TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>1</td>
</tr>
<tr>
<td>Organising Committee</td>
<td>2</td>
</tr>
<tr>
<td>Referees</td>
<td>3</td>
</tr>
<tr>
<td>Program</td>
<td>5</td>
</tr>
<tr>
<td>Papers (In order of presentation)</td>
<td>9</td>
</tr>
<tr>
<td>&quot;A value can belong to many types&quot;</td>
<td></td>
</tr>
<tr>
<td>B H Venter, University of Fort Hare</td>
<td>10</td>
</tr>
<tr>
<td>&quot;A Transputer Based Embedded Controller Development System&quot;</td>
<td></td>
</tr>
<tr>
<td>M R Webster, R G Harley, D C Levy &amp; D R Woodward, University of Natal</td>
<td>16</td>
</tr>
<tr>
<td>&quot;Improving a Control and Sequencing Language&quot;</td>
<td></td>
</tr>
<tr>
<td>G Smit &amp; C Fair, University of Cape Town</td>
<td>25</td>
</tr>
<tr>
<td>&quot;Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers&quot;</td>
<td></td>
</tr>
<tr>
<td>P Machanick, University of Witwatersrand</td>
<td>40</td>
</tr>
<tr>
<td>&quot;Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface&quot;</td>
<td></td>
</tr>
<tr>
<td>L van Zijl &amp; D Mitton, University of Stellenbosch</td>
<td>51</td>
</tr>
<tr>
<td>&quot;Product Form Solutions for Multiserver Centres with Hierarchical Classes of Customers&quot;</td>
<td></td>
</tr>
<tr>
<td>A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig</td>
<td>69</td>
</tr>
<tr>
<td>&quot;A Reusable Kernel for the Development of Control Software&quot;</td>
<td></td>
</tr>
<tr>
<td>W Fouché and P de Villiers, University of Stellenbosch</td>
<td>83</td>
</tr>
<tr>
<td>&quot;An Implementation of Linda Tuple Space under the Helios Operating System&quot;</td>
<td></td>
</tr>
<tr>
<td>P G Clayton, E P Wentworth, G C Wells and F de Heer-Menlah, Rhodes University</td>
<td>95</td>
</tr>
<tr>
<td>&quot;The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System&quot;</td>
<td></td>
</tr>
<tr>
<td>K Macgregor, University of Cape Town &amp; R Campbell University of Illinois at Urbana-Champaign</td>
<td>107</td>
</tr>
</tbody>
</table>
"Concurrency Control Mechanisms for Multidatabase Systems"
A Deacon, University of Stellenbosch 118

"Extending Local Recovery Techniques for Distributed Databases"
H L Victor & M H Rennhackkamp, University of Stellenbosch 135

"Analysing Routing Strategies in Sporadic Networks"
S Melville, University of Natal 148

"The Design of a Speech Synthesis System for Afrikaans"
M J Wagener, University of Port Elizabeth 167

"Expert Systems for Management Control: A Multiexpert Architecture"
V Ram, University of Natal 177

"Integrating Similarity-Based and Explanation-Based Learning"
G D Oosthuizen and C Avenant, University of Pretoria 187

"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town 201

"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch 211

"Building a secure database using self-protecting objects"
M Olivier and S H von Solms, Rand Afrikaans University 228

"Modelling the Algebra of Weakest Preconditions"
C Brink and I Rewitsky, University of Cape Town 242

"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch. 262

"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal 276
FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLA conference.

* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.

* I believe that it is first time that a Symposium has been held outside of the Transvaal.

* Over 85 people will be attending. Nearly all will have attended both events.

* A Sponsorship package for both SACLA and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLA and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLA, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman
SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.
P S Kritzinger, University of Cape Town.
M H Linck, University of Cape Town.

SPONSORS

ISM
GENMIN
FRD
### LIST OF REFEREES FOR 6th RESEARCH SYMPOSIUM

<table>
<thead>
<tr>
<th>NAME</th>
<th>INSTITUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnard, E</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Becker, Ronnie</td>
<td>UCT</td>
</tr>
<tr>
<td>Berman S</td>
<td>UCT</td>
</tr>
<tr>
<td>Bishop, Judy</td>
<td>Wits</td>
</tr>
<tr>
<td>Berman, Sonia</td>
<td>UCT</td>
</tr>
<tr>
<td>Brink, Chris</td>
<td>UCT</td>
</tr>
<tr>
<td>Bodde, Ryn</td>
<td>Networks Systems</td>
</tr>
<tr>
<td>Bornman, Chris</td>
<td>UNISA</td>
</tr>
<tr>
<td>Bruwer, Piet</td>
<td>UOFS</td>
</tr>
<tr>
<td>Cherenack, Paul</td>
<td>UCT</td>
</tr>
<tr>
<td>Cook Donald</td>
<td>UCT</td>
</tr>
<tr>
<td>de Jaeger, Gerhard</td>
<td>UCT</td>
</tr>
<tr>
<td>de Villiers, Pieter</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Ehlers, Elize</td>
<td>RAU</td>
</tr>
<tr>
<td>Eloff, Jan</td>
<td>RAU</td>
</tr>
<tr>
<td>Finnie, Gavin</td>
<td>Natal</td>
</tr>
<tr>
<td>Gaynor, N</td>
<td>AECI</td>
</tr>
<tr>
<td>Hutchinson, Andrew</td>
<td>UCT</td>
</tr>
<tr>
<td>Jourdan, D</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kourie Derrick</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kritzinger, Pieter</td>
<td>UCT</td>
</tr>
<tr>
<td>Krzesinski, Tony</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Laing, Doug</td>
<td>ISM</td>
</tr>
<tr>
<td>Labuschagne, Willem</td>
<td>UNISA</td>
</tr>
<tr>
<td>Levy, Dave</td>
<td>Natal</td>
</tr>
</tbody>
</table>


