Frank, A. U., and M. Egenhofer. "Object-Oriented Database Technology for GIS." San Antonio, Texas, USA: Department of Civil Engineering, 1988.

OBJECT-ORIENTED DATABASE TECHNOLOGY FOR GIS

Andrew Frank
Max J. Egenhofer

Report 95

OBJECT-ORIENTED DATABASE TECHNOLOGY FOR GIS

Andrew U. Frank
Max J. Egenhofer

National Center For
Geographic Information And Analysis
University Of Maine
107 Boardman Hall
Orono, Me 04469

OBJECT-ORIENTED DATABASE TECHNOLOGY FOR GIS

Andrew Frank
Max J. Engenhofer

University of Maine
Dept. of Surveying Engineering
107 Boardman Hall
Orono, ME 04469

Manual for workshop delivered at GIS/LIS '88, San Antonio, Texas, November 29, 1988.

Workshop Overview:

- Session 1: Background
  - Introduction
  - Current and upcoming GIS technology
  - Conventional DBMS for GIS
- Session 2: Object-oriented data models and programming
  - Data models
  - Abstraction mechanisms
  - Software engineering concerns
  - Multi-sorted algebras and abstract data types
  - Object-oriented programming languages
- Session 3: Object-oriented DBMS
  - DBMS functionalities
  - Conceptual Schema Design
  - Data Definition Language
  - Complex Objects and Value Types
  - Object-oriented programmer's interfaces
  - Complex consistency constraints
- Session 4: Architecture and query languages
  - Layered architecture
  - Object-oriented query languages
  - Existing object-oriented DBMS
  - Education
  - Conclusion

Session 1: Background

- Introduction
- Current and upcoming GIS technology
- Conventional DBMS for GIS

Geographic Information Systems a.k.a.

- Land Information Systems (LIS)
- Multi-purpose cadastre

We prefer in this context the generic term "spatial information systems".

GIS are complex software systems to manage spatially related data.

Spatially related data has two components:

- a description of the location and extension of the objects,
- attribute data describing non-geometric properties of objects.

Spatial concepts

Humans use multiple concepts concurrently.

Need for formalization in order to program.

Fundamental concepts used in current GIS:

- raster
- vector

Topology-based vector systems:

- graphs (points and lines)
- cells (points, lines, and areas)
- subdivisions (points, lines, areas, and (partial) ordering)

A GIS consists of

- Organization
- Software
- Hardware

Hardware development will continue to be fast.
Hardware is no impediment.

Software moves slower.
The object-oriented paradigm for software engineering is the major new concept.

Changing organizations is extremely slow and difficult.
Adapt technical system to organization.

GIS builds a model of some aspects of reality. The ease of designing such a model is crucial to constructing GIS.

There are four trends in computer science investigating object-orientation:

- Modeling,
- Programing languages,
- Database management systems,
- Artificial intelligence.

Future GIS need all four components.

The GIS software consists of:

- A component to manage storage and retrieval of data.
- Application programs that use the data.
- Computer graphics to produce map output.
- A human interface system to interact with the users.

GIS and other areas, like Computer Aided Design (CAD) and Computer Aided Engineering (CAE) have had similar problems:

- Inadequate traditional programming methods for managing geometric data.
- Complexity of spatial relationships.
- Algorithmic difficulties (computational geometry).

Current GIS technology:

- Organization of data in map sheets.
- Use of dedicated file structures for geometric data.
- Separation of attribute data (in files or connected DBMS).

Goal:

- A seamless database.
- A single DBMS for spatial and non-spatial data.

A GIS can be used to support

- planning and policy setting
- administration

GIS for planning and policy setting can tolerate (limited) inaccurracies. They are often built for small areas and for a specific project; data is not maintained and updated.

Effective graphical presentations are very important.

GIS for administrative purposes must

- be precise,
- completely cover an area,
- be updated.

GIS are often used as 'map maintenance systems'.

The system stores an image of the map and is used for editing. The users are provided with updated maps.

The tendency is towards interactive usage (dialog).

Concepts:

- Paperless office
- Maps on demand

GIS data are recognized as valuable 'integrated resource' for a community.

- long term usability
  - updates
  - consistency
- integration of mutiple users
  - complex data models
  - concurrent users and access control
  - distributed databases

Database management systems (DBMS) are necessary components of the next generation of GIS.

Database Concept

A central repository for data accessible for many application programs through a standardized interface.

Benefits

Independence of the resource from the application program.
One program can be changed without affecting the database or other programs.

Conventional DBMS

Designed to manage commercial data with weakly structured data.

Existing DBMS provide a limited number of fundamental data types
(integer, real, string of character).

They are not sufficient for most applications.

Extensions for commercial programs (date, money) are available.

More complex data types (points, lines, complex numbers) for GIS and other scientific applications are not available.

The relational data model is built on the simple concept of a table with a few relational operators:

- selection
- projection
- Cartesian product
- set union
- set difference

This data model is not adequate to model complex real situations. (distance user concepts - tools)

Relational algebra is table-oriented.

number	name	location	budget
3	Jones	Orono	100,000
6	Smith	Bangor	80,000
9	Miller	Bangor	135,000

Current programming languages are record oriented. Use of relational databases from regular program code is difficult and sometimes cumbersome.

Data structures

Relational DBMS force separation of data in smallest parts (normalization rules).
The re-building of complex objects from such parts reduces performance.

Performance

The typical user interaction with a GIS results in a map sketched on a terminal. This requires that the database retrieves 2000 ... 5000 objects to be drawn.
In a relational implementation of a database this results in several thousands of accesses (each 30 millisec.).

Consistency constraints

Geometric data must obey special consistency constraints e.g., two parcels must not overlap.
Relational DBMS do not address the complex consistency constraints for geometric data.

Transactions

Changes in a GIS may require a large number of user interactions and a long time for completion.
Current databases use a single 'transaction' concept that serves multiple purposes.

Transaction concept (ACID)

- Atomicity
- Consistency
- Isolation
- Durability

GIS need

- versions and
- long transactions.

Long transactions can be implemented on top of regular (short) transactions.

Summary Session 1:

DBMS are needed for GIS to manage

- consistency
- multi-user access
- transactions

Relational DBMS provide insufficient

- data structures
- performance

Session 2: Object-oriented data models and programming

- Data models
- Abstraction mechanisms
- Software engineering concerns
- Multi-sorted algebras and abstract data types
- Object-oriented programming languages

A major shortcoming of current DBMS is their lack of integration into programming languages and software engineering methods.

- Separation of data description and application program.
- Data manipulation with commands which are not fully integrated into the language (precompiler, embedded SQL and QUEL)

The object-oriented paradigm is an encompassing concept, including

- software engineering
- programming languages, and
- databases

A discussion of an object-oriented database must include a discussion of object-oriented programming.

An information system is a model of some aspects of reality.
The data model provides the means to describe the data in an information system.

The fundamental abstraction mechanisms are:

- Classification
- Generalization
- Aggregation

Classification is the abstraction from individuals with common properties to a class (instance_of-relation).

Example: from 'P. Miller', 'J. Doe', etc. to the class 'person'.
'P. Miller' is an instance of the class 'person'.

Generalization is the combination of several classes to a more general superclass (is_a-relation).

Example: from the classes 'dog', 'cow', 'fox', etc. to the superclass 'animal'.
Every dog is an animal. Every dog has all properties of an animal (inheritance).

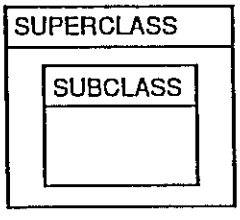
Aggregation groups multiple individuals to a new (complex) object (part_of-relation).

Example: 'J. Doe' and 'N. Allan' are working on the 'Project X'.
Some authors differentiate between aggregation and association.

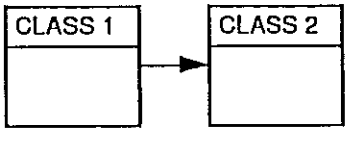
All these abstraction methods are necessary to adequately model complex systems.
Missing methods can be replaced (at the cost of more complex programs).
The relational data model does not provide generalization.

Graphical Methods for Representation

Classification



Generalization



Aggregation

Exercise 1

Design a model with the following knowledge: Rome, Australia, Mexico City, Germany, Europe, Boston, Los Angeles, Sydney, New York, USA, Paris, America, Mexico, London, Australia, Italy, England, France, Berlin, Hamburg

- Group similar objects to classes.
- How are these classes related?

Solution Exercise 1

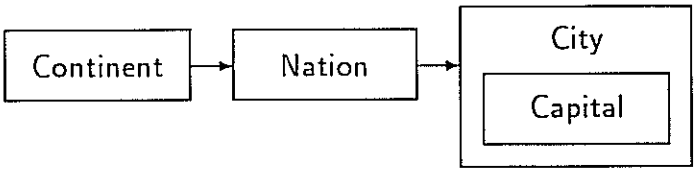
Classes:

- Continents: Australia, Europe, America
- Nations: Germany, USA, Mexcio, Australia, Italy, England, France
- Cities: Rome, Mexico City, Boston, Los Angeles, Sydney, New York, Paris, London, Berlin, Hamburg
- Capitals: Rome, Mexico City, Paris, London

Relations:

- Aggregate: Continent-Nation
- Association: Nation-City
- Generalization: City-Capital

Graphical model:



Software engineering discusses methods for

- design
- implementation
- maintenance
- management

of software systems.

Complexity of coding:

- Assembler vs. high-level programming language:
  Programming productivity 5 : 1
- High-level programming language vs. object-oriented language:
  Programming productivity 5 : 1

Current programming methods (structured programming etc.) deal with problems of 'programming in the small.'
'Programming in the large' deals with the construction of large programs.

Tools for programming in the large:

- Encapsulation (information hiding).
- Modularization.
- Independent compilation with type checking across modules.
- Intelligent linker and builder.

Programming in the large encourages reusability:

- avoid redundancy
- reduce size of software systems
- well-defined interfaces
- specifications
- faster implementations
- better maintenance

Reusabilty requires code management systems for information about existing modules.

GIS are very large software systems and need these tools.
Object-orientation specifically addresses

- the design of modules,
- the interfaces between modules.
- the type structure.

The object-oriented concept

Encapsulate a data object with the pertinent operations in a single unit.
This data object may represent a 'real world object' and the operations applicable to it.
Other programs can access the data object only through the defined operations.

Object-oriented example:

- object: 'building'
- operations: 'add a story', 'sell building', etc.

Why is object-orientation advantageous?

- reuse
- decoupling
- combining type and operations

Multi-sorted algebras

An algebra (e.g. complex numbers) is a mathematical structure, consisting of

- a collection of things (the sort of carrier)
- operations on these things
- axioms which explain the effects of the operations.

Example for a multi-sorted algebra: complex numbers

Constants 0, 1

Operations

_ + _: complex × complex ⇒ complex

_ * _: complex × complex ⇒ complex

etc.

Axioms

a + b = b + a

1 * a = a

a + 0 = a etc.

Another example: stack of items

Constant empty-stack

Operations

create: ⇒ stack

push: stack × item ⇒ stack

pop: stack ⇒ stack

top: stack ⇒ item

isEmpty: stack ⇒ Boolean

Axioms

isEmpty (create) = true

top (push (s, i)) = i

pop (push (s, i)) = s

pop (create) = error

Axiomatic definitions are difficult to specify and often not helpful for implementations.
Abstract data types are definitions of types together with operations.
The operations can then be implemented in a regular program.
An object-oriented language is based on the definitions of modules that describe ADTs.

Example: rational numbers in MOOSE

```
ADT rat -- rational numbers
  USE int
  TYPE RECORD num, denom: int

OPS make (i1,i2: int): rat ==
              result.num := i1
              result.denom := i2
    num (r: rat): int == r.num
    denom (r: rat): int == r.denom
    mult (r1, r2: rat): rat ==
       make (mult (num (a), num (b)),
              mult (denom (a), denom (b)))
```

An ADT can depend on other ADTs:

- Quotient algebras: limiting the ranges of values.
- Extensions: adding new operations.
- Derivations: explaining a new ADT in terms of previously defined ones (abstract implementation).

Example for abstract implementation:

The algebra for radians restricts the algebra of numbers to values between $0 \ldots 2\pi$.

```
radians.add (r1, r2) == int.add (r1, r2),
                if result ≥ 2π
                      then sub (result, 2π)
                elseif result< 0 then
add (result, 2π)
```

Another example for abstract implementation:

The four axioms of a distance function are:

- The distance from a point to itself is zero:

$$d(p, p) = 0$$

- The distance between two points is greater than zero if the two points are not identical:

$$d(p1, p2) > 0 \text{ if } p1 \neq p2$$

- The distance is symmetric:

$$d(p1, p2) = d(p2, p1)$$

- The sum of the lengths of two legs of a triangle is greater than or equal to the length of the third leg (triangle inequation):

$$d(p1, p3) \leq d(p1, p2) + d(p2, p3)$$

Exercise 2:

- Find some (> 2) implementations for a distance function.
- Show that your functions obey the four axioms of a distance function.

Solution for Exercise 2

Boolean Distance

$$d(p1, p2) == \text{ if p1 = p2 then 0 else 1}$$

axiom 1: $d(p1, p1) = 0$

axiom 2: $d(p1, p2) = 1$

axiom 3: $d(p1, p2) = d(p2, p1) = 1$

axiom 4: $d(p1, p2) + d(p2, p3) = 1 + 1 = 2$

Euclidian distance

$$d(p1, p2) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

axiom 1: $d(p1, p1) = \sqrt{(0-0)^2 + (0-0)^2}$

axiom 2: square root is defined as positive number.

axiom 3: apply commutative law.

axiom 4:

City block distance

$$d(p1, p2) = |a_x - b_x| + |a_y - b_y|$$

axiom 1: $d(p1, p1) = |0| + |0| = 0$

axiom 2: $|x|$ is always ≥ 0.

axiom 3: $|a_x - b_x| = |b_x - a_x|$

axiom 4:

Why is the following function not a distance function?

$$d(p1, p2) = ROUND(\sqrt{(a_x - b_x)^2 + (a_y - b_y)^2})$$

axiom 2: $d(p1, p2) \neg \geq 0$

e.g., p1 (0, 0.4), p2 (0, 0)

Generalization and Inheritance

If a class is a specialization (or subclass) of another class, then all objects of the subclass have all properties of the superclass.
This is the semantic of this abstraction mechanism: all dogs are animals.
Inheritance describes behavior, not implementation.

A programming example for inheritance:
Numbers have operations like $+$, - and axioms like $a + 0 = a$.

- A rational number is a number.
- Rational numbers inherit the operations of numbers.

The behavior of $+$ is valid for all numbers.
For example: commutative law: $a + b = b + a$
Thus the addition of rational numbers must obey the commutative law too.

The implementation for +, however, is different for rational numbers and for complex numbers.

- Addition of complex numbers $(a_r + b_i)$:
  $a + b == (a_r + b_r, a_i + b_i)$
- Addition of rational numbers $\frac{a_n}{a_d}$:
  $a + b == (a_n * b_d + b_n * a_d, a_d * b_d)$

Exercise 3:

Prove that the commutative law (a+b = b+a) holds true for both additions.
$a + b == (a_r + b_r, a_i + b_i)$ (complex numbers)
$a+b == (a_n*b_d+b_n*a_d, a_d*b_d)$ (rational numbers)

Solution for Exercise 3

Complex Numbers

$(a_r + b_r, a_i + b_i) = (b_r + a_r, b_i + a_i)$

apply commutative law:

$(a_r + b_r, a_i + b_i) = (a_r + b_r, a_i + b_i)$

Rational Numbers

$(a_n * b_d + b_n * a_d, a_d * b_d) = (b_n * a_d + a_n * b_d, b_d * a_d)$

apply comutative law:

$(a_n * b_d + b_n * a_d, a_d * b_d) = (a_n * b_d + b_n * a_d, b_d * a_d)$

Polymorphism

A program may contain variables a, b: number which can be, in turn, a complex or a rational number.

Every time a+b is executed, it must be determined which operation to use.

isRational (a) & isRational (b) → rat.add (a, b)

isComplex (a) & isComplex (b) → complex.add (a, b)

Example for Polymorphism

draw (object)

applies to all spatial objects. The implementation of the draw-operation differs for the type of object, e.g.,

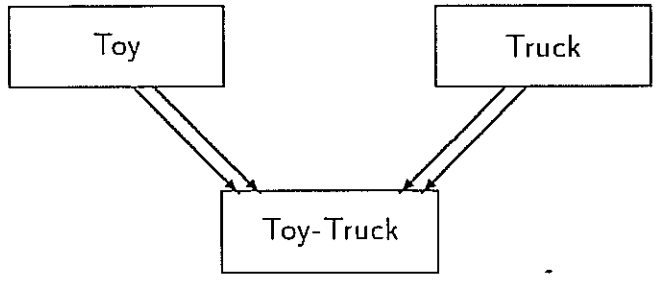
- isBuilding (object) → drawBuilding (object)
- isRailroad (object) → drawRailroad (object)

In Smalltalk this is called 'message passing', but it is semantically not different from a procedure call. Syntax: object message parameters.

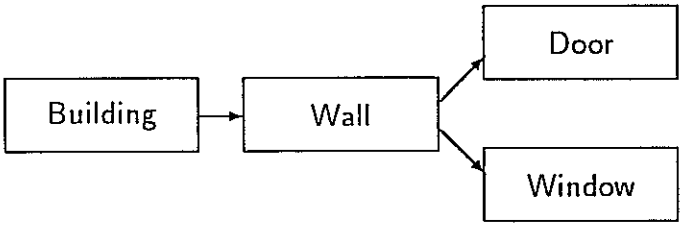
Multiple Inheritance

A class may be a subclass of several distinct superclasses.
For example, a toy-truck is_a truck and is_a toy.
The toy-truck inherits properties from both classes toy and truck.

Graphical representation of multiple inheritance:



Propagation derives values in aggregations.



Two types of propagation:

- bottom-up
- top-down

bottom-up propagation:

- from the smaller to the larger parts.
- using aggregate functions, e.g., SUM, AVE, COUNT, MIN, MAX.

For example, the area of a city is the SUM of all parcel areas which belong to the city.
formal: propagates (parcel, area, city, area, SUM)

top-down propagation:

- from the largest part to all smaller parts.
- immediate propagation of value.

For example, the currency of a state is the currency in the county, too.
formal: propagates (state, currency, county, currency).

Generic modules for parametric types

The behavior of a type is often not dependent on the types to which it is applied.
For example, a stack of integer behaves similar to a stack of real or a stack of complex.

A generic 'sorted list' can be written as follows:

```
TYPE    comparable
OP      before

TYPE    ring
AXIOM   a + b = b + a

TYPE    sortedList of comparable
        – before operation available
OP      insert (l, i) ==
        – needs 'before op' to decide where to insert

TYPE    ratNumber is a comparable
        is a ring
OP      before ==
    – implemented here

TYPE    sortedList of ratNumber
    – permitted because ratNumber is comparable
```

A complete object-oriented language must contain

- multiple inheritance
- generic modules

Object-oriented programming languages:

- SIMULA: the 'father' of object-oriented languages:
  - single inheritance
  - concurrency
- Smalltalk-80: the object-oriented 'prototype':
  - uses message-passing to explain polymorphism
  - single inheritance
  - inheritance of implementation
  - dynamic binding (execution penalty)
- A number of LISP dialects (OOPS, Scheme, etc.).

C precompilers:

- Objective C
  - single inheritance
- C++
  - single inheritance (multiple planned)
- Eiffel
  - multiple inheritance
  - generic modules

Summary Session 2:

- The three abstraction mechanisms *classification*, *generalization*, and *aggregation* are an essential part of object-oriented modeling.
- Object-oriented software systems must support object-oriented abstractions.
- Multi-sorted algebras are helpful, because they separate specification and implementation.
- Only a few current programming languages support object-orientation sufficiently.

Session 3: Object-oriented DBMS

- DBMS functionalities
- Conceptual Schema Design
- Data Definition Language
- Complex Objects and Value Types
- Object-oriented programmer's interfaces
- Complex consistency constraints

DBMS functions

- Storage and retrieval of data.
- Multiple concurrent users.
- Protection of data against loss and misuse.
- Maintenance of consistent data collections.

Object-oriented languages have been extended with the concept of 'persistent objects':
Data values that are stored and available in later runs of a program.

In this simple form, 'persistent objects' cannot replace a DBMS.
Problems with persistent objects:

- Single user.
- Data security and protection.
- Effective buffer management and storage clustering.

Object-oriented DBMS are needed.

Problems with merging DBMS with the object-oriented design:

- DBMS defines data types with implied DB operations. All other operations are defined in programs.
- Object-orientation defines a data type and all operations in a single module. No other program can access the data except through defined operations.

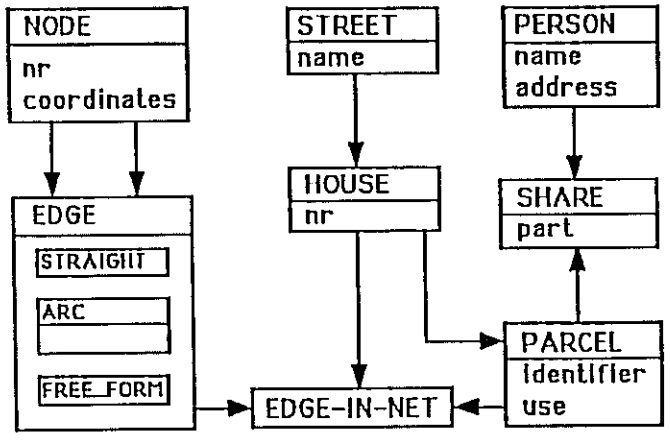
Conceptual schema

In an object-oriented design multiple views of the data are necessary:

- Conceptual schema for overview.
- Detailed module definitions for specifics.

Future CASE systems will help to allow for multiple views and varying resolution based on the same code.

Graphical Tools (like Entity-Relationship diagrams) are very helpful for the schema design, because they show the model in a very concise form.



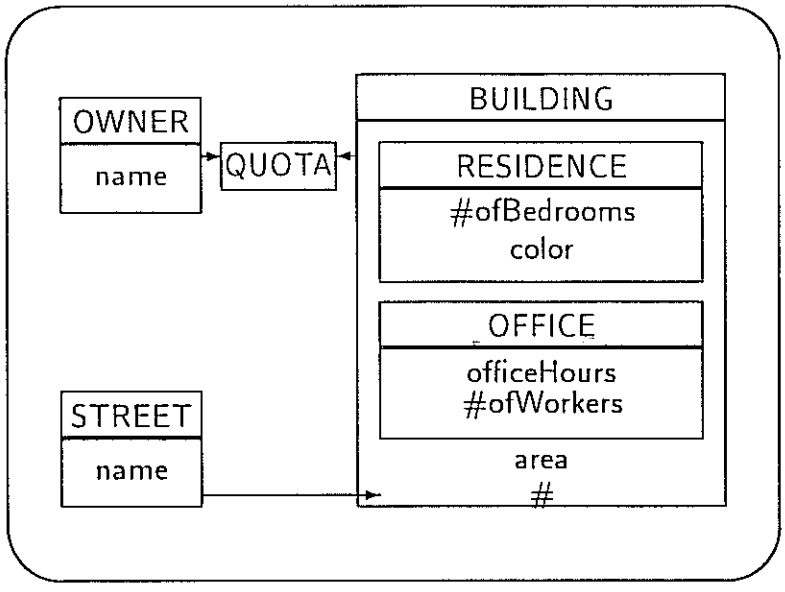
Simple Conceptual Database Schema

Exercise 4:

Draw a graphical schema modeling the following situation:

- Residences have the properties number of bedrooms, color, and area.
- Offices have the properties area, number of workers, and office hours.
- A residence has one or more owners, and one person can own more than one house.
- Residences and offices are located on streets, and have addresses (e.g., 30 Grove Street). Each building belongs to exactly one street, and its address is unique.

Solution for Exercise 4



The conceptual schema must be translated into a form which is understood by the database (Data Definition Language). An object-oriented DDL describes:

- names of object types (classification),
- relations among object types (aggregation, generalization),
- implementation of object types (properties).

A description of a DDL:

```
BUILDING (buildingId, streetId, number)
STREET (streetId, name)
PERSON (socSec, name, firstName, birthDate)
TENANT (secSec, buildingNo)
OWNER (socSec, building, ownershipQuota)
```

Format in an extended relational form:

```
relation BUILDING
  key buildingId: positive integer;
  field theStreet: street; {the reference
     will be replaced by the key of street,
     implies an existing constraint}
  field number: integer;
```

In CODASYL style:

```
01 building.
   05 buildingId pic 99999.
   05 streetNo   pic 999.
set building-street.
    owner is street.
    member is building optional automatic.
    sorted by streetNo of building.
```

Object-oriented database management systems must be extensible:

- Objects can be arbitrarily complex.
- A predefined set of data types is insufficient for complex objects.
- It is necessary that the user can define her/his own ADTs using other ADTs.

An object-oriented DBMS sees data as typed, uninterpreted bit strings.
Access operations are needed.
The modules defining the objects must provide specific operations for the database to access parts of the data necessary for the DBMS.

Required object operations:

- Compare two objects for equality.
- Compare two objects for order (less than operation).
- Access a spatial object by its minimal bounding rectangle.
- Compute a hash value.

Example

In order to keep all buildings along a street in a sorted list, the operations 'buildingBefore' and 'buildingEqual' must be defined.

We observed that many object types have similar components. We call ADTs, which are the building blocks for object types, *value types*.
Fundamental value types:

- integer with intEqual, intBefore, intHash
- string with strEqual, strBefore, strHash
- etc.

Example for a user-defined value type:

```
ADT iv1 -- one-dimensional interval
  USE int
  TYPE RECORD low, high: int
```

Such value types can be used for further definitions of more complex value types:

```
ADT iv2 -- two-dimensional interval
  USE iv1 -- from two 1-d intervals
  TYPE RECORD low, high: iv1
```

The decomposition of objectTypes into value types is a technical issue. It releases the database administrator from considerable overhead when defining the necessary object operations.

Example

The component 'buildingNr' is of valueType 'integer'.

The object operation 'buildingBefore' can be implemented in terms of the value type:

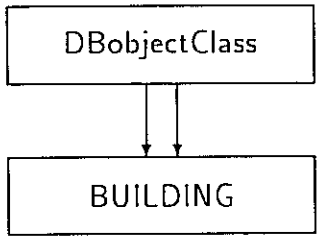
```
buildingBefore (b1, b2) ==
    intBefore (b1.building#, b2.building#)
```

The object-oriented programmer's interface is a collection of general 'object' operations.

Three categories:

- property operations
- unary object operations
- binary object operations

All object operations are defined for a generalized 'DBobjectClass'.
The DBobjectClass inherits all DB operations to each subclass.



Similarly, other concepts, like spatial properties and graphics, can be inherited from a generalized object type (multiple inheritance).

The DB object operations are:

- put a property into an object
- get a property from an object
- store object
- modify object
- delete object
- access object by key
- aggregate two objects
- dissolve two aggregated object
- get next component

Level 1: get and put operations:

For each property there is a 'get' and a 'put' operation to assign and access values of objects, respectively.

- putProperty (object, property)
- getProperty (object): property

For instance, the class 'person' with the property 'name' has the following operations:

- putPersonName (personObject, 'Jack')
- name := getPersonName (personObject)

Level 2: Unary object operations:

For each object class, the three fundamental DB operations *store*, *modify*, and *delete* are implemented.

- store (object, classType)
- modify (object)
- delete (object)

For instance, the class 'person' in an 'objectType' and inherits the operations

- storePerson (personObject)
- modifyPerson (personObject)
- deletePerson (personObject)

Direct Access

An instance of the object class can be accessed with the knowledge of a key value.
For example, access to a 'person' through the property 'name':

- getPersonByName (personObject)

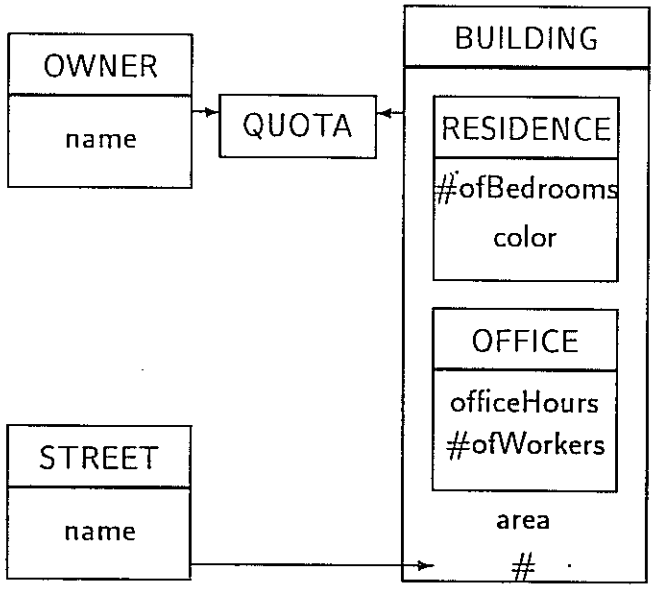
Level 3: Binary object operations for aggregatedclasses

For instance, the operations for aggregate 'streetNo' between 'street' and 'building' are:

- streetBuildingAggregate (streetObject, buildingObject)
- streetBuildingDissolve (streetObject, buildingObject)
- streetGetNextBuilding (buildingObject)

Exercise 5:

Write the set of object operations for this schema:



Solution for Exercise 5

Level 1

owner1	ownerPutName	ownerGetName
street1	streetPutName	streetGetName
building1	buildingPut# buildingPutArea	buildingGet# buildingGetArea
office1	officePutOfficeHours officePut#ofWorkers officePutArea officePut#	officeGetOfficeHours officeGet#ofWorkers officeGetArea officeGet#
residence1	residencePut#ofBedrooms residencePutColor residencePutArea residencePut#	residenceGet#ofBedrooms residenceGetColor residenceGetArea residenceGet#

Level 2

owner2	ownerStore ownerModify ownerDelete
street2	streetStore streetModify streetDelete
office2	officeStore officeModify officeDelete
residence2	residenceStore residenceModify residenceDelete
building2	buildingStore buildingModify buildingDelete
quota2	quotaStore quotaModify quotaDelete

Level 3

owner3	ownerQuotaAggregate ownerQuotaDissolve ownerGetNextQuota
building3	buildingQuotaAggregate buildingQuotaDissolve buildingGetNextQuota
street3	streetBuildingAggregate streetBuildingDissolve streetGetNextBuilding
building3	buildingGetStreet
quota3	quotaGetOwner quotaGetBuilding

Consistency constraints:

- with respect to the object
- among objects

Coded in the module in the regular programming language. Provides operations on complex objects.

Consistency constraints with respect to a single object:

- No two objects may exist with the same value. Applies for storing and modifying an object.
- Values derived from others (functional dependent)

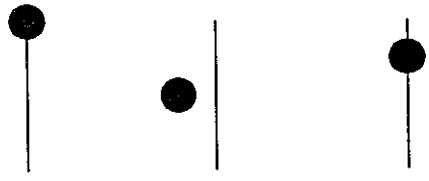
Consistency constraints with respect to related objects:

- An object cannot exist without another object. Applies for storing, modifying, and deleting (existence constraint).
- An object cannot exist if another object exists.
- Derived values.

Example for complex consistency constraints

Situation: An edge connects always exactly two nodes. If a node, which is not start or end node, lies on an edge, then the edge must be split into two parts.

Operation addNode:



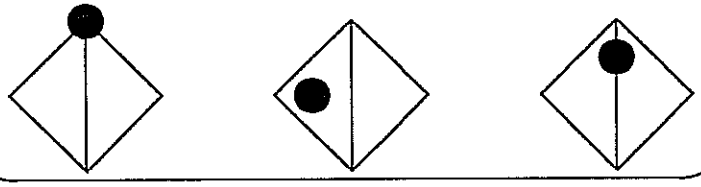
- Node is the start or end of an edge.
- Node does not lie on an edge.
- Node lies on an edge.

Example for very complex consistency constraints

Situation: A completely triangulated area.
Constraints:

- Only triangular areas are permitted.
- No two triangles must overlap.
- An edge connects exactly two nodes.

Operation addNode:



- Node is identical to an exisiting node.
- Node lies on a edge (but not start or end).
- Node lies within a triangle.

Consistency constraints for removing objects are even more complex.
Simple consistency constraints must be often violated during long transactions in order to make to required changes.
Include the consistency to the object operations.

First object layer: put operations

- include checking of valid ranges.

e.g. coordinates must be in the range between 0 and 10,000:

```
putXCoordinate (object, xValue) ==
-- check whether xValue lies within
-- allowed range.
```

[Christensen 1987] A. Christensen and T.U. Zahle. A Comparison of Self-Contained and Embedded Database Languages. In: P. Stocker and W. Kent, editors, Proceedings 13th VLDB Conference, Brighton, England, September 1987.

[CODASYL 1971] Data Base Task Group Report. CODASYL. Technical Report, New York, NY, April 1971.

[Codd 1970] E.F. Codd. A Relational Model for Large Shared Data Banks. Communications of the ACM, 13(6), June 1970.

[Dahl 1968] O.-J. Dahl et al. SIMULA 67 Common Base Language. Technical Report N. S-22, Norwegian Computing Center, Oslo, Norway, October 1968.

[Dittrich 1986] K. Dittrich. Object-Oriented Systems: The Notation and The Issues. In: K. Dittrich and U. Dayal, editors, International Workshop in Object-Oriented Database Systems, Pacific Grove, CA, 1986.

[Edelson 1987] D. Edelson. How Objective Mechanisms Facilitate the Development of Large Software Systems in Three Programming Languages. SIGPLAN Notices, 22(9), September 1987.

[Egenhofer 1987] M. Egenhofer. EMS—Easy Monitor System, User's Manual. 1987. Internal Documentation, University of Maine, Orono, Department of Surveying Engineering, Orono, ME.

[Egenhofer 1988a] M. Egenhofer and A. Frank. MOOSE: Combining Software ENgineering and Database Manageмenst Systems. In: Second International Workshop on Computer-Aided Software Engineering, Advance Papers, Cambridge, MA, May 1988.

[Egenhofer 1988b] M. Egenhofer and A. Frank. A Precompiler For Modular, Transportable Pascal. SIGPLAN Notices, 23(3), March 1988.

[Frank 1982] A. Frank. PANDA—A Pascal Network Database System. In: G.W. Gorsline, editor, Proceedings of the Fifth Symposium on Small Systems, Colorado Springs, CO, 1982.

[Frank 1983a] A. Frank. Data Structures for Land Information Systems—Semantical, Topological, and Spatial Relations in Data of Geo-Sciences, (in German). PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1983.

[Frank 1983b] A. Frank. Problems of Realizing LIS: Storage Methods for Space Related Data: The Field Tree. Technical Report 71, Institut for Geodesy and Photogrammetry, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 1983.

[Frank 1984] A. Frank. Requirements for Database Systems Suitable to Manage Large Spatial Databases. In: International Symposium on Spatial Data Handling, Zurich, Switzerland, August 1984.

[Frank 1986] A. Frank and W. Kuhn. Cell Graph: A Provable Correct Method for the Storage of Geometry. In: D. Marble, editor, Second International Symposium on Spatial Data Handling, Seattle, WA, 1986.

[Goldberg 1983] A. Goldberg and D. Robson. Smalltalk-80. Addison-Wesley Publishing Company, 1983.

[Guttag 1977] J. Guttag. Abstract Data Types And The Development Of Data Structures. Communications of the ACM, June 1977.

[Härder 1985] T. Härder and A. Reuter. Architecture of Database Systems for Non-Standard Applications, (in German). In: A. Blaser and P. Pistor, editors, Database Systems in Office, Engineering, and Scientific Environment, Springer Verlag, New York, NY, March 1985. Lecture Notes in Computer Science, Vol. 94.

[Jacky 1987] J.P. Jacky and I.J. Kalet. An Object-Oriented Programming Discipline For Standard Pascal. Communications of the ACM, 30(9), September 1987.

[Johnson 1983] H.R. Johnson et al. A DBMS Facility for Handling Structured Engineering Entities. In: ACM Engineering Design Applications, 1983.

[Liskov 1981] B. Liskov et al. CLU Reference. Lecture Notes in Computer Science, Springer Verlag, New York, NY, 1981.

[Lorie 1983] R. Lorie and W. Plouffe. Complex Objects and Their Use in Design Transactions. In: ACM Engineering Design Applications, 1983.

[Maier 1986] D. Maier. Why Object-Oriented Databases Can Succeed Where Others Have Failed. In: K. Dittrich and U. Dayal, editors, International Workshop in Object-Oriented Database Systems, Pacific Grove, CA, 1986.

[Olthoff 1985] W. Olthoff. An Overview on ModPascal. SIGPLAN Notices, 20(10), October 1985.

[Parnas 1978] D.L. Parnas and J.E. Share. Language Facilities for Supporting the Use of Data Abstraction in the Development of Software Systems. Technical Report, Naval Research Laboratory, Washington, DC, 1978.

[Rowe 1987] L.A. Rowe and M.R. Stonebaker. The POSTGRES Data Model. In: P. Stocker and W. Kent, editors, Proceedings 13th VLDB Conference, Brighton, England, September 1987.

[Sernades 1987] A. Sernades et al. Object-Oriented Specification of Databases: An Algebraic Approach. In: P. Stocker and W. Kent, editors, Proceedings 13th VLDB Conference, Brighton, England, September 1987.

[Storm 1986] R. Storm. A Comparison of the Objet-Oriented and Process Paradigms. SIGPLAN Notices, 21(10), October 1986.

[Stroustrup 1986] B. Stroustrup. The C++ Programming Language. Addison-Wesley Publishing Company, 1986.

[Wilkins 1984] M.W. Wilkins and G. Wiederhold. Relational and Entity-Relationship Model Databases and VLSI Design. IEEE Database Engineering, 7(2), June 1984.

[Woelk 1987] D. Woelk and W. Kim. Multimedia Information Management in an Object-Oriented Database System. In: P. Stocker and W. Kent, editors, 13th VLDB conference, Brighton, England, 1987.

[Zilles 1984] S.N. Zilles. Types, Algebras, and Modelling. In: M.L. Brodie et al., editors, On Conceptual Modelling, Springer Verlag, New York, NY, 1984.

Object-Oriented Databases: Database Requirements for GIS*

Max J. Egenhofer
Andrew U. Frank
Surveying Engineering Program
University of Maine
Orono, ME 04469, USA
MAX@MECAN1.bitnet
FRANK@MECAN1.bitnet

Abstract

Conventional database management systems have proven to be insufficient to model and perform non-standard applications such as spatial information systems, CAD/CAM, etc. The object-oriented approach overcomes some crucial deficiencies with refined methods for data abstraction and suitable tools to structure data allowing the usage of complex object types. For the implementation of an object-oriented database management system appropriate tools in programming languages are needed. A Geographic Information System (GIS) which treats spatial data asks for additional object-oriented extensions such as storage clusters to provide fast access on spatial objects, concurrent management of spatial and non-spatial data, and treatment of properties of spatial objects in query languages.

1 Introduction

Geographic information systems (GIS) contain substantial amounts of data which must be stored in computer readable and accessible form. Computer scientists have studied database management systems (DBMS) for several years and database management systems for commercial usage are currently available; however, several studies have shown that these systems are not suitable for non-standard applications, such as GIS, Land Information Systems (LIS) [Frank 1981] [Frank 1984b], and CAD/CAM [Eastman 1980] [Stonebraker 1982]. Several experimental database management systems have been tried, but a complete system is not yet available commercially and additional research is needed.

*Work on these concepts was partially funded by a grant from NSF under No. IST-8609123

During the last years, research in software engineering has promoted an object-oriented design method by which real world objects and their relevant operations are modelled in a program. This approach is most useful for application areas like GIS, because it naturally supports treatment of complex, in this case geometric, objects [Kjerne 1986]. Unlike conventional data models, an object-oriented design is more flexible to describe the complex data structures, primarily by refined techniques for conceptual modeling, such as generalization and inheritance.

Spatial information systems will benefit from the use of object-oriented databases in various forms:

- The system architecture will become clearer and easier to maintain so that software systems will have a longer life cycle.
- Programmers need not worry about aspects of the physical location where to store data; instead, a unified set of commands provides the functionality to store and retrieve data.
- By using a database, data is treated by its properties; by using an object-oriented database, these properties are logically combined to objects which can be often very complex.

The paper starts with a review on existing database systems and their deficiencies in serving as a suitable tool for modeling space related applications. In particular, the theory for data structuring techniques is compared to the tools in programming languages available to implement them. Following this chapter, an object-oriented data model is introduced which is built upon the three major abstraction methods of classification, generalization, and aggregation. The important concepts of hierarchical and multiple inheritance are explained. Software engineering aspects are investigated in chapter 4. Tools, such as programming languages and programming environments, are seen as especially crucial to implement object-oriented designs. In chapter 5, properties and requirements of object-oriented database management systems suitable for GIS are discussed. The implications of object-oriented structures to query languages are sketched. The paper closes with a summarization of the GIS requirements for a database management system.

2 Databases for Non-Standard Applications

Databases have become an accepted tool for storing data in readable and accessible form, and systems for CAD/CAM or spatial information (Geographic or Land Information Systems) integrate a database system into a larger software system [Frank 1983a].

Unlike conventional systems, spatial information systems deal with data describing the real world, and they combine several domains which have not been studied sufficiently. Think of a Geographic Information System in which a large variety of diverse tasks, such as

- sophisticated treatment of real-world geometry
- measurements of different resolution and accuracy
- uncertain measurements and attribute classifications
- legal aspects
- management of time series of veriable attributes
- representation of data in different generalization levels

are combined in a single system. This combination of difficult areas may be a reason why spatial information systems have been so slow to come and only similar, but more restricted applications are used successfully [Tripp 1987] [Beckstedt 1987].

Spatial information systems can benefit from using database management systems because these provide a unique form of storing and accessing structured data. The problems of low-level data management are removed from the programmer's and end user's responsibility, and data can be descibed by their properties, not their physical structure.

Unfortunately, existing commercial database systems are not sufficient for applications to engineering tasks, often called non-standard applications, such as CAD/CAM for VLSI design or cartographic and geographic information systems. Each of these areas struggle with the same kind of problem: they contain substantial amounts of 'real world' data including geometric aspects which must be managed by a computer. Their composition is too complex to be managed in conventional database systems efficiently. Interactive systems require a certain degree of performance, independent of the size of the data sets—a demand which cannot be fulfilled by existing systems.

2.1 Deficiencies of Conventional Database Systems

Members of the CAD/CAM-community have complained that there are currently no database systems which are appropriate for their demands [Buchmann 1985] [Sidle 1980] [Udagawa 1984]. They state that standard database systems do not fulfill requirements which are crucial for engineering applications:

- Performance is unacceptable when a database is populated with large amounts of data [Wilkins 1984] [Härder 1985] [Maier 1986]. Dealing with spatial data, this deficiency is particularly visible during interactive graphic sessions.
- The treatment of complex objects [Lorie 1983], such as molecules in chemistry [Batory 1984], spatial objects in GIS/LIS [Frank 1984b], or circuits in VLSI is not supported. Spatial objects, for example, should not be forced to be artificially decomposed in smaller parts.

- Appropriate mechanisms for data structuring [Johnson 1983], especially for structuring real-world data, are missing.

Most conventional database management systems are built on the relational model [Codd 1982]. In the relational model, data are organized in tables or relations. Columns of the tables are called attributes and all values in an attribute are elements of a common domain; rows are records, tuples, or relation elements.

While this concept is suitable for modeling commercial data, such as bank accounting, it is too simplistic for modeling data describing the real world. Geographic data cannot be modelled as strings, reals, and integers in table format—the necessary data types are more complex. One property of complex object types is that an object type can be part of another (more) complex object type. Data types for two-dimensional coordinates, for example, consist of an x- and y-value. A more complex object type includes some error value stating the accuracy of the coordinates. Such 'error-affected' coordinate types may be part of a point type which combines the coordinates with a number and a theme.

Moreover, the relational model lacks the powerful concept of recursion which is crucial for modeling complex situations such as spatial data and their subdivisions: areas can be decomposed into several sub-parts which themselves can be continuously decomposed further. For example, the area of a town is composed of several other areas, such as the house lots, streets, etc.

Only recently, the design of databases for spatial information system has become a research topic [Lipeck 1986] [Schek 1986] and only a few experimental database systems exist which pursue new concepts [Frank 1982b] [Dayal 1986] [Batory 1986].

In existing GIS software systems, a trend towards sophisticated software engineering methods [Aronson 1983] and architecture [Smith 1986] can be observed, stressing object-oriented concepts for geometric data handling [Herring 1987]; however, database management systems, and especially object-oriented ones, have not yet been incorporated into commercial GIS systems.

What are the technical reasons that database systems have failed to support complex non-standard situations? First, database requirements are not a hardware problem. It is commonly known that hardware currently develops much faster than software; however, faster and less expensive CPUs, larger hard disks, and more memory do not overcome certain problems in database technology for engineering. Solutions which can be achieved by exploiting additional and faster hardware are not the problems which will be addressed in this paper. For example, new technologies for hard disks provide more storage capacity at less expensive costs, but the disk access has not become faster. This is important for database management systems which struggle with growing data collections which get too complex to be managed efficiently.

Rather than hardware, software engineering is an impediment for better-suited engineering databases. We claim that theoretical and conceptual problems are the subject of major

improvements. For example, even with much faster access to storage devices, internal data structures organizing spatial data will be needed in order to provide adequate performance for range queries.

2.2 Deficiencies of Software Engineering Tools

Programming is still seen as an art in which the programmer can accomplish a goal by whatever means he believes to be suitable. The contrary is true: each program is a very formal piece of code following strict rules; programs must not only be compatible between hardware, but also 'compatible' among programmers, i.e., different programmers must be able to read and understand each others' code. By following strict rules and restrictions, these goals can be achieved. Programming languages tend towards offering a large variety of tools, while the opposite is needed, namely restrictions, such that several programmers come up with very similar solutions.

The other deficiency observed is that concepts in software engineering do not match with database models or are not suitable for them. Theory in software engineering provided some suitable techniques, such as the concept of abstract data types [Guttag 1977] [Parnas 1978] [Zilles 1984], in which each module encapsulates an object type with all its pertinent operations. Standard database systems, based on networks [CODASYL 1971] or the relational [Codd 1970] data model do not fit easily into this concept.

2.3 Deficiencies of Implementing Data Structures

A number of methods to capture more semantics in the data model have been proposed in the literature (for an overview and critique see [Brodie 1984a] which contains an extensive list of references), but most of these methods have not been implemented, and none of them is readily available. Complex tasks require complex structuring tools. 'Complex', however, need not mean 'complicated': the support of data structuring must be powerful without sacrificing the ease of use, and it must be extensive without becoming excessive.

In the past, considerable efforts were made to enrich existing data models with facilities to treat complex objects. 'QUEL as a Datatype' [Stonebraker 1983b] and ADT-INGRES [Stonebraker 1983a] extend the relational model with features to define more complex types; DAPLEX [Shipman 1981] is a functional language which includes hierarchical relationships and transitive closure; the NF^2 model [Schek 1985] supports composite attribute types being tuples or relations performing a hybrid of relational and hierarchical data model. Recently, it was investigated, how geometry could be modelled by using these extended data models [Kemper 1987], and it was shown that only partial remedies are provided with these extensions.

3 An Object-Oriented Data Model

This chapter introduces the notation of objects and the abstraction tools. An entity of whatever complexity and structure can be represented by exactly one object in the database [Dittrich 1986], meaning that no artificial decomposition into simpler parts should be necessary due to some technical restrictions. Complex data types for large objects, such as an entire city (with its details about streets, houses, and their details such as owners, neighbors, etc.), do not overcome the problem of data structuring.

The object-oriented data model is built on the three basic concepts of abstraction [Brodie 1984b]: classification, generalization, and aggregation. Furthermore, inheritance describes how the properties of a class are derived from properties of related classes.

3.1 Classification

Classification can be expressed as the mapping of several objects (instances) to a common class. The word *object* is used for a single occurrence (instantiation) of data describing something that has some individuality and some observable behavior. The terms *object type*, *sort*, *type*, *abstract data type*, or *module* refer to types of objects, depending on the context. In the object-oriented approach, every object is an instance of a class. A type characterizes the behaviour of its instances by describing the operators that can manipulate those objects [O'Brien 1986]. These operations are the only means to manipulate objects. All objects that belong to the same class are described by the same properties and have the same operations.

For example, the model for a *town* may include the classes *residence*, *commercial building*, *street*, *park*, etc. A single instance, such as the house with the address '30 Grove Street', is an object of the corresponding object type, i.e. the particular object is an instance of the class *residence*. Operations and properties are assigned to object types, so for instance the class *residence* may have the property *number of bedrooms* which is specific for all residences. Similarly, the class *street* may have an operation to determine all adjacent parks.

In the implementation, classes translate to abstract data types or modules.

3.2 Generalization

Generalization is a well-known term in cartography for varying representations of objects according to the level of detail needed in a given scale. Similarly, generalization as an abstraction mechanism provides views in different levels of detail.

Several classes of objects which have some operations in common are grouped together into a more general *superclass* [Dahl 1968] [Goldberg 1983]. The converse relation of superclass, the *subclass*, describes a specialization of the superclass. *Ancestor* and *descendant* are often used as equivalent terms for superclass and subclass. Subclass and superclass are

related by an *is_a* relation. For example, the object type *residence* is a *building*; *residence* is a subclass of *building*, while *building* is its superclass.

3.2.1 Inheritance

The properties and methods of the subclasses depend upon the structure and properties of the superclass(es). Inheritance defines a class in terms of one or more other classes. Properties which are common for superclass and subclasses are defined only once (with the superclass), and inherited by all objects of the subclass, but subclasses can have additional, specific properties and operations which are not shared from the superclass. Inheritance is transitive propagating the properties from one superclass to all related subclasses, to their subclasses, etc. This concept is very powerful, because it reduces information redundancy [Woelk 1987]. For example, *building* shares properties and operations with *residence*, such as having an *area* and being *neighbor* of some other building. The class *residence* inherits all operations from its superclass *building* without the need to redefine them explicitly. Additional properties, such as the *number of bedrooms*, are specific for the *residence*. Specializations of *residences*, such as *rural residences* and *city residences* inherit the specific properties of the *residences*, i.e. *number of bedrooms*, and by transitivity the properties of the super-superclass *building*, i.e., *area* and *neighbor*.

Operations of the superclass are compatible between objects of the superclass and subclass. Every operation on an object of a superclass can be carried out on the subclass as well; however, operations specifically defined for the subclass are not compatible with superclass objects. For example, *neighbor* is an operation of the superclass *building*, and it is thus compatible with objects of the type *residence*. On the other hand, the operation *number of bedrooms* is specific fo the subtype *residence* and thus not applicable for objects of the superclass *building*.

Hierarchies: Inheritance can be strictly hierarchical as in the example of *building*, *residences*, etc (Figure 1). Hierarchical inheritance implies that each subclass belongs only to a single group of hierarchies; one class cannot be part of several distinct hierarchies, i.e., in hierarchical generalization, a class can have only one direct superclass.

Multiple Inheritance: The structure of a strict hierarchy is an idealized model and fails most often when applied to real world data. Most 'hierarchies' have a few non-hierarchical exceptions in which one subclass has more than a single, direct superclass. Thus, pure hierarchies are not always the adequate structure for inheritance; instead, class lattices [Woelk 1987] with acyclic directed graphs are more suitable. This concept, allowing a multitude of distinct superclasses for a single class, is called multiple inheritance [Nguyen 1986].

For example, *highways* and *channels* are *artificial transportation ways*, while *rivers* are *natural transportation systems*; however, *channels* and *rivers* both belong to another

hierarchy, the *water system* as well. So, *channel* and *river* participate in both *water systems* and *transportation ways*, two distinct hierarchies which cannot be compared with each other (Figure 2).

3.3 Aggregation

Several objects can be combined to form a semantically higher-level object where each part has its own functionality. This is different from the way generalization is defined: operations of aggregates are not compatible with operations on parts.

Aggregation establishes a relation which is often called a 'part-of'-relation since aggregated classes are 'parts of' the aggregate. For example, the class *county* is an aggregate of all related *settlements*, *forests*, *lakes*, *streets*, etc.

3.3.1 Dependencies Among Values of Different Object Types

Complex objects often do not own independent data, but properties which rely upon values of other objects. In GIS and LIS, for example, a large amount of attribute values is propagated from one level of abstraction to another. When combining local and regional data, this concept must be used to pursue the dependencies among data of different levels of resolution [Egenhofer 1986]. The population of a county, for example, is the sum of the population of all related settlements; therefore, the property *population* of the aggregate *county* is derived by adding all values of the property *population* owned by the class *settlement*.

While inheritance is the propagation of operations, functional dependencies describe how *values* of one class are derived from values of another class. Often, a value is directly propagated from the property of one class to another property of a different class. In the object-oriented model, objects may have properties with values which rely on values of other objects. Dependent values must be derived.

If more than a single value contributes to the derived value, the combination of the values must be described by a function. Common operations are minimum, maximum, sum, average, and weighted average.

3.4 Tools for Modeling

Even for relatively small models of a mini-world, the structure can become too confusing to be presented with pure alphanumeric means. Graphical methods have proven to be suitable tools for a better and clearer understanding of data structures; representations such as entity-relationship diagrams [Chen 1976] are essential tools in an object-oriented design. By using the power of graphical representation, modeling becomes more clear and understandable compared to pure alphanumerical descriptions.

The concept of multiple inheritance must be incorporated into the entity-relation- ship model by using some simple graphical means.

4 Software Engineering Aspects

A database is only one part in the large effort to produce a CAD/CAM system, a land information system, or any of the other non-standard applications. It is thus necessary that database methods and techniques fit well within a software engineering environment. Theoretically founded methods have proven to be well-suited for modularization based on abstract data types.

4.1 Abstract Data Types

The software engineering method for abstraction and data structuring is based on the work on formal specifications using abstract data types. An abstract data type (ADT) or a multi-sorted algebra is a mathematical structure which fully defines the behaviour of objects, i.e., the semantics of the object-type and its operations. Abstract data types are specified as an algebra describing what sorts of objects (types) are dealt with and what kinds of operations they are subject to. A set of axioms determines the effects of the operations. It is up to the designer to assure that the sorts and their operations form a reasonable and meaningful object.

Abstract data types can be combined in layers, where higher-level abstract data types are first described independently (specifications) and it is then shown how this behaviour can be achieved using other, hierarchically lower abstract data types (called an 'abstract implementation' [Olthoff 1985] [Frank 1986]). Such abstract implementations can be formally checked for correctness, by proving that the axiomatically specified behaviour of the upper abstract data type follows from the abstract implementation and the axiomatically specified behaviour of the lower abstract data type.

Abstract data types are very useful in the design of large systems. They allow a much more comprehensive and complete way of specifying routines together with their axioms than an extensive programming language does.

4.2 Object-Oriented Programming

The programming language must support the abstraction methods. The development of object-oriented databases has had problems, mostly because suitable software tools are missing or old and inappropriate tools have been used. Certain programming tools must be available to implement object-oriented databases and data structures.

- In order to implement complex objects, language tools, such as RECORD structures in Pascal [Jensen 1978], Ada [Ada 1983], etc., are needed to create user-defined object

types. For example, in a spatial information system a large amount of the data are points with coordinates, either 2-dimensional or 3-dimensional. By combining the coordinates to a RECORD, a complex data type, such as a pointType, can be created.

```
TYPE pointType = RECORD
                    x, y, z: coordType;
                 END;
```

- In object-oriented programming, objects consist of a type definition and a collection of operations. Objects can be only accessed or manipulated by using these operations. For example, the object type pointType, as introduced above, has a specific operation to shift a point in x-, y-, or z-direction (translation); the user can access a point only via the interface of the pointType.

  Object-oriented programming needs (1) tools to tie types and their operations closely together (modularisation), and (2) methods to hide internal parts of the routines from unauthorized external usage (encapsulation). Modula-2 [Wirth 1982], Smalltalk-80 [Goldberg 1983], and C++ [Stroustrup 1986] are examples for languages which support modularization and encapsulation. Encapsulation provides implementation-independent code outside of a module.

- Genericity [Cardelli 1985] has proven to be a suitable and powerful method to reduce redundant definitions of ADTs. A generic type or object is a definition which is a backbone for a series of detailed and specified definitions. A *pair*, for instance, is a generic object type which combines two objects; *pair* can be applied to two integers forming a *pair of integers*. Operations which are common for all *pairs*, such as *equality*, are defined only once, while their implementation might vary from type to type.

  Programming code is considerably reduced by using generic types, because operations which apply to all instances are coded only once.

- Only recently, inheritance has been recognized as a crucial issue in object-oriented approaches. Two categories of inheritance are identified, hierarchical (or straight) and multiple inheritance.

  Hierarchical inheritance deals with a strict structure in which for each child only a single direct ancestor exists. Some programming languages, such as Smalltalk-80 and C++, support straight inheritance, other languages with variant RECORD structures,

such as Pascal or Ada, can at least simulate hierarchical inheritance of types, but not of operations. For example, two-dimensional and three-dimensional coordinates are both representations for points. From this superclass, they both inherit operations such as calculating the distance between two points. The implementation of these operations may be different, however, from outside they look the same. With a variant RECORD structure, the inheritance can be implemented as follows:

```
TYPE pointType = RECORD
                   CASE dimension: dimensionType of
                      twoD: (x, y: coordType);
                      threeD: (x, y, z: coordType);
                 END;
```

Multiple inheritance allows one object to be part of several distinct hierarchies. Only a few languages, such as some LISP dialects (CommonObjects [Snyder 1986], Zeta Lisp [Weinreb 1981]), ObjectLOGO [Davidson 1987], and Trellis/Owl [O'Brien 1986] allow the definition of multiple inheritance.

These concepts for object-oriented programming do not include a message-passing paradigm [Goldberg 1983]; message-passing is based on passive objects which can receive messages directing what actions can be executed on an object. Message-passing is often cited as necessary for an object-oriented design; in our opinion it is not an essential feature, but certainly a desirable and helpful paradigm. It was outlined that message-passing is rather a pedagogical than a semantical difference to conventional routine calls, and any procedure call in an Algol-like language could be seen as message-passing [Storm 1986].

Object-oriented programming of object-oriented applications depends on the choice of the programming language; only a few languages support object-orientation sufficiently. We claim that object-oriented applications cannot be achieved without object-oriented tools and concepts.

4.3 Support System

Database systems are large software systems which must be embedded in some environment supporting development and maintainance. Especially the latter is crucial for the life cycle of a system. Such an environment must be tailored to the object-oriented approach by automatically propagating object definitions to other objects, controlling which object uses which other objects, and which objects are used elsewhere.

In software engineering, abstract data types may be represented by modules which encapsulate a type and its operations. Facilities such as packages and use clauses in Ada

or modules and import commands in Modula-2 allow the programmer to easily translate abstract data types into executable code. For other languages with capabilities for separate compilation of modules, these concepts can be applied by using a precompiler which propagates the ADT-definitions to the modules where they are used.

5 Database Management Systems Tailored to GIS

This chapter investigates how the object-orientation reflects in the architecture of a database management system for GIS. Ideally, an object-oriented database consists of subsystems which can be added or exchanged to support specific tasks [Batory 1986]. Before investigating the GIS-specific parts of a tailored object-oriented database, subsystems of standard database systems are compiled which can be adopted from an object-oriented database.

5.1 Applicable Subsystems of a Conventional Database

5.1.1 Disk Storage Subsystem

A database must provide persistent storage for objects modelled in the information system. Due to the large size of the data sets, use of magnetic or optical disk systems for permanent storage is required.

A storage subsystem must provide operations to store and retrieve data. The storage subsystem does not know anything about other operations owned by the objects or their internal structure. Disk access is expensive and improvements in performance can be achieved by reducing the number of disk accesses by buffering the storage elements.

5.1.2 Multi-User Facilities

In multi-purpose information systems, a database is shared by a large variety of users who often want to access the same data in parallel. The known straight forward techniques are sufficient as long as concurrent users only access the data without changing them. This is, however, usually not appropriate to accomodate users and their needs. Most organizations cannot restrict update operations to a single user without severe distortion of their flow of work.

A database management system must provide control mechanisms in order to guarantee concurrent access for multiple users to all data and to prevent users from accessing data during inconsistent states, i.e., the database management system guarantees that no user sees the preparative stages of a change until the change is completed and made visible to all other users. This is done by a dedicated subsystem, the so-called transaction manager. A transaction is a sequence of operations by one user which transforms the database from one consistent state into another one, such that outside of a transaction a consistent image of the

mini-world exists at all times. A transaction can either be committed, i.e., all modifications will be made visible, or aborted, i.e., none of the modifications started during the transaction will become effective. Modified data are isolated during the transaction preventing other users from uncontrolled access. The transaction subsystem can fulfill with essentially the same mechanisms other goals as well. It is usually assumed that a transaction system will maintain all the consistency of the database across hardware or software malfunctionings, and prevents loss of data. A transaction has the following four classical properties (ACID) [Härder 1985]:

- Atomicity states that in a transaction either all modifications become effective, or no change at all are made in the data set.
- Consistency is guaranteed before and after the transaction.
- Isolation preventes from unauthorized access during a transaction.
- Durability keeps comitted transactions permanently.

5.1.3 Distributed Systems

Central databases are important resources and access to them must be available for many operations in parallel and at the same time. For organizational reasons, large databases are often not centralized, but distributed over several computer systems. This may improve their availability during local hardware failures, reduce data transfer cost, etc. In our opinion, support for distribution should be built into this layer of the subsystem.

5.2 Specific GIS requirements

In this chapter it is proposed which additional concepts must be integrated into an object-oriented database management system, such that it would fulfill the high expectations and requirements on a persistent management system for very large spatial data sets.

5.2.1 Performance Enhancements for Spatial Data

Large spatial data collections should be managed without artificially subdividing them into several user-visible map sheets. Ideally, the user should have a single database covering the entire area of his or her application; however, the performance penalties are serious when very large amounts of spatial data should be managed. Interactive graphic representations of spatial data with map output require especially adequate performance.

Generally, database performance depends upon the number of disk accesses needed to retrieve the data. Conventional systems do not pay attention to the distribution of spatial data and store them 'as they are collected'. This treatment is not appropriate for spatial

data collections, because the access on a set of data to be drawn as a map will take a long time due to too many disk accesses. The larger the data collection grows, the slower the system will perform.

Access on spatial data must be enhanced by integrating spatial storage clusters [Frank 1983b]. Such techniques take advantage of the spatial distribution of data and store them such that spatial neighbors are neighbors on the storage device. It is assumed that (1) data used on one drawing are likely to be re-used soon for further display or interactive modification, and (2) not only a single object, but several adjacent objects will be used on a drawing. Spatial access is organized such that not a single instance, but a set of adjacent data is read at a time from a storage device and afterwards kept in main memory for future access. This buffering scheme needs to be organized according to some rules, e.g., a least-frequently-used strategy. In combination with spatial access methods, fast response time can be guaranteed essentially independent from the size of the data collection.

It is worthwile to study how storage structures for heterogeneously distributed spatial data can be improved such that access on less frequent objects performs well [Egenhofer 1987b].

5.2.2 Complex Geographical Objects

The complexity of geographical objects, primarily spatial objects, requires methods to define and use appropriate data types and operations. The object-oriented model is tailored to this task. Data structures for recursive object definitions, such as areas being subdivided in other areas, and transitive closure operations are necessary.

Geometric and non-spatial data should not be forced to be physically separated from each other or managed in different types of databases. Instead, both categories must be integrated in one single system. This requires that

- user-defined, complex types can be stored in the database.
- a buffering schema is incorporated to reduce physical disk access; the geometric neighborhood should be exploited for a system with primarily graphical output.

Since objects can be more complex, their transactions will last longer and may be nested. A transaction concept is necessary which goes beyond the classical properties atomicity, consistency, isolation, and durability [Härder 1985].

5.3 Object-Orientation in Query Languages

Query languages have been disregarded for a long time, while they are one of the most crucial issues of an object-oriented database. Query languages are expected to benefit from the object-oriented approach by providing object operations at the user interface. Standard query languages for conventional database mangement systems, such as SQL [Chamberlin 1976], QUEL [Stonebraker 1976], or Query-By-Example [Zloof 1977], are not suited to deal with

spatial data. because they do not include the specific properties of spatial objects. Non-standard database management systems must be furnished with query languages which support the treatment of complex objects including their specific properties. Proposals have been made to develop completely new query languages which are tailored to their specific application [Frank 1982a]. A database system for GIS which is able to manage spatial objects efficiently is incomplete if it does not support spatial data and their properties in the query language.

We do not believe that natural language interfaces overcome the problems in query languages for non-standard applications and rather promote structured, object-oriented query languages.

5.3.1 Support of Spatial Relations

In order to treat spatial objects in query languages, properties must be included which are specific for spatial data. Typical properties among spatial objects are topological and metrical relations describing neighborhood, inclusion, distance, and direction. Object-oriented spatial relations must be dimension-independent, i.e., such that they can be applied to any spatial object. 'Disjoint', for example, is a relation which can hold for any two spatial objects (two points, two lines, two areas, two volumes, but also a point and a line, a point and an area, etc.).

Currently, we are investigating how the syntax of a standard query language must be extended such that it includes an adequate treatment of spatial objects [Egenhofer 1987a].

5.3.2 High-Level Object-Oriented Operators

Conventional query languages address subparts of objects explicitly and compare them with standardized operations. This is the reason for the 'artificial' style of queries such as

```
RETRIEVE all roads
WITH road.width > 12
```

In natural language, humans would combine the operation ¿ with the type of the object-subpart to some meaningfull term such as *broader than*. The object-oriented approach pursues this concept since the structure of the object and its operations are closely combined. Consequently, query languages must apply this concept, too, and express relations between objects such as

```
RETRIEVE all roads
```

```
WHICH ARE broader than 12
```

By combining object parts with operators and mapping them onto high-level operators, an object-oriented view of operators can be presented.

5.3.3 Graphical Display

Conventional query languages deal only with alphanumerical data, consequently, the graphical display for spatial objects cannot be specified. The specification is more complex for graphical output than for alphanumerical output where the sequence of columns in a table is described. Methods are needed to specify colors, patterns, and symbols.

Furthermore, interactive sessions with graphical output need mechanisms for user feedback, such as input via mouse on a drawing. The concept of direct manipulation on objects is more advanced and object-oriented than conventional input methods, because it does not reference a specific value, but screen coordinates which correspond to some object.

The output system directed by a query language has some other interesting problems, such as the selection of context [Frank 1982a] which is dispalyed to make the graphical output understandable. Other proposals investigate, how methods from artificial intelligence like LOBSTER [Frank 1984a] can be incorporated into an intelligent interface for query languages.

6 Conclusion

It has been investigated how object-oriented database management systems can serve as suitable tools for spatial information systems. It was outlined that current database technology is not sufficient for the specific tasks when dealing with large amounts of spatial data. Recently, research in non-standard database environments promoted an object-oriented model which looks promissing to overcome some problems that make conventional database management systems unsuitable, such as the lack of modeling power to adequatly describe complex objects and the unacceptably slow performance of current implementations. This paper presented an object-oriented data model based on the abstraction concepts of classification, generalization, and aggregation. Hierarchical and multiple inheritance are crucial for modeling complex object types. The implementation of object-oriented data structures was investigated from the software engineering aspects (object-oriented programming languages).

With respect to geographic applications, several components of an object-oriented database system were identified which are mandatory for treating very large collections of spatial and non-spatial data. In particular, data structures for complex spatial data types, internal storage clusters for fast access on spatial data, and concurrent treatment of spatial and

non-spatial objects must be incorporated in the database management system. Finally, the impact of the object-oriented design on query languages was investigated. Query languages for spatial information systems need high-level relations and operators for spatial objects, methods for specifying graphical display, and language tools for direct manipulation.

Acknowledgement

Thanks to Doug Hudson and Jeff Jackson who helped prepare this paper.

References

[Ada 1983] United States Department of Defense. Reference Manual of the ADA Programming Language. Springer Verlag, New York, NY, 1983.

[Aronson 1983] P. Aronson and S. Morehouse. The ARC/INFO Map Library: A Design for a Digital Geographic Database. In: Auto-Carto VI, 1983.

[Batory 1984] D.S. Batory and A.P. Buchmann. Molecular Objects, Abstract Data Types, and Data Models: A Framework. In: 10th VLDB conference, Singapore, 1984.

[Batory 1986] D.S. Batory. GENESIS: A Project to Develop an Extensible Database Management System. In: K. Dittrich and U. Dayal, editors, International Workshop in Object-Oriented Database Systems, Pacific Grove, CA, 1986.

[Beckstedt 1987] M. Beckstedt. Clark County Land Parcel Mapping System Pilot Study. In: Proceedings AM/FM International Automated Mapping/Facilities Management Conference X, Snowmass, CO, 1987.

[Brodie 1984a] M.L. Brodie et al., editors. On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages. Springer Verlag, New York, NY, 1984.

[Brodie 1984b] M.L. Brodie. On the Development of Data Models. In: M.L. Brodie et al., editors, On Conceptual Modelling, Springer Verlag, New York, NY, 1984.

[Buchmann 1985] A.P. Buchmann and C.P. de Celis. An Architecture and Data Model for CAD Databases. In: 11th VLDB conference, Stockholm, Sweden, 1985.

[Cardelli 1985] L. Cardelli and P. Wegner. On Understanding Type, Data Abstraction, and Polymorphism. ACM Computing Surveys, 17(4), April 1985.

[Chamberlin 1976] D.D. Chamberlin et al. SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Journal of Research and Development 20(6), 1976.

[Chen 1976] P.S.S. Chen. The Entity-Relationship Model: Towards a Unified View of Data. ACM Transactions on Database Systems, 1(1), March 1976.

[CODASYL 1971] Data Base Task Group Report. CODASYL. Technical Report, New York, NY, April 1971.

[Codd 1970] E.F. Codd. A Relational Model for Large Shared Data Banks. Communications of the ACM, 13(6), June 1970.

[Codd 1982] E.F. Codd. Relational Data Base: A Practical Foundation For Productivity. Communications of the ACM, 25(2), February 1982.

[Dahl 1968] O.-J. Dahl et al. SIMULA 67 Common Base Language. Technical Report N. S-22, Norwegian Computing Center, Oslo, Norway, October 1968.

[Davidson 1987] L.J. Davidson et al. ObjectLogo. Coral Software Corporation, Cambridge, MA, 1987.

[Dayal 1986] U. Dayal and J.M. Smith. PROBE: A Knowledge-Oriented Database Management System. In: M.L. Brodie and J. Mylopoulos, editors, On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies, Springer Verlag, New York, NY, 1986.

[Dittrich 1986] K. Dittrich. Object-Oriented Systems: The Notation and The Issues. In: K. Dittrich and U. Dayal, editors, International Workshop in Object-Oriented Database Systems, Pacific Grove, CA, 1986.

[Eastman 1980] C. Eastman. System Facilities for CAD Databases. In: 17th Design Automation Conference, Minneapolis, 1980.

[Egenhofer 1986] M. Egenhofer and A. Frank. Connection between Local and Regional: Additional 'Intelligence' Needed. In: FIG XVIII. International Congress of Surveyors, Commission 3, Land Information Systems, Toronto, Ontario, Canada, 1986.

[Egenhofer 1987a] M. Egenhofer. An Extended SQL Syntax To Treat Spatial Objects. In: Y.C. Lee, editor, Proceedings of the Second International Seminar on Trends and Concerns of Spatial Sciences, Fredericton, New Brunswick, Canada, 1987.

[Egenhofer 1987b] M. Egenhofer. Managing Spatial Storage For Structured Data. In: Congres Conjoint Carto-Québec / A.C.C. / C.C.A., Québec, Canada, 1987.

[Frank 1981] A. Frank. Applications of DBMS to Land Information Systems. In: C. Zaniolo and C. Delobel, editors, Seventh International Conference on Very Large Data Bases, Cannes, France, 1981.

[Frank 1982a] A. Frank. MAPQUERY—Database Query Language for Retrieval of Geometric Data and its Graphical Representation. In: D. Bergeron, editor, SIGGRAPH '82, ACM Computer Graphics, Boston, MA, July 1982.

[Frank 1982b] A. Frank. PANDA—A Pascal Network Database System. In: G.W. Gorsline, editor, Proceedings of the Fifth Symposium on Small Systems, Colorado Springs, CO, 1982.

[Frank 1983a] A. Frank. Data Structures for Land Information Systems—Semantical, Topological, and Spatial Relations in Data of Geo-Sciences, (in German). PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1983.

[Frank 1983b] A. Frank. Problems of Realizing LIS: Storage Methods for Space Related Data: The Field Tree. Technical Report 71, Institut for Geodesy and Photogrammetry, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 1983.

[Frank 1984a] A. Frank. Extending a Database with Prolog. In: L. Kerschberg, editor, Proceedings of the First International Workshop on Expert Database Systems, Kiawah Island, SC, October 1984.

[Frank 1984b] A. Frank. Requirements for Database Systems Suitable to Manage Large Spatial Databases. In: International Symposium on Spatial Data Handling, Zurich, Switzerland, August 1984.

[Frank 1986] A. Frank and W. Kuhn. Cell Graph: A Provable Correct Method for the Storage of Geometry. In: D. Marble, editor, Second International Symposium on Spatial Data Handling, Seattle, WA, 1986.

[Goldberg 1983] A. Goldberg and D. Robson. Smalltalk-80. Addison-Wesley Publishing Company, 1983.

[Guttag 1977] J. Guttag. Abstract Data Types And The Development Of Data Structures. Communications of the ACM, June 1977.

[Härder 1985] T. Härder and A. Reuter. Architecture of Database Systems for Non-Standard Applications, (in German). In: A. Blaser and P. Pistor, editors, Database Systems in Office, Engineering, and Scientific Environment, Springer Verlag, New York, NY, March 1985. Lecture Notes in Computer Science, Vol. 94.

[Herring 1987] J. Herring. TIGRIS: Topologically Integrated Geographic Information Systems. In: N.R. Chrisman, editor, Eighth International Symposium on Computer-Assisted Cartography, Baltimore, MD, March 1987.

[Jensen 1978] K. Jensen and N. Wirth. Pascal User Manual and Report. Springer-Verlag, New York, NY, 1978.

[Johnson 1983] H.R. Johnson et al. A DBMS Facility for Handling Structured Engineering Entities. In: ACM Engineering Design Applications, 1983.

[Kemper 1987] A. Kemper and M. Wallrath. An Analysis of Geometric Modeling in Database Systems. ACM Computing Surveys, 19(1), March 1987.

[Kjerne 1986] D. Kjerne and K.J. Dueker. Modeling Cadastral Spatial Relationships Using an Object-Oriented Language. In: D. Marble, editor, Second International Symposium on Spatial Data Handling, Seattle, WA, 1986.

[Lipeck 1986] U.W. Lipeck and K. Neumann. Modelling and Manipulation of Objects in Geoscientific Databases. In: 5th International Conference on the Entity-Relationship Approach, Dijon, France, 1986.

[Lorie 1983] R. Lorie and W. Plouffe. Complex Objects and Their Use in Design Transactions. In: ACM Engineering Design Applications, 1983.

[Maier 1986] D. Maier. Why Object-Oriented Databases Can Succeed Where Others Have Failed. In: K. Dittrich and U. Dayal, editors, International Workshop in Object-Oriented Database Systems, Pacific Grove, CA, 1986.

[Nguyen 1986] V. Nguyen and B. Hailpern. A generalized Object Model. SIGPLAN Notices, 21(10), October 1986.

[O'Brien 1986] P. O'Brien et al. Persistent and Shared Objects in Trellis/Owl. In: K. Dittrich and U. Dayal, editors, International Workshop in Object-Oriented Database Systems, Pacific Grove, CA, 1986.

[Olthoff 1985] W. Olthoff. An Overview on ModPascal. SIGPLAN Notices, 20(10), October 1985.

[Parnas 1978] D.L. Parnas and J.E. Share. Language Facilities for Supporting the Use of Data Abstraction in the Development of Software Systems. Technical Report, Naval Research Laboratory, Washington, DC, 1978.

[Schek 1985] H.-J. Schek. Towards a Basic Relationsal NF^2 Algebra Processor. In: Conference on Found. Data Org. (FODO), Kyoto, Japan, 1985.

[Schek 1986] H.-J. Schek and W. Waterfeld. A Database Kernel System for Geoscientific Applications. In: D. Marble, editor, Second International Symposium on Spatial Data Handling, Seattle, WA, 1986.

[Shipman 1981] D. Shipman. The Functional Data Model and the Data Language DAPLEX. ACM Transactions on Database Systems, 6(1), March 1981.

[Sidle 1980] T.W. Sidle. Weakness of Commercial Database Management Systems in Engineering Applications. In: 17th Design Automation Conference, 1980.

[Smith 1986] T. Smith et al. KBGIS-II: A Knowledge-Based Geographic Information System. Technical Report TRCS86-13, Dept. of Computer Science, University of California, Santa Barbara, CA, May 1986.

[Snyder 1986] A. Snyder. CommonObjects: An Overview. SIGPLAN Notices, 21(10), October 1986.

[Stonebraker 1976] M. Stonebraker et al. The Design and Implementation of INGRES. ACM Transactions on Database Systems, 1(3), September 1976.

[Stonebraker 1982] M. Stonebraker and A. Guttman. Using a Relational Database Management System for CAD data. IEEE Database Engineering, 5(2), June 1982.

[Stonebraker 1983a] M. Stonebraker et al. Application of Abstract Data Types and Abstract Indices to CAD Databases. In: D. DeWitt and G. Gardarin, editors, Proceedings of ACM SIGMOD Conference on Engineering Design Applications, San Jose, CA, 1983.

[Stonebraker 1983b] M. Stonebraker et al. QUEL as a datatype. Technical Report UCB/ERL M83/73, University of California, Berkley, CA, 1983.

[Storm 1986] R. Storm. A Comparison of the Objet-Oriented and Process Paradigms. SIGPLAN Notices, 21(10), October 1986.

[Stroustrup 1986] B. Stroustrup. The C++ Programming Language. Addison-Wesley Publishing Company, 1986.

[Tripp 1987] R. Tripp. AM/FM Progress at Carolina Power & Light Company. In: Proceedings AM/FM International Automated Mapping/Facilities Management Conference X, Snowmass, CO, 1987.

[Udagawa 1984] Y. Udagawa and T. Mizoguchi. An Extended Relational Database System for Engineering Data Management. IEEE Database Engineering, 7(2), June 1984.

[Weinreb 1981] Lisp Machine Manual. Symbolics, Inc., 1981.

[Wilkins 1984] M.W. Wilkins and G. Wiederhold. Relational and Entity-Relationship Model Databases and VLSI Design. IEEE Database Engineering, 7(2), June 1984.

[Wirth 1982] N. Wirth. Programming in Modula-2. Springer-Verlag, New York, NY, 1982.

[Woelk 1987] D. Woelk and W. Kim. Multimedia Information Management in an Object-Oriented Database System. In: P. Stocker and W. Kent, editors, 13th VLDB conference, Brighton, England, 1987.

[Zilles 1984] S.N. Zilles. Types, Algebras, and Modelling. In: M.L. Brodie et al., editors, On Conceptual Modelling, Springer Verlag, New York, NY, 1984.

[Zloof 1977] M.M. Zloof. Query-by-Example: A Database Language. IBM Systems Journal 16(4), 1977.

Session 4: Architecture and query languages

- Layered architecture
- Object-oriented query languages
- Existing object-oriented DBMS
- Education
- Conclusion

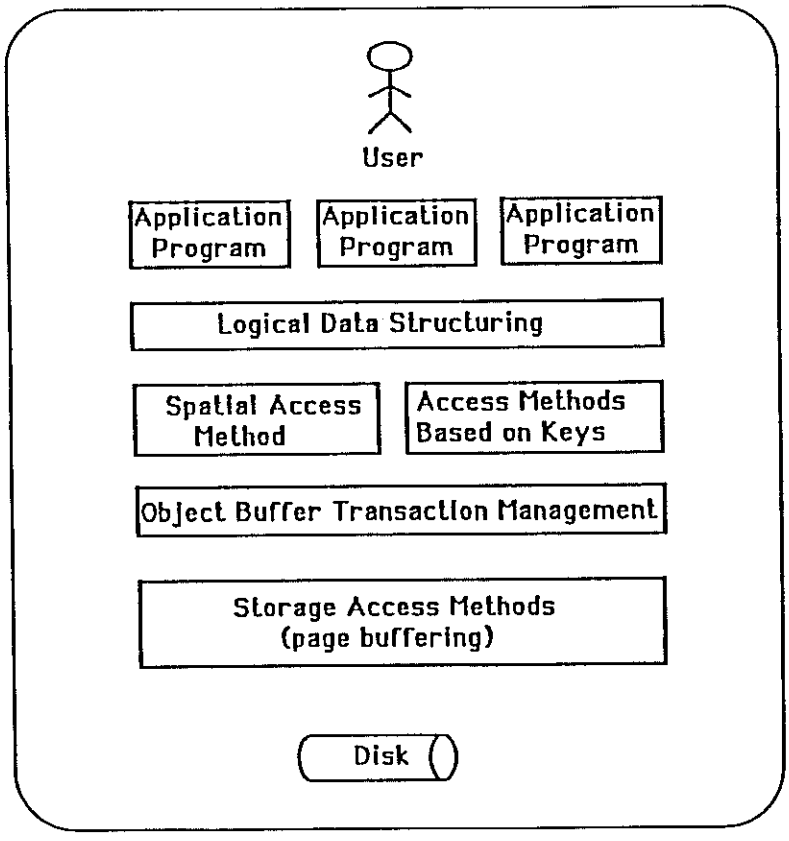
An object-oriented database can be

- a database with an object-oriented interface integrated in an object-oriented language.
- a database implementation using an object-oriented concept.

The implementation of a database with an objct-oriented interface need not be substantially different from a relational or network database. Different optimizations are possible, especially to support complex objects.

Layered Architecture

- Useful for implementation and for discussion.
- Each layer provides some functionality to the layer atop.



Storage Access Layer:

- Interface to the operating system
- Physical clustering
- Buffer management

Interface to the Operating System:

- Standardized access to files
- Guaranteed physical writes
- Possible inference with operating system optimization
- Physical structure of files

(Caveat: UNIX file system is not well suited for dbms operations)

Physical Clustering:

Access to data is by reading 'disk pages' from disk (average 30 msec per read).

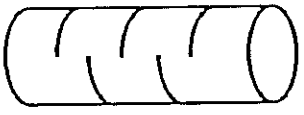
Store more data than necessary on the same page (cluster).

Prediction of what data will be used at the same time.

For Spatial Information Systems:

Data in a neighborhood is likely to be used at the same time.

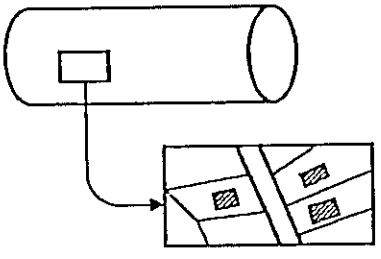
Storage should preserve neighborhood.



Regular Distribution of Data on Disk

→ Many Accesses

→ Slow Response



Clustered Storage on Disk

→ Few Accesses

→ Fast Response

Buffering:

Access to data in main memory is much faster than access to data on disk (100 nano sec (10^{-7}) compared to 30 msec (10^{-3})).

Keep data once read in a buffer in main memory, expecting it will be used again.

Access Methods:

- Access by unique key
- Access using logical data structures:
  - Aggregates
  - Access in determined order (sorted)
- Access by spatial location

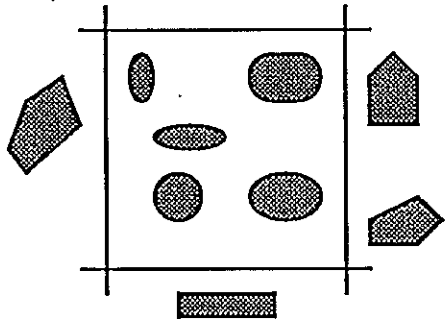
Access by spatial location is typical for spatial DBMS like AM/FM.

Typical Query:

Retrieve all <object-type> within <area>

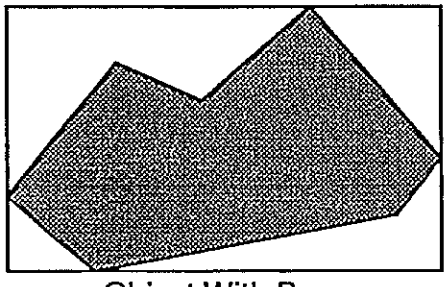
e.g. Retrieve all water-mains within Brooklyne-north.

Can be simplified to



Query Window

Provided the minimal bounding rectangle is stored with every spatial object in the DBMS.



Object With Box
(Minimal Bounding Rectangle)

SQL and other relational query languages

- access records and fields
- provide only standard operations on the standard data types

This leads to complicated expressions, e.g.:

```
SELECT  name, width
FROM    road
WHERE   width > 12
```

should be:

```
LIST name, width OF
     roads broaderThan 12 feet.
```

An object-oriented query language must include user defined operations on the objects:

- How to combine with standard syntax?
- How to prompt the user about available operations?

Would a 'functional style' be more appropriate?

An SQL interface to an object-oriented database could be constructed.

- All value returning functions which access a single object could be considered fields.
- Aggregates can be replaced by key-fields, suitable for joins.

This gives up most advantages of object-orientation, including performance.

Existing object-oriented DBMS

Two commercially available systems:

- GEMSTONE: an extension of Smalltalk with persistent objects and most DBMS functionalities.
- Vbase: based on a C precompiler (COP). Most DBMS functionalities including transaction processing and SQL.

A number of research systems:

- Postgres: extends the relational data model with ADTs (Stonebraker).
- PROBE: A DBMS based on a functional model, deals with complex objects (Dayal).
- DASDBMS: NF^2 (non-first normal form) extension of relational DBMS, an extension of relational algebra to include complex objects (Schek).
- PRIMA: A non-standard DBMS for complex objects (e.g., solid modeling, molecules in chemestry), incl. SQL-type query language (Haerder).
- PANDA: Pascal based, evolves to a new language/software engineering system called MOOSE. Contains special support for storage of spatial objects and access based on location.

1st Workshop on Object-Oriented Database Management Systems in Monterey (CA), 1986
Proceedings (ACM - currently out of print).

2nd workshop in Germany, October 1988

Education

Object-oriented design and programing: Courses are available from programming languge vendors e.g., Stepstone (Objective C), Ontologic (Vbase) Short courses with instructors from research groups.

Object-oriented design is a new topic in university computer science curricula:
Courses at universities: Most advanced COS departments offer regular courses dealing with object-oriented design. North Eastern University and other similar institutions offer evening graduate courses.

Conclusions

- GIS must be built using DBMS technology with multi-user and transaction support.
- Spatial data are too complex to be managed within the relational data model.
- Object-oriented DBMS are needed for the treatment of complex objects in GIS.

An object-oriented DBMS cannot be considered isolated from the object-oriented design and object-oriented programming.
An object-oriented software environment is required. Such an environment will include also graphic support, human interface manager, etc.

PANDA: An Extensible DBMS Supporting Object-Oriented Software Techniques*

Max J. Egenhofer
Andrew U. Frank
National Center for Geographic Information and Analysis
and
Department of Surveying Engineering
University of Maine
Orono, ME 04469, USA
MAX@MECAN1.bitnet
FRANK@MECAN1.bitnet

Abstract

Non-standard database management systems are expected to overcome some of the problems that conventional relational systems cannot solve. PANDA is a databases management system that was designed for non-standard applications which deal with spatial data and has been used in research and university teaching for several years. PANDA supports an object-oriented program design with modularization, encapsulation, and reusability, and can be easily embedded into complex applications, such as spatial information systems, CAD/CAM systems, or cartographic expert systems. It is presented how complex objects and their operations are defined such that they can be easily incorporated into an object-oriented application. A layered structure on top of the programmer's interface provides object operations which can include complex consistency constraints.

1 Introduction

Databases have become an accepted tool for storing data in readable and accessible form. Like other engineering applications, such as CAD/CAM or VLSI design, spatial information systems integrate a database system into a larger software system [Frank 1983a] providing the fundamental facilities for sharing data in a multi-user environment, such as persistency,

*This work was partially funded by a grant from NSF under No. IST-8609123 and equipment grants from Digital Equipment Corporation.

concurrency, and transaction management. Existing commercial database systems are not sufficient for these tasks: Performance is unacceptable when a database is populated with large amounts of data [Wilkins 1984] [Härder 1985] [Maier 1986]; the treatment of complex objects [Lorie 1983], such as molecules in chemistry [Batory 1984], spatial objects in GIS/LIS [Frank 1984], or circuits in VLSI, is not supported; appropriate mechanisms for data structuring [Johnson 1983], especially for structuring real-world data, are missing. Object-oriented database management systems [Dittrich 1986] may overcome some of the problems that relational systems cannot solve.

The interface of a non-standard DBMS must fit well into the methods used for the implementation of the applications; therefore, software engineering considerations are of particular importance for non-standard database management systems. Interfaces, like Embedded SQL, have demonstrated to be cumbersome [Christensen 1987]. Theory in software engineering provided some suitable techniques, such as the concept of abstract data types [Guttag 1977] [Parnas 1978] [Zilles 1984] and abstract object types [Sernades 1987], in which each module encapsulates an object type with all its pertinent operations. Standard database systems, based on networks [CODASYL 1971] or the relational data model [Codd 1970], do not fit easily into this concept. A number of methods to capture more semantics in the data model have been proposed in the literature [Brodie 1984a], but most of these methods have not been implemented, and none of them is readily available.

This paper describes the software engineering techniques used in PANDA [Frank 1982], an object-oriented database tailored for applications with spatial data. PANDA is an acronym for Pascal Network Database Management System. The system was originally designed as a network DBMS with object-oriented concepts and enhancements for storage and access of spatial data and evolved over the years to an object-oriented design. PANDA is a large software system consisting of about 40,000 lines of program code, written consistently in precompiled Pascal [Egenhofer 1988b]. PANDA uses commonly available software engineering techniques which made it easy to transport PANDA from one system to another. Its development started at the Swiss Federal Institute for Technology, Zurich, and has been continued at the University of Maine. Code was transferred between such vastly different hardware and operating systems as DEC-10 (under TOPS-10), IBM 370 (under VM/CMS), and VAX/MicroVAX (under VMS), profitting from the flexible structure of precompiled Pascal. The PANDA database system has been running for several years, primarily used for research and teaching in undergraduate and graduate courses.

PANDA's data model is based on the basic concepts of abstraction: classification, generalization, and aggregation [Brodie 1984b]. Frequently used terms in this paper are *object*, for a single occurrence (instantiation) of data; *type*, *object type*, *class*, or *abstract data type*, refering to sorts of objects; *superclass*, describing the grouping of several classes in an is_a relation [Dahl 1968] [Goldberg 1983]; and *subclass*, being the specialization of a superclass. *Inheritance* defines a superclass in terms of one or more other classes, propagating the properties of the superclass to all subclasses transitively. Hierarchical or strict inheritance implies

that each subclass belongs only to a single group of hierarchies, while multiple inheritance models class lattices with acylcic directed graphs [Cardelli 1984] [Woelk 1987].

Throughout this paper, the examples will refer to the following user model: A *node* describes a zero-dimensional spatial object with a coordinate-tuple describing its location; an *edge* is a one-dimensional spatial object; and, a *face* is a two-dimensional object. For the sake of simplicity, the latter two objects will not have lower structured parts. *Nodes*, *edges*, and *faces* are specializations of *spatial objects*. Spatial objects of dimension n are delimited by a number of spatial objects of dimension n-1. Each spatial object has a minimal bounding rectangle (MBR) approximating its spatial location.

The paper starts with a discussion of object-oriented software engineering. PANDA's software environment and architecture are introduced. Chapter 3 focusses on PANDA's object model. The four phases during the definition of complex object types are presented. In chapter 4 the implementation is of object operations is described. The design of a layered structure on top of the generic programmer's interface is presented. The paper concludes that better object-oriented programming languages can help to implement object-oriented databases closer to the designed models.

2 Software Engineering Aspects

It is important that database methods and techniques fