3. OVERVIEW OF THE CLIP^{++}

The Clip^{++} system combines the basic features of the object-oriented and the database managing systems. Some of them are:

— class and object definition;

— sub-class definition with inheritance;

— object encapsulation;

— polymorphism;

— persistence.

The Clip^{++} system is entirely build using only the Clipper 5.0 features.The existing preprocessor is used essentially. The object-oriented features of Clip^{++} are implemented as user-defined commands and a set of functions.

3. CLIP^{++} OBJECT MODEL

The four principal concepts of the Clip^{++} object model are: object, message, class and inheritance.

A new class is defined using a command with the following general format:

```
DEFINE CLASS <class_name> : <parent_class> ;
            [ <attr> {, <attr> } : <type> ]
```

In curly brackets ("{", "}") are enclosed expressions that can appear 0, 1 or more times and in square ones ("[", "]"), expressions that appear at least once.

The allowable types are: `NUMBER (<digits> [,<decimals>])`, `CHARACTER (<length>)`, `DATE`, `MEMO`, `OBJECT (<of_class>)`. Let us note that the type of an object field can be another object of any already defined class. Thus an object hierarchy can be build. The class methods are defined as attributes of type `FUNCTION (<para#>)` or `INITIALIZER (<para#>)`. The last one represents a constructor method. The appending and removal of attributes are performed respectively by the commands:

```
ADD TO CLASS  <class_name>  ATTRIBUTES;
            [  <attr> {,  <attr> }  :  <type> ]
DELETE FROM CLASS <class_name> ATTRIBUTES <attr> {, <attr> }
```

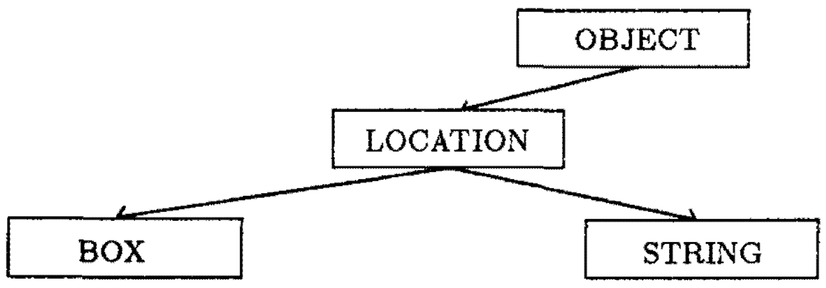
The removal of a class is possible if it has no subclasses and is not nested in other class definitions. The command is:

```
DELETE CLASS <class_name>  {, <class_name> }
```

A subclass is defined using the class definition command. Each class can have many subclasses but only one superclass. There is only one predefined root-class called `OBJECT`. Example:

```
DEFINE CLASS location : object ;
     x, y : NUMERIC (2) ;
     init : initializer (2) ;
     move : function  (2)
DEFINE CLASS box : location ;
     h, w : NUMERIC (2) ;
     typ  : CHARACTER (1) ;
     clear: function (0)
DEFINE CLASS string : location ;
     s    :  character (24)
```

In the above example three classes are defined, building the following hierarchy:



3.2. OBJECT DECLARATION

Each object is an instance of an existing class. Its declaration is similar to the Clipper 5.0 variable declaration. The object declaration command has the following format:

```
| PRIVATE | PUBLIC | DECLARE | LOCAL : <class_name> ;
<obj_name>(<param_list>) {, <obj_name>(<param_list>) }
```

An object can be declared with no round brackets. The `<param_list>` may be empty. Here are some examples:

```
:location L1, L2 (4, 4), L3 ()
PUBLIC :box B1(4, 4, 10, 10, '-')
```

When no initializing parameter list is passed during an object declaration, no constructor (initializer) is invoked for that object (ex. L1). If a parameter list is passed (even if it is empty), Clip^{++} tries to initialize the object calling a constructor-method of that class. If there are several constructors defined in one and the same class, Clip^{++} selects that one which expects number of parameters, closest to the number of the parameters passed, but not less if possible. For example, if the BOX class definition has three constructors INIT1, INIT3, INIT6, expecting 1, 3 and 6 parameters respectively, INIT3 will be selected. The constructor is a method designed to initialize the object at declaration time. If a class contains nested classes in its definition, the best place for initializing the nested objects is in the constructor-method.

3.3. METHOD DEFINITION

The method is a regular Clipper 5.0 function. When a method is defined, its name is preceded by the class name separated with colon (":"). For example:

```
FUNCTION LOCATION:MOVE( X, Y )
..... <method body> .....
RETURN <value>
```

To compile a method and make the late binding possible, Clip^{++} contains a component called *pre-preprocessor*. It creates preprocessor directives converting the methods into regular Clipper 5.0 functions. The pre-preprocessor gives the methods new names, described in the _C_TABLE_ structure (explained later) as internal method identifiers. During the late binding from the object class and the sent message, the internal identifier is generated and the real compiled function with that name is called. For example, the method name LOCATION:MOVE is translated to an identifier, which looks like F00100202. When a message MOVE (3, 4) is sent to an object of class LOCATION, the F00100202 identifier is generated and the F00100202 function is called.

The attributes of an object are accessible only through its methods. When describing a method body, the class attributes can be accessed. In order to distinguish the attributes from the other variables and objects, their names are preceded by a dot ("."). All the methods of the same class are called in the same way. For example:

```
FUNCTION LOCATION:INIT( X, Y )
.X := X // .X and .Y are object attributes of class LOCATION;
.Y := Y // X and Y are the parameters passed
RETURN Nil
FUNCTION LOCATION:MOVE( X, Y )
.Init(X, Y) // .Init is a method of the same LOCATION class;
RETURN Nil // if used Init(X,Y) function INIT will be called
```

If an attribute is a nested object, a message can be sent using a dot in front of its name (as when accessing attributes) and a colon between its name and the message. The universal way is to assign the object-attribute to a local variable of the same class, work with it as with any object, and then assign it back to the object-attribute. In this way nested objects with unlimited depth can be handled. For example:

```
FUNCTION CLASS1:METHOD1
PRIVATE :location L1(1, 2)
.LOC := :L1        // LOC is a nested object-attribute of class
.LOC:Move(4, 4)    // LOCATION to which message Move is sent.
:L1 := .LOC        // This shows the processing of a nested
:L1:Move(6, 6)     // object-attribute using a temporary
.LOC := :L1        // private object :L1
RETURN Nil
```

Looking back to the object declaration, let us note that the constructor calling is nothing more than an automatic sending of one of the messages marked as initializing.

4. OBJECT MANAGEMENT IN CLIP^{++}

4.1. CLASSES AND ATTRIBUTES DESCRIPTION

One of the basic requirements to a database system is the persistence. For this reason we must have tools to save objects in secondary storage. It is even more necessary the class description to be persistent. Therefore, Clip^{++} maintains two database files: @CLASSES.DBF and @ATTRIBS.DBF. The @CLASSES.DBF contains the class definitions and the relations of classes, while in the @ATTRIBS.DBF the attributes and methods are described. For faster access to these descriptions the file @CLASSES.DBF is indexed on its attributes CLASS_NAME, CLASS_ID, and PARENT_ID. The @ATTRIBS.DBF file is indexed on CLASS_ID to link the attributes and methods to their class.

The relational schemas of these files are the following:

	Attr. Name	Type	Length	Decimals
@CLASSES.DBF:	CLASS_ID	Character	3	
	CLASS_NAME	Character	11	
	PARENT_ID	Character	3	
	CLASS_ATTR	Numeric	3	0
@ATTRIBS.DBF:	CLASS_ID	Character	3	
	ATTR_NAME	Character	15	
	ATTR_TYPE	Character	1	
	ATTR_LEN	Numeric	3	0
	ATTR_DEC	Character	3	
	ATTR_MODE	Character	1	

Their contents are, for example, the following:

@CLASSES.DBF:

CLASS_ID	CLASS_NAME	PARENT_ID	CLASS_ATTR
000	OBJECT		0
001	LOCATION	000	4
002	BOX	001	6
003	STRING	001	6
...	...	...	.

@ATTRIBS.DBF:

CLASS_ID	ATTR_NAME	ATTR_TYPE	ATTR_LEN	ATTR_DEC
001	X	N	2	0
001	Y	N	2	0
001	INIT	I	2	001
001	MOVE	F	2	002
002	H	N	2	0
002	W	N	2	0
002	TYP	C	9	0
002	INIT	I	5	001
002	MOVE	F	2	002
002	CLEAR	F	0	003
...	...	...	...	...

4.2. OBJECTS IN MEMORY

For a proper object manipulation Clip^{++} supports a memory structure (_C_TABLE_) describing the classes used. In the terms of Clipper 5.0 it is a 2-dimensional array, but some of its elements are also 2-dimensional arrays. It is possible to read all class definitions into memory. If this is not done, when an attempt is made to use a class, its description is added to the structure. This slows down the system when a class is accessed for the first time, but as only the necessary class descriptions are loaded into _C_TABLE_ , the look-up in it is faster and less memory is used.

The _C_TABLE_ structure describing the classes in memory is the following:

```
           ClassNAME  ClassID   ClassMETHOD  ClassDATA    ClassPID
          ┌──────────┬─────────┬────────────┬────────────┬──────────┐
Type:     │<char-11> │<char-3> │<array[3,x]>│<array[3,x]>│<char-3>  │
          ├──────────┼─────────┼────────────┼────────────┼──────────┤
Exam:     │ LOCATION │     001 │     *      │     *      │   000    │
          ├──────────┼─────────┼─────/──────┼──────\─────┼──────────┤
           ...............          /               \
                                   /                 \
           Mtd.NAME  Mtd.ID     Para#     DataNAME  DataTYPE Orig.CID
          ┌─────────┬──────────┬───────┐ ┌─────────┬────────┬────────┐
Type:     │<char-15>│<char-9>  │<num-2>│ │<char-15>│<char-1>│<char-3>│
          ├─────────┼──────────┼───────┤ ├─────────┼────────┼────────┤
Exam:     │ Move    │F00100202 │   2   │ │ X       │   N    │   001  │
          ├─────────┼──────────┼───────┤ ├─────────┼────────┼────────┤
           .............                 .............
```

ClassPID — the CLASS Parent IDentifier;

Orig.CID — the Class IDentifier, where the attribute is originally defined (it can be inherited);

Mtd.ID — internal (memory only) Method IDentifier; consists of: method type ('F'unction or 'I'nitializer), class identifier (3 characters), method identifier (3 characters) and number of expected parameters (2 characters);

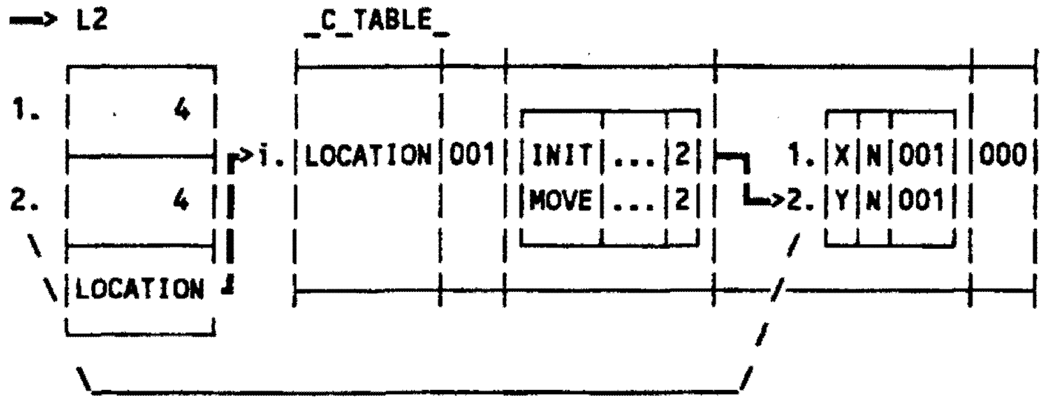
DataTYPE — Attribute TYPE: C, N, D, L, M, O: nnn, where nnn is the identifier of the class of which the nested object is.

When a new class description must be loaded into _C_TABLE_ , the following actions take place. First, the parent class (if any) is determined and if necessary loaded recursively into memory. Second, its description is appended (doubled) as last element of the structure. This ensures the inheritance of all its attributes and methods. At the end, the descriptions of the elements, defined in the class itself, are read and if they overlap existing elements, the new definitions replace the old ones (ensuring the polymorphism), else they are added to the sub-arrays describing the methods or the attributes.

4.3. MEMORY REPRESENTATION OF OBJECT VALUES

The objects themselves are represented in memory as arrays with one element more than the number of their attributes. In Clipper 5.0 each array element can be of different type, even another array. Using this feature, the value of each attribute of an object is kept in a certain element of the array representing that object. The array takes the name of the object and is directly accessed using it. The attribute name descriptions and their types are kept in the _C_TABLE_, from where the access to the corresponding attribute values takes place. The last element of the array is the object class name.

The access to a specific attribute of an object is done as follows. First, using the class name (the last array element) the class description in `_C_TABLE_` is located. Then in the attribute description sub-array the specific attribute name is found. Its index is the index of the element in the array, where the attribute value is stored.



To ensure the encapsulation of the objects, their values can be accessed only through the methods of the corresponding class.

The access of the methods is absolutely identical: from the class name (last array element) and the method name the appropriate function is called. This automatically implements the late binding and the polymorphism.

4.4. MESSAGE SENDING

To distinguish the objects from the memory variables, their names are preceded by a colon (":"). For example:

```
:L2:Move(10,10)   // a message Move(10,10) to the object L2
:L1 := L2         // assigns object L2 to L1 (":") in front of
                  // the second object are not compulsory.
```

The construction `:<identifier>` is recognizes by the preprocessor and the above example is translated to the functions:

```
_Obj_MExec_( L2, "Move", 10, 10 )
_Obj_Assign_( L1, L2 )
```

The format of sending messages to objects is the following:

```
:<object_name>:<method_name>{( <param_list> )}
:<object_name1> {:}= {:}<object_name2>
```

Here the curly brackets designate that the expression within them can appear not more than once. The `<param_list>` may be empty.

The message sending to a nested attribute-object is similar but the object name is quoted as an attribute (details in 3.3.):

```
.<obj_attr>:<method_name>{( <parameter_list> )}
```

When passing an object as a function argument the colon is not compulsory:

```
......
Funct1( L1 )        // equivalent to funct1( :L1 )
:02:Method2( L1 )   // equivalent to :02:Method2( :L1 )
......
```

When defining a new class Clip^{++} creates a database file named with the first 8 characters of the class name and with the following attributes: one attribute for each class attribute with the same names and one additional attribute called `_OID_` (object identifier). The object identifier is generated by Clip^{++} upon saving a new object in secondary storage and has unique value for the whole system. It is in the range from 1 to $36^3 - 1 = 46655$ (ZZZ as a 36-decimal number). Therefore, up to 46 655 objects can be saved in secondary storage at the same time. The increase of that number is only a technical problem.

The objects of each class are saved in as many files as the number of the superclasses of the class plus one for the class itself is. This may seems rather clumsy but gives some benefits, for example: easy overlapping of attributes, easy transformation of an object from one class to another, much easier and faster look-up, objects of all subclasses of a class can be treated as pertaining to the parent class. The last one means that if the X-coordinate of all `LOCATION` objects is increased by one, the same will happen to all objects of the subclasses `BOX` and `STRING`. Looking back to the `LOCATION` and `BOX` classes, the following relational schemas will be created:

```
LOCATION ( _OID_, X, Y )
BOX      ( _OID_,       H, W, TYP )
```

Note: The root class has at least `_OID_` attribute.

Each object in secondary storage is decomposed into its attributes which are saved in the file corresponding to the class where they are originally defined. For example, the object `B` of class `BOX`: `B(X=4, Y=5, H=10, W=20, TYP ='-')`, goes into 3 files:

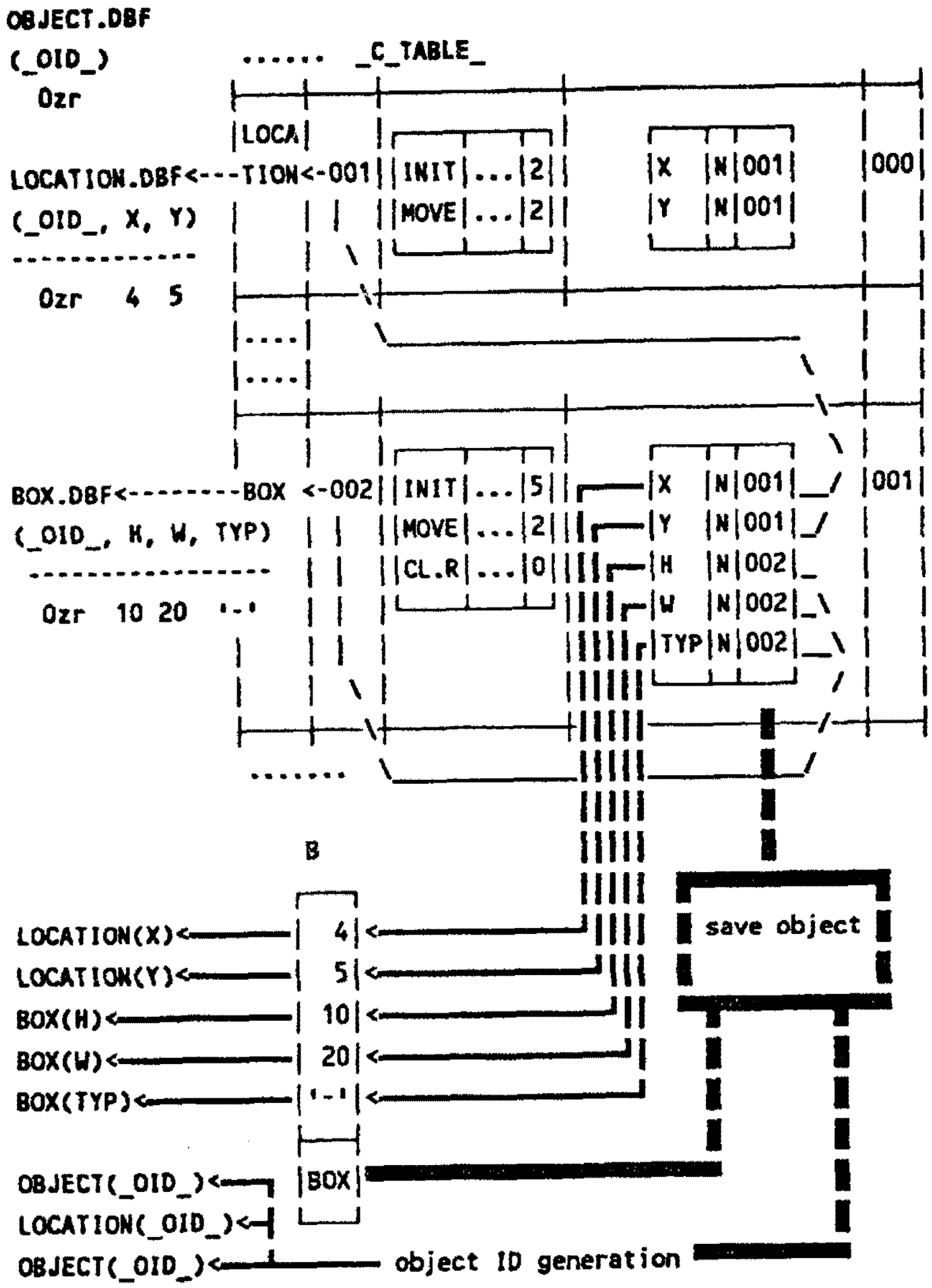
```
OBJECT (_OID_)   LOCATION (_OID_, X, Y)   BOX (_OID_, H, W, TYP)
--------------   ----------------------   ----------------------

      ...              ..........          ..................
      Ozr              Ozr    4  5               Ozr  10 20   '-'
```

When an object is read from secondary storage into memory, first a selection of `_OID_` (it is unique) is performed and then a join on `_OID_` of all relations is build of. The resulting tuple is put into the object structure (array) together with the class name as last element.

```
           _OID_  X Y  H   W   TYP
           -----------------------
OBJECT     Ozr
LOCATION   Ozr    4 5
BOX        Ozr         10  20  '-'
```

An object is saved in secondary storage in much the same way: it is decomposed into tuples (of attributes defined in the same class together with an object identifier) and saved in the corresponding file. Let `B` is the above object of class `BOX`. It is saved in the following way: each attribute is checked, where it is originally defined, and using the class name the database file is determined (see the following scheme).



4.6. LOCATING OBJECTS

The location of objects in secondary storage by the values of their attributes uses one specific feature of Clipper 5.0: each element of an array can be of any type including code block. So to solve the contradiction between the object encapsulation and the direct access of the data in the relations, the notion functional object is introduced in Clip^{++}. A functional object has as many attributes as the corresponding object and it is declared in the same way, but instead of values contains code blocks that are Boolean expressions on the corresponding attributes. Thus the requirement for object encapsulation is fulfilled and at the same time an object can be located by the values of its attributes. The functional object is used in the object locating commands. For each object in the secondary storage the Boolean expressions are calculated and if they are true, the desired object is found.

In the subsequent command formats the curly brackets designate that the expression can appear not more than once.

The object locating command is :

```
LOCATE OBJECT FOR <fobjfor> WHILE <fobjwhile>
       {NEXT <next>} {<rest:REST>}
```

Example:

```
Private :LOCATION L4( {|x| x>5 .and. x<10}, {|w| w # z} )
LOCATE OBJECT FOR :L4
```

The argument of the code block is always the corresponding attribute, no matter what its name is. In the above example the `'x'` argument represents the `'X'` attribute and the `'w'` argument represents the `'Y'` attribute, because the Boolean expression occupies the `'Y'` attribute place. The `'z'` is a memory variable.

Objects are added in the database with the command:

```
APPEND {OBJECT} :<obj>
```

Objects replace other objects with the commands:

```
REPLACE {CURRENT} {OBJECT} WITH :<obj>
REPLACE OBJECT <oid> WITH :<obj>
```

In the first format the object replaces the "current" object. (Clip^{++} supports "current" object and "default" class. They are usually the last accessed object and class.) In the second format an object with a definite identifier is replaced.

The object is read into memory using a function:

```
<object_name> := ReadObject( { <object_identifier> } )
```

A "blank" object is added with the commands:

```
APPEND {BLANK} OBJECT                    // of current class
APPEND {BLANK} OBJECT OF CLASS <class>
```

Skipping $< n >$ objects from the default class is done by the command:

```
SKIP OBJECT  {<n>}
```

The current object can be deleted using:

```
DELETE OBJECT
```

`LOCATE` in combination with other commands can perform group operations. For example:

```
Do While ! Eof ()
   LOCATE OBJECT FOR :L4 REST
   DELETE OBJECT
EndDo
```

deletes all the objects matching the functional object `:L4`.

The above database commands can be made object-oriented. One approach is to add some predefined methods to the root class OBJECT. Let us note that the object itself can be accessed from within its methods by the name "`_a_`" (in most 00 languages it is "self" or "this"). The only thing these methods would do is to execute the corresponding database command. For example:

```
:<obj>:Append()          <=> APPEND [OBJECT] :<obj>
:<obj>:Replace(<oid>)    <=> REPLACE OBJECT <oid > WITH :<obj>
:<obj>:Read({<oid>})     <=> <obj> := ReadObject( {<oid>} )
```

Another approach for object-orienting of the commands is to introduce a special class (**dbaseclass**) with an instance (**dbaseobj**). The methods of that class can implement the database commands by just executing them. For example:

```
:dbaseobj:Append(<obj>)      <=> APPEND [OBJECT] :<obj>
:dbaseobj:Skip( {<n>} )      <=> SKIP OBJECT {<n>}
:dbaseobj:Delete()           <=> DELETE OBJECT
```

This is a more universal approach, but more complicated. Both of them can be easily implemented using the Clip^{++} features.

REFERENCES

[Gar 89] Gardarin, G., et al. Managing Complex Objects in an Extensible Relational DBMS. — Proc. of the 15th International Conf. on VLDB, 1989, 55–65.

[Ong 84] Ong, J., et al. Implementation of Data Abstraction in the Relational Database System Ingres. SIGMOD, rec. 14, 1984, 1–14.

[Osb 86] Osborn, S., T. Heaven. The Design of Relational Database System with Abstract Data Types for Domains. — ACM Transactions of Database Systems, **11**, No 3, Sept. 86, 357–373.

[Unl 90] Unland R., G. Schlageter. Object-Oriented Database Systems: Concepts and Perspectives. LNCS 466, Database Systems of the 90's, A. Blaser (Ed.), 154–197.

Received 12.06.1992