MacGregor, Ken
Machanick, Philip
Mattison Keith
Messerschmidt, Hans
Mutch, Laurie
Neishlos, N
Oosthuizen, Deon
Peters Joseph
Ram, V
Postma, Stef
Rennhackkamp, Martin
Shochot, John
Silverberg, Roger
Smit, Riel
Smith, Dereck
Terry, Pat
van den Heever, Roelf
van Zijl, Lynette
Venter, Herman
Victor, Hema
von Solms, Basie
Wagenaar, M
Wentworth, Peter
Wheeler, Graham
Wood, Peter

UCT
Wits
UCT
UOFS
Shell
Wits
Pretoria
Simon Fraser
Natal, Pmb.
Natal, Pmb
Stellenbòsch
Wits
Council for Mineral Technology
UCT
UCT
Rhodes
UP
Stellenbosch
Fort Hare
Stellenbosch
RAU
UPE
Rhodes
UCT
UCT
6TH RESEARCH SYMPOSIUM - 1991
FINAL PROGRAM

TUESDAY 2nd July 1991

10h00 - 13h00  Registration
13h00 - 13h50  PUB LUNCH

14h00 - 15h30  SESSION 1A
Venue: Hassner
Chairman: Prof Basie von Solms

14h00 - 14h30  "A value can belong to many types."
B H Venter, University of Fort Hare

14h30 - 15h00  "A Transputer Based Embedded
Controller Development System"
M R Webster, R G Harley, D C Levy &
D R Woodward, University of Natal

15h00 - 15h30  "Improving a Control and Sequencing
Language"
G Smit and C Fair, University of Cape
Town

14h00 - 15h30  SESSION 1B
Venue: Hassner C
Chairman: Prof Roelf v d Heever

14h00 - 14h30  "Design of an Object Orientated
Framework for Optimistic Parallel
Simulation on Shared-Memory
Computers" P Machanick, University of
Witwatersrand

14h30 - 15h00  "Using Statecharts to Design and
Specify the GMA Direct-Manipulation
User Interface" L van Zijl & D Mitton,
University of Stellenbosch

15h00 - 15h30  "Product Form Solutions for Multiserver
Centres with Hierarchical Classes of
Customers" A Krzesinski, University of
Stellenbosch and R Schassberger,
Technische Universität Braunschweig

15h30 - 16h00  TEA
16h00 - 17h30 SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30 PRE-DINNER DRINKS

20h00 GALA CAPE DINNER
(Men: Jackets & ties)
**WEDNESDAY 3rd July 1991**

7h00 - 8h15  BREAKFAST

8h15 - 9h45  SESSION 3A

Venue: Hassner

Chairman: Assoc Prof P Wood

8h15 - 8h45  
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon, University of Stellenbosch

8h45 - 9h15  
"Extending Local Recovery Techniques for Distributed Databases" H L Victor & M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45  
"Analysing Routing Strategies in Sporadic Networks" S Melville, University of Natal

9h45 - 10h15  TEA

10h15 - 11h00  SESSION 4

Venue: Hassner

Chairman: Prof P S Kritzinger

Invited paper: E Coffman

11h00 - 11h10  BREAK

SESSION 3B

Venue: Hassner C

Chairman: Prof G Finnie

8h15 - 8h45  
"The Design of a Speech Synthesis System for Afrikaans" M J Wagener, University of Port Elizabeth

8h45 - 9h15  
"Expert Systems for Management Control: A Multiexpert Architecture" V Ram, University of Natal

9h15 - 9h45  
"Integrating Similarity-Based and Explanation-Based Learning" G D Oosthuizen and C Avenant, University of Pretoria
SESSION 5A

Venue: Hassner
Chairman: Prof C Bornman

11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town

11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch

12h10 - 12h40
"Building a secure database using self-protecting objects" M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B

Venue: Hassner C
Chairman: Prof A Krzesinski

11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town

11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch

12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES

Venue: Hassner
Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM
PAPERS
of the
6TH RESEARCH SYMPOSIUM
Abstract

There is, as yet, no single formal definition for the object-oriented approach. The object-oriented paradigm is described in terms of the concepts accentuated by the approach, namely classes, objects, methods, messages and class hierarchies. In view of these concepts, the object-oriented approach supports the properties of encapsulation, inheritance and polymorphism.

The object-oriented approach stands in contrast to many aspects of the relational data model traditionally used in databases. The data in a relational database is viewed as a semi-passive component, while the objects in the object-oriented approach are viewed as active components. The relational data model and the object-oriented approach also differ in many other respects.

Second generation relational database management systems do provide features which make them more object-oriented than their traditional first generation counterparts. In order to develop applications utilizing these systems, it may be necessary to consider object-oriented approaches. Ingres, as an example of a second generation relational database management system, provides some of the concepts of the object-oriented paradigm and it supports some of the properties of the object-oriented approach.

INTRODUCTION

The intent of the object-oriented approach is to provide a natural and straightforward way to describe real-world concepts and semantics. This approach allows the flexibility of expression necessary to capture the variable nature of the world being modelled and the dynamic ability to represent changing situations. A fundamental part of the naturalness of expression provided by the object-oriented approach is the ability to share data, code and definition.

At this stage, there is no single clear formal definition of object-orientation. It is a developing technology; there is as yet no agreement on the set of features and mechanisms belonging to object-orientation. It is usually just interpreted as an approach that exploits encapsulation (packaging). The object-oriented approach can, however, be described fairly accurately in terms of the concepts supported and the properties gained through this support.

The second generation relational database management systems (RDBMSs) provide features which make them more object-oriented than their traditional first generation counterparts. An example of such a second generation RDBMS is Ingres. With its Object Management Extension, it supports a degree of object-orientation. When developing methodologies for designing applications utilizing these systems [28], it is necessary to consider object-oriented approaches. It may be necessary to utilize object-oriented analysis and design approaches, should these systems prove to be substantially object-oriented.

The extent to which second generation RDBMSs provide for the concepts of the object-oriented paradigm and support of its properties are investigated in this paper. Ingres is used as the prime example, being more explicitly object-oriented than the other second generation RDBMSs. A few of the discussions refer to other RDBMSs as well. The first section addresses the object-oriented concepts and the second the properties of object-orientation. In the third
section, the second generation RDBMSs are evaluated with respect to the criticisms of the relational data model and the differences often cited between object-oriented systems and the relational data model.

1. CONCEPTS

The principle terms and concepts of the object-oriented approach include class, object, method, message and class hierarchy.

1.1 CLASS

The object-oriented approach involves classifying objects in classes according to the similarities between them. Every object is an instance of some class. A class consists of the following:

- **Name**: A symbol that identifies the class in representations or programs.
- **Set of instance variables**: A definition of the structure (fields or attributes) of all the objects of the class. Every instance variable has a name. The instance variable can be augmented with a type, which is the name of the class that determines the valid set of values for the instance variable [20, p. 193-194].
- **Set of methods**: A class, whether system- or user-defined, carries an understanding of the methods (operators) that can be applied to objects of the class. The methods defined for a class are very closely associated with the class. The instance variables of an object can only be modified indirectly by invoking the methods of its class [9, p. 689].

Thus, an object class specifies a name, a set of visible methods, a set of hidden instance variables and a set of hidden operations implementing the methods. An object has specific values for each of the instance variables defined in its object class. It shares the methods' operations with other instances of its class. The objects of a class often refer to other objects. In many systems the static properties of objects, as well as the references to other objects, are represented as instance variables. The value of an instance variable is an object itself, of a specified class [23, p. 5].

The objects of a class can be complex. A system usually provides a set of built-in primitive object classes, eg integer or float. A user can create simple object classes based on these primitive classes (see methods). Complex object classes can be constructed from combinations of existing object classes, which in turn may be complex.

Two cases of Ingres' support for object-orientation are considered in this paper. Case A addresses user-defined abstract data types (ADTs), while Case B addresses relations and views.

Case A: Through the Ingres Object Management Extension, it is possible for a user to define ADTs applicable to a particular application. For each ADT, a set of pre-specified functions, called SQL functions, have to be defined as well. These functions define the actions to be executed when the ADT is manipulated or accessed through a SQL data manipulation or query command. An Ingres user-defined ADT is analogous to a class, with the attribute values assumed for the ADT as the object occurrences. An ADT, as an object class, consists of the following [16, p. 1.4-1.6, p. 133-3.39]:

- **Name**: The name of the data type is the name of the class.
- **Instance variables**: The component fields of the ADT are the instance variables of the class.
- **Methods**: The SQL functions defined for the ADT are the methods of the class.

EXAMPLE

Consider the definition of the ADT to represent ordered pairs:

```c
typedef struct_ORD_PAIR
{
  double op_x; /* x coordinate */
  double op_y; /* y coordinate */
} ORDPAIR;
```

212
extern II_STATUS op_compare();
extern II_STATUS op_keybld();
extern II_STATUS op_getempty();

... 
II_STATUS op_compare(scb, opl, op2, result);
II_DATA_VALUE = *opl; kill
II_SCB
   *scb;
II_DATA_VALUE *opl;
II_DATA_VALUE *op2;
int
   *result
{
ORD_PAIR *opv_1 = (ORD_PAIR *) opl->da_data;
ORD_PAIR *opv_2 = (ORD_PAIR *) opl->da_data;
if = (opv_1->op_x > opv_2->op_x)
   { *result = l; }
else if (opv_1->op_x < opv_2->op_x)
   { *result = -l; }
else if (opv_1->op_y > opv_2->op_y)
   { *result = 1; }
else if (opv_1->op_y < opv_2->op_y)
   { *result = -1; }
else
   { *result = 0; }
return(II_OK);
}
...

This definition is far from complete. It still requires a plethora of functions, as prescribed in the Ingres Object Management manual, to allow full SQL support. However, it illustrates that an ADT addresses the components of a class, for example:

- **Name:** The name of the ADT is ORDPAIR.
- **Instance variables:** The instance variables of the ADT are op_x and op_y.
- **Methods:** The methods of the ADT include the SQL functions op_compare(), op_keybld() and op_getempty().

Case B: A relation (table) structure is analogous to a class, with the tuples of the relation forming the object occurrences. The relation, as an object class, consists of the following [15,p2.34-2.38]:

- **Name:** The name of the relation is the name of the class.
- **Instance variables:** The attributes of the relation are the instance variables of the class.
- **Methods:** The procedures defined on the relation are the methods of the class.

It is necessary to consider the methods, as supported by Ingres, in more detail. In Ingres, it is possible to define database procedures associated with a relation. These procedures can be called through interactive interfaces and by applications to perform operations on the data contained in relations. All the necessary data manipulation operations can be implemented through procedures. Thus in Ingres, as an example of a second generation RDBMS, it is possible to associate procedures as methods with a relation as a class [15,p1.61-1.65],[33,p350-357],[34,p897-900],[35,p5-10]. The Ingres procedures are analyzed in more detail in the subsequent section on methods.

A relation, as a complex object, may be considered as a set of tuples, a tuple as a set of scalars and each scalar as an instance of a primitive built-in class or of a user-defined ADT. The components of a complex object are also called instance variables as they appear in every instance of the class of object [9,p688].

**EXAMPLE**

Consider the definition of an Ingres relation to document rivers occurring on a map, shown with two of its procedures:
Create table RIVER
( RIVERNAME varchar(30) not null,
  COUNTRYNAME varchar(30) not null,
  RIVERLENGTH integer,
  STARTPOS ORDPAIR,
  ENDP0S ORDPAIR );

Create procedure RETRIEVE_RIVER_LENGTH
( RNAME varchar(30) not null ) as
declare
RLENGTH integer not null;
begin
select RIVERLENGTH
into :RLENGTH
from RIVER
where RIVERNAME = :RNAME;
return :RLENGTH;
end;

Create procedure UPDATE_RIVER_LENGTH
( RNAME varchar(30) not null,
  RLENGTH integer ) as
begin
update RIVER
set RIVERLENGTH = :RLENGTH
where RIVERNAME = :RNAME;
end;

These definitions illustrate that a relation adheres to the description of class as follows:
- Name: The name of the class is RIVER.
- Instance variables: The instance variables of the RIVER class are RIVERNAME, COUNTRYNAME, RIVERLENGTH, STARTPOS and ENDP0S.
- Methods: Two of the methods defined for the RIVER class are the procedures named RETRIEVE_RIVER_LENGTH and UPDATE_RIVER_LENGTH.

Note that RIVER is a complex object class. Its instance variables refer to other classes, namely STARTPOS and ENDP0S of the ORDPAIR class, as well as COUNTRYNAME of the COUNTRY class.

1.2 OBJECT

In the object-oriented approach, all conceptual entities are modelled using a single concept, namely objects. An object belongs to a class and has a state (value). An object is an instantiation of its class. When a new instance of an object class is created, it has its own set of instance variables and it shares the methods of its class with other instances of the class [20,p193],[23,p5].

Every object should have identity as part of its structure. The identity of an object distinguishes it from all other objects. In most object-oriented systems, every object has a unique system-generated object-id, guaranteed never to change. The object-id can be considered as a hidden attribute; it is not visible to users, nor can be manipulated [9,p688].

Case B: A tuple of a relation can be considered as an object. A relational tuple contains as many instance variables (attributes) as the number of instance variables of its class (relation). Each tuple in a relation can be uniquely identified, usually by the values of the attributes participating in the relation's primary key [9,p688]. In some relational systems each tuple is uniquely identified by the RDBMS through a row identifier. In Ingres, the row identifier is specified by augmenting an attribute with the system_maintained clause. This system_maintained
attribute is system-generated and cannot be changed by a user. It can merely be accessed and used in cross references [15,p2.34-2.36].

EXAMPLE

Consider the following tuple as an example object of the RIVER class:

< "Nile", "Egypt", 4145, "(31,0)", "(32,32)" >

This object contains values for all the instance variables of the RIVER class. Assuming river names are unique, it is identified by the instance variable RIVERNAME. It can also be identified by an internal system_maintained attribute.

1.3 METHOD

The objects of each class have active properties specifying their allowed behavior. A method is an operation or a function which can be performed on objects of a particular class with which the method is associated. The set of methods defined for a class forms part of the definition of the class. The class (name) and its methods form the "public interface" for the objects of the class. The only way to operate on the objects of a class is through the methods defined for that class [9,p689].

Similar to built-in objects, there are usually a set of primitive methods for fundamental operations, eg scalar comparisons, arithmetic operations, string manipulation operations, etc. Some of the built-in methods are generic; they are understood by every class. In some systems, a generic function defines the public interface, while a method for each class supporting the interface is an implementation of that interface.

According to Moon, a method has five properties [22,p55]:

- A generic function that it specializes; thus a public interface supported by the method.
- An applicability condition, indicating when the method is appropriate when an object is passed as an argument to the generic function.
- Qualifiers that identify the method's role, where applicable.
- A parameter list that receives the arguments.
- The body of code executed when the method is called.

Case A: Where an ADT is considered as a class, the SQL functions defined for the ADT form the methods of the class. In the Ingres implementation, the instances of the ADTs can only be accessed and manipulated through the SQL functions defined as part of the ADT.

EXAMPLE

In the ORD_PAIR class defined in a previous example, the SQL functions op_compare(), op_keybld() and op_getempty() are some of the methods defined for the class.

Case B: Where a relation is considered as a class, the procedures defined on the relation would then be the methods defined for the objects of the class. As stated, the objects of the relation class would be the tuples of the relation. The tuples of the relation should then only be accessed and manipulated through the methods defined for the class. In the Ingres implementation it is possible to eliminate access to relations through the standard SQL data manipulation commands. This can be done through security control mechanisms; the activator of a procedure can have different security-based capabilities for the procedure and the relation.
EXAMPLE

The two procedures RETRIEVE_RIVER_LENGTH and UPDATE_RIVER_LENGTH defined in a previous example are examples of methods of the RIVER class.

The RETRIEVE_RIVER_LENGTH method satisfies Moon’s characterization as follows:

- **Generic function:** The method specializes the generic SQL "select" operation.
- **Applicability condition:** The method is applied to a particular object, namely the instance (tuple) of the RIVER relation with the same RIVERNAME as identified in the parameter list.
- **Qualifiers:** The only qualifier applicable to this method is the "not null" clause on the RNAM parameter.
- **Parameter list:** The parameter list (RNAME, RLENGTH) defines the values interchanged between the method and the calling mechanism.
- **Body:** The body of the method is the "select" statement between the "begin" and "end" statements. It is not visible to the calling mechanism.

1.4 MESSAGE

Objects communicate and perform all computations via messages. A message is a specification to activate one of an object’s methods. In order to apply a given method to a given object, it is necessary to send a message to the class of the object. The message is dispatched to the method by the class of the receiving object, to be applied to the indicated object. The behavior of an object is captured in the messages to which an object responds. The messages completely define the operational semantics of an object [9,p689],[13,p522],[22,p69].

A message to an object requests that it should carry out a specified operation defined as a method, but the message does not indicate how the operation should be performed. The receiving object class contains the method which specifies how the operation should be performed. The sender, presuming objects are intelligent, trusts the receiver to perform its request correctly. This "call-by-desire" notion is central to the object-oriented approach [13,p163],[22,p24]. When an object receives a message, it executes the method activated by the message and returns an object as a result to the sender [9,p689].

A message is a syntactic construct, usually consisting of three parts [13,p522]:

- An identification of the object singled out as the receiver of the message.
- The name of the method (operation) to be performed, also called the selector. It is merely a symbolic name for the actual operation to be performed.
- A list of arguments, namely additional objects or values passed as parameters.

The set of messages an object can respond to is called its protocol. The external view of an object is nothing more than its protocol [29,p7].

Case A: The activation of the SQL functions defined as part of an Ingres user-defined ADT can be considered a type of message. Whenever a data manipulation command is applied to a relation, the corresponding SQL functions are activated. The data manipulation message sent to the relation class causes a message to be sent to the underlying ADT class. The activated SQL function is the selector; it specifies the method to be executed. The affected object occurrences are specified by the SQL data manipulation command. The arguments are utilized as specified in the function definition.

Case B: The procedure call mechanism implemented in Ingres can also be considered a type of message. The procedure call explicitly names the selector through the name of the procedure. The exact objects to be affected by the message have to be identified through the value parameters passed to the procedure. The parameters can also be used to pass values utilized in the execution of the method associated with the procedure.
EXAMPLE

Consider the following activation of the UPDATE_RIVER_LENGTH procedure from the UPDATE_RIVER rule:

Create rule UPDATE_RIVER
   after update(RIVERLENGTH) of RIVER
   execute procedure UPDATE_RIVER_LENGTH
   ( RNAME = new.RIVERNAME,
     RLENGTH = new.RIVERLENGTH );

This rule is fired whenever the RIVERLENGTH attribute of the RIVER relation is updated. It activates the UPDATE_RIVER_LENGTH procedure to perform the update. It can be considered as sending a message to the RIVER class to activate the UPDATE_RIVER_LENGTH method.

With a security-strict Ingres implementation, as described previously, users cannot access the tuples of relations through SQL data manipulation commands. The protocol of a class (relation) therefore only consists of the set of defined messages (procedure names).

1.5 CLASS HIERARCHY

Classes can be related hierarchically to each other. If class B is a subclass of class A, then every instance of B is automatically an instance of A, but the converse is not true. The class hierarchy captures the IS-A relationship between a class and its subclass. The objects of related classes share properties through inheritance. All subclasses of a class inherit all properties defined for the classes and can have additional properties local to them. The subclass is a specialization of the class and the class is a generalization of its subclasses. The implication of a class hierarchy is that if class B is a subclass of class A, an object of class B may be used wherever an object of class A may be used. Subclass B of class A will automatically inherit features from A [1,p4-6],[14,p21]. Inheritance is discussed in more detail in a subsequent section, under the properties of the object-oriented approach.

Case A: The user-defined ADTs of Ingres do not support class hierarchies. An instance of one ADT cannot be an instance of another ADT. One ADT can be used in the definition another ADT, but that forms an IS-PART-OF relationship encountered in complex objects and not an IS-A relationship.

Case B: Relation B forms a subclass of another relation A if the set of tuples of B form a subset of the set of tuples of A. This is typically implemented through select views in second generation RDBMSs. The set of tuples visible through a view B, defined on a relation or view A, form a horizontal subset of the set of tuples visible through A.

EXAMPLE

In Ingres, a view is defined as a virtual relation, based on a “select” retrieval from either a base relation or another view. Thus, a subclass LONGRIVER can be defined for the class RIVER:

Create view LONGRIVER as
   select RIVERNAME, COUNTRYNAME, RIVERLENGTH,
       STARTPOS, ENDPOS
   from RIVER
   where RIVERLENGTH > 500
   with check option;

This LONGRIVER subclass (view) inherits all the specified instance variables (attributes) from the RIVER class (relation). Note the attributes are not automatically inherited; they have to be specified. Only those objects (tuples) with a RIVERLENGTH longer than 500 will be instances of the subclass, but all instances of LONGRIVER will also be instances of RIVER.
A view is a virtual relation. This is particularly valid in some other second generation RDBMSs, such as PowerHouse StarBase, where fully updatable views can be defined [6]. In such systems, views can be used in all data manipulation and data definition commands where relations can be used. This is not valid for all Ingres views.

2. PROPERTIES

The important properties of the object-oriented approach can be classified under encapsulation, inheritance and polymorphism.

2.1 ENCAPSULATION

Central to the object-oriented approach is the notion that the objects of interest in the real world can be modelled most effectively through recording them together with the operations that are permitted on them [3, p41]. Encapsulation enables the restriction of the effects of change on software. It provides a strong form of data independence. By including the methods with the definition of an object class and by only providing access to the objects of the class through the methods, it hides implementation and representation details from users. This allows the representation details to be changed, if necessary, in a controlled manner without affecting applications. This also enforces disciplined access to objects, only through the public interfaces provided by the methods defined for the class of the object [9, p689-700], [24, p263].

Case A: The implementation of ADTs in Ingres provides a high degree of encapsulation. The SQL functions applicable to an ADT are defined and recorded together with the structure of the ADT. The instances of the ADTs can only be manipulated through the SQL functions encapsulated with the ADT.

Due to its high degree of encapsulation, the ADTs as implemented in Ingres, provide a high degree of data independence. The actual storage structures can be changed. By changing the corresponding SQL functions, the applications utilizing the ADTs can remain unaffected by the changes.

Case B: The Ingres relations and views, if considered as object-oriented, only support a limited degree of encapsulation. It has been illustrated that classes can be defined, with methods inherent to the class. If not properly controlled, the objects (tuples), as instances of the classes (relations or views), can also be accessed through other mechanisms than just the methods (procedures) associated with the classes. The tuples can also be accessed through standard, generic SQL data manipulation commands.

In the Ingres implementation, if all applications only access the data stored in the database through procedures, a strong form of data independence can be achieved. This data independence could be improved even more through the use of updatable views, not presently available in Ingres. The representations of the objects (tuples) of a class could then be changed. By changing the corresponding procedures, the applications could remain unimpaired.

2.2 INHERITANCE

Inheritance is the concept used in object-oriented approaches to define objects that are almost like other objects, possibly with a few incremental changes. Inheritance makes it possible to declare that certain specifications are shared by multiple parts of a system. Inheritance allows classes to pass properties to each other. A subclass may share or inherit various characteristics of its superclass. These characteristics usually include the storage representation of the supertypes and the methods provided by the supertype [31, p188].

The advantage of inheritance is that it reduces the need to specify redundant information and it simplifies updating. Information about many object instances can be changed and entered in a single update action [13, p523].

Case A: Because the Ingres ADTs do not support class hierarchies, they do not provide inheritance properties either.
Case B: The implementation of updatable views in second generation RDBMSs such as Ingres provide a high degree of inheritance. It is standard procedure to define a view which is almost like a relation or almost like another view. A view defined on a base relation or another view also inherits a number of properties of the base relation or the view, as described subsequently. With a view being a virtual relation, merely a materialization of its underlying base relation, it shares many properties of the base relation.

2.2.1 Types of inheritance

Two types of inheritance have been identified, namely structural inheritance and behavioral inheritance [9,p690]:

- With structural inheritance subclass B automatically inherits all the instance variables of class A. Class B might have additional instance variables of its own which A does not have, as described in the next section on property modification.
- With behavioral inheritance subclass B automatically inherits all the methods that apply to class A. Class B might have additional methods of its own which do not apply to A, as described in the next section.

Case B: In relational views, the following inheritances are applicable:

- Structural inheritance: View B only inherits the attributes of A included in its definition. It can only define limited new attributes, namely constants and attributes derived from the attributes of A.
- Behavioral inheritance: Procedures cannot be defined for views, thus a view cannot inherit methods.

2.2.2 Property modification

The characteristics of a class inherited by a subclass may be changed or overridden in the subclass. A subclass may re-specify the storage representation or give new definitions for methods inherited from the supertype [14,p12],[24,p263]. A specialized class can modify the properties inherited from its superclass through additions and / or substitution [31,p189]:

- Addition allows the introduction of new variables, properties or methods in a class, which do not appear in one of its superclasses in the hierarchy.
- Substitution (or overriding) is the specification of a new value of a variable or property, or a new method for a selector that already appears in some superclass. All descriptions in a class, namely variables, properties and methods, are inherited by a subclass unless overridden in the subclass.

Case B: In relational views, class modification is supported as follows:

- Addition: A view can have additional attributes (instance variables), for example derived attributes. It cannot have additional procedures (methods) only applicable to the view.
- Substitution: An attribute can be renamed in a view, thus a derived attribute can be renamed to override its base relation definition. Methods cannot be applied to a view.

2.2.3 Source of inheritance

The source of inheritance indicates the superclasses from which a class inherits its properties. Class inheritance can be classified under single inheritance, multiple inheritance and partial inheritance [23,p7-8]. For both single and multiple inheritance, the following rule applies: All properties of a class, namely its instance variables and methods, are inherited by a subclass, unless overridden in the subclass. It should be possible to override inherited methods, eg a display method may have to display additional instance variables [13,p523].

With single inheritance, a subclass may inherit instance variables and methods of a single parent class, possibly adding some instance variables and methods of its own [23,p7].

Case B: Single inheritance occurs in a select view or in a view exactly mapped on a single base relation. The LONGRIVER view illustrated previously is an example of a subclass with single inheritance.
In many object-oriented systems, class hierarchies are essentially hierarchical in nature, thus only supporting simple inheritance. There is no direct support for associations, e.g., many-to-many relationships [9, p.689]. In other systems, a class can have more than one superclass. Through multiple inheritance a subclass can then inherit properties from more than one class. Multiple inheritance is a natural extension to single inheritance, where a union of instance variables and methods are inherited from multiple parent classes. Then the class hierarchy is generalized to a class lattice, as represented by a directed acyclic graph structure. The class lattice often simplifies data modelling and requires fewer classes to be specified than required with a class hierarchy [1, p.7], [23, p.7].

Name clashes can occur in the presence of multiple class inheritance. Two similarly named, but differently implemented methods can be inherited from two parent classes. The same applies to the inheritance of identically named instance variables. In new methods such clashes can be distinguished by prefixing the inherited method or instance variable with the name of the parent. However, such clashes are most pronounced when class hierarchies are changed. It must be considered how the changes to the definition or implementation of a class will affect inheriting subclasses. There are two types of name clashes that can occur in a class lattice. In most systems these naming conflicts are resolved by precedence:

- A naming conflict can occur between a class and its superclass. This can also happen in a class hierarchy. The precedence is usually given to the definition of the class, overriding the definition of the superclass.
- A naming conflict can occur between the properties of the superclasses of a class. The superclasses of a class are usually represented as a list. The precedence is usually given to the definitions of the superclasses occurring earlier in the list. Alternatively, conflicting properties may be explicitly renamed in the definition.

Case B: Consider a join view, containing all the attributes of the underlying base relations. It is the typical example of a subclass supporting multiple inheritance. The view inherits the attributes from both the underlying base relations. In the Ingres implementation, as in most other relational systems, naming conflicts have to be explicitly resolved in the view definition. The context of attributes occurring in both underlying base relations have to be indicated, by prefixing the attribute name with the relation name in the view definition.

EXAMPLE

Consider the relation RIVER shown in a previous example, together with the following relation:

COUNTRY (COUNTRYNAME, CONTINENT).

The following view has multiple inheritance properties:

Create view RIVERLOCATION as

```sql
select RIVERNAME, RIVER.COUNTRYNAME, CONTINENT,
      RIVERLENGTH, STARTPOS, ENDP,POS
from RIVER, COUNTRY
where RIVER.COUNTRYNAME = COUNTRY.COUNTRYNAME;
```

Note the explicit name resolution by augmenting the COUNTRYNAME attribute name with the RIVER relation name. COUNTRYNAME occurs in both the RIVER and CONTINENT relations. Note such a join view is not updatable in Ingres.

Partial inheritance is when some properties are inherited and others are suppressed. In such a case neither class is a subclass of the other, but they are undeniably related. Partial inheritance is convenient for aspects like code sharing, but it can create a mess in a class hierarchy [23, p.8].

Case B: A project-join view is the typical example of a subclass with multiple, but partial inheritance. Although the view inherits properties from both its underlying base relations, it does not inherit all the properties, as some attributes can be eliminated through the project part of the specification. Similar to the problems encountered in a class hierarchy or class lattice, a non-key preserving project-join view can introduce many problems. One such problem is that view may not necessarily be updatable [7, chap 17].
2.3 POLYMORPHISM

In general, polymorphism means "having or assuming different forms". In the context of object-orientation, polymorphism refers to the capability of different classes of objects to respond to exactly the same message protocols. The protocols enable a system to treat objects that arise from different classes uniformly. Message protocols extend the notion of modularity (reusability) to polymorphism (interchangeability) [31,p183]. Polymorphism is closely related to class inheritance. The same operations that apply to a parent class also apply to instances of its subclasses [23,p10].

The term overloading is also used for polymorphism. Operator overloading means that the same operator symbols can be used for the same operations on different data types. Thus, distinct methods can be given the same name for distinct classes. Overloading is performed when the definition of a method that applies to class A is extended or even totally changed to perform additional or totally different operations when applied to an object of a subclass B of class A. The operator is therefore overloaded and its true meaning can only be resolved by looking at its operand types in the context of the class where overloading is applied [9,p690],[13,p522].

A polymorphic function can be applied uniformly to a variety of objects. The "same" operation maintains its behavior transparently for different argument types. With ad hoc polymorphism or "mere" overloading, two operations coincidentally share the same name, but otherwise have completely different behaviors [23,p10].

Case A: The user-defined ADTs defined in Ingres support polymorphism to a large extent. For every ADT, the same SQL functions are defined. These "same" functions behave similarly; they perform the same operations on the ADT, but the implementation of a function for each different ADT can differ substantially. Thus the behavior (execution) of the method (SQL function) is determined by the context of its class (ADT).

Case B: The procedures used with relations do not support polymorphism at all. This is due to the fact that each procedure is required to have a distinct name.

3. EVALUATION

It has been stated that object-oriented databases have advantages over databases based on the traditional data models [13,p524-525]. This section evaluates the object-orientendness of the second generation RDBMSs with respect to the criticisms often cited against the relational data model, as well as in terms of the major differences usually cited between the object-oriented approaches and the relational data model.

3.1 PASSIVE VS ACTIVE

The main difference between a traditional (relational) DBMS and an object-oriented DBMS is in the passive and active behavior of the underlying database. A traditional database is passive, it embodies a collection of structured data, such as data contained in relations. The data to be processed and manipulated are accessed by applications via the DBMS. Some of the data are temporarily stored and processed in application program data structures; the results of the actions are often written back to the database. In the object-oriented approach, the database contains objects that are made up of both passive data objects and active methods to reflect the behavior of the data. Once a user request is transmitted to the object base, the objects respond to the request in a way consistent with their defined behavior. The methods take the necessary action and invoke other methods through messages to complete the request. The entire request is completely performed within the object base [24,p262-263].

In most of the second generation RDBMSs, for example Ingres, Sybase and StarBase, dynamic properties are considered part of the database. In these systems, functionality in the form of procedures, rules and triggers are defined as part of the database. These active components are automatically activated and executed when predefined conditions occur [27,p183-200].
3.2 APPLICATION DOMAIN

The simplicity of the relational data model is dependent on its use in modeling applications where the data is confined to a small number of different types of data related in well-defined ways. Its use can be limited when applied to complex, highly structured application domains [3,p39].

With the advent of advanced applications like engineering, design, office or knowledge-based systems, stored entities tend to show very complex internal structures. These structures may overlap and may comprise large numbers of, possibly substructured, properties. Relational systems impose the first normal form constraint. This means that the object space must be mapped into a collection of flat relations. Using relational record-oriented data models for the applications listed above result in awkward database designs and complex retrieval operations. With this approach, much of the inherent semantics of complex object composition is lost. Foreign key joins have to be performed to reconstruct complex objects. It has been stated that the simple relational record structure, with atomic attributes, can be considered a major disadvantage of relational systems. It is difficult to express the semantics of complex objects in this fashion [3,p39],[12,p54],[17,p275]. Various solutions to these problems have been proposed. Some current approaches preserve the relational model [36], some add new features including user-defined types and procedures [32], while others abandon the relational model in favor of object-oriented approaches [1],[32]. While some parties state that a relational system can be made object-oriented, others state that object-orientation, including object identity and encapsulation, cannot co-exist with the declarative access to data and flexibility found in RDBMS [11,p179].

The second generation RDBMSs already support many more object-oriented concepts and have many more of the properties associated with object-oriented systems than their first generation counterparts. The extent of this object-orientation is discussed in the conclusion. In addition to object-orientation per se, these systems support much more advanced data types. While Ingres supports ADTs [16], systems like Informix and StarBase support BLOBs and other multi-media data types [6]. With these data types, it is possible to store the data typically encountered in the application domains mentioned above. These include text, graphical and voice images, which can easily be accommodated in BLOBs.

3.3 INTEGRITY SEMANTICS

There seems to be controversy regarding the support of semantics related to integrity control.

Advocates of the object-oriented approaches state that it is impossible to capture and control much of the semantics related to the integrity control of a complex application with such a simple framework as the basic relational model. They state that although the relational model enforces referential integrity, it has no mechanism for distinguishing and enforcing the different types of relationship that may exist between entities, such as association (many-to-many), designation (one-to-many) and characterization (one-to-many existence dependent) links. If such a distinction were to be made, it would be possible to define the semantics of operations to create and delete relationships differently for each case [3,p39].

Garey & Jackson reiterated the point by stating that the object-oriented "data model" differs from the relational data model in the following respects [13,p525]:

- In the relational data model tuples in a relation are explicitly identified through primary key attributes. In the object-oriented approaches, an internal identifier is assigned to each object occurrence. Relationships are implemented using the internal identifiers. Any key changes are immediately visible to all objects, while in the relational model primary key updates have to be propagated to all foreign keys.
- In the object-oriented approaches, referential integrity can be achieved using the hidden identifiers, as relationships are explicitly defined. In the relational systems relationships are implicitly defined through foreign keys.
- An object-oriented schema captures integrity constraints, by recording properties of the entities in the system in a data dictionary. Object methods can be viewed as functions, allowing a more precise notation for specifying constraints.
These statements are not valid in view of the extensive facilities provided by second generation RDBMSs. For example, in the Database Application Development approach, the different types of relationships, with more semantics than mentioned above, can be modelled and can eventually be incorporated in the database definition. The integrity controls associated with these relationship types are then continuously and automatically enforced by the RDBMS [28,p103-213].

Advocates of the “pure” relational data model state that the object-oriented approach has shortcomings related to integrity control. The object-oriented user effectively has to understand three forms of different “referential integrity” control mechanisms [9,p702-703]:

- Embedding object B within object A.
- The traditional foreign key mechanism, through messages and methods.
- The class hierarchy concept.

They state that the implications of choosing one of these three representations over another is not clear. The asymmetry that such considerations give rise to is unfortunate. Even more, the object-oriented approach does not provide convenient declarative means of stating the referential integrity constraints or the actions to be performed on the violation of the constraints. For example, the operation corresponding to cascading delete does not even exist.

3.4 REAL WORLD SEMANTICS

It has been stated that due to the rigid framework of traditional data models, a semantic gap between the modelled world and its database representation usually cannot be avoided. It is impossible to represent all the semantics of interest in the database. The “remainder” has to be captured by the application programs using the database or by the interpretation of the result of database queries by the user self [12,p53].

The same comments apply here as stated under integrity semantics. Numerous application semantics, specifically related to data relationships, constraints and active semantics control can be modelled during database application design. These can then be incorporated in the database definition, where they are automatically activated and enforced [28, 103-213].

3.5 SCHEMA CHANGES

Banerjee et al stated that most existing relational systems allow only a few types of schema changes. They only allow the dynamic creation and deletion of relations (classes) and the addition of new columns (instance variables) in a relation. This is because the conventional record-oriented business applications they support do not require more than a few types of schema changes; also the data models they support are not as rich as object-oriented data models [1,p10-11].

In the object-oriented approaches, the database schema consists of the class definitions and the inheritance structure of the class lattice. Schema evolution is the ability to dynamically make changes to this schema. Banerjee et al classified schema changes according to a complete taxonomy, based on the representation of a class lattice as a graph. The essentials of the taxonomy is that it should be possible to add, drop and change classes, instance variables (descriptors and relationships), methods and inheritance [1,p10-16].

The set-oriented paradigm of the relational data model is exploited in RDBMSs to provide adaptability. The RDBMSs, for example Ingres, allow considerable schema and view changes. Columns can be added or deleted from relations (both descriptive attributes and foreign keys). Procedures can also be added and deleted from the database schema [15,p2.67-2.71].

Advocates of the relational data model state that although the object-oriented schemas provides support for version management and changes in databases, it has schema related problems. The mentioned drawbacks of the object-oriented approach, in contrast to the traditional data models, are the following [13,p525]:

- The object-oriented data models are more difficult to implement.
3.7 IMPEDANCE MISMATCH

Object-oriented databases alleviate the impedance mismatch between programming languages and DBMSs. In conventional database systems there is a mismatch between conventional record-at-a-time third generation programming languages and the set-at-a-time data manipulation language supported by the RDBMSs. Object-oriented databases are more complete in that they provide the necessary expressive power to perform the computations of an application through object-oriented programming languages [17,p274-275].

In some of the languages of the second generation RDBMSs, this impedance mismatch is reduced considerably. For example, the GDML language used in StarBase has built-in looping and set handling constructs. GDML can be used in a standard programming language such as C, Cobol or Pascal, to process a set of relational tuples without having to utilize a cursor [5,p2.7-2.12].

3.7 QUERIES

There will always be the need to access data in unforeseen ways for ad hoc queries. Object-oriented systems seem too rigid in only allowing operations through predefined methods [9,p701]. In addition, many of the object-oriented systems do not include the notion of "the set of all instances of a class", which is fundamental in database management systems [4,p17,p30],[9,p701].

Accessing data by means of user-defined methods, instead of (standard) system-supplied methods seems contradictory to the objective of optimization. The record-at-a-time flavor of object-oriented systems means that query optimization as found in relational (set-at-a-time) systems is impossible. To quote Date, "the record-at-a-time approach of object-oriented programs is a throwback to pre-relational systems". The notion of an object-oriented query language does not seem viable either. Indexing in an object-oriented system seems even more complex [9,p702-704].

CONCLUSION

This paper documents the results of an investigation into the object-orientation supported by second generation RDBMSs, as the background for developing application design methodologies for these systems. In particular, it addresses the ADTs, relations and views as implemented in Ingres. These results are summarized in the following tables:

<table>
<thead>
<tr>
<th>Concept</th>
<th>ADT</th>
<th>Relations &amp; Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Object</td>
<td>Partly</td>
<td>Supported</td>
</tr>
<tr>
<td>Method</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Message</td>
<td>Partly</td>
<td>Implicitly</td>
</tr>
<tr>
<td>Class hierarchy</td>
<td>Not supported</td>
<td>Read only</td>
</tr>
<tr>
<td>Property</td>
<td>ADT</td>
<td>Relations &amp; Views</td>
</tr>
<tr>
<td>---------------</td>
<td>-----</td>
<td>-------------------</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>High</td>
<td>Limited*</td>
</tr>
<tr>
<td>Data independence</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Inheritance</td>
<td>None</td>
<td>Limited</td>
</tr>
<tr>
<td>Structural</td>
<td>None</td>
<td>As defined</td>
</tr>
<tr>
<td>Behavioral</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Addition</td>
<td>None</td>
<td>Limited*</td>
</tr>
<tr>
<td>Substitution</td>
<td>None</td>
<td>Limited</td>
</tr>
<tr>
<td>Single</td>
<td>None</td>
<td>Limited</td>
</tr>
<tr>
<td>Multiple</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Partial</td>
<td>None</td>
<td>High</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>High</td>
<td>None</td>
</tr>
</tbody>
</table>

The following notes are applicable to the tables:

1. Values are validated through functions defined for domain integrity, but occurrences are not uniquely identified. There can be many occurrences of the same object.
2. Objects are not explicitly identified as the targets of messages. The objects affected by a message are implied by the context of the message in terms of a higher class message, namely the SQL data manipulation language command.
3. Encapsulation applied to a relation or view is dependent on proper security enforcement.
4. Views and relations have limited instance variable addition; they can only add attributes derived from the attributes of an underlying base relation or view.

Thus the second generation RDBMSs, such as Ingres, support a reasonable degree of object-orientation. This indicates that aspects of object-orientation could be considered when developing methodologies for designing applications targeted for these systems. There are, however, many object-oriented application analysis and design approaches; only a limited number of these seem appropriate. According to the results presented in this paper, approaches which stress the object-oriented concepts, together with encapsulation, should be considered. Approaches based on or stressing inheritance and polymorphism may not be applicable.

References


[27] MH Rennhackkamp, Relational Database Triggers, In [19].


[29] D Robson, Object-Oriented Software Systems, In [25].


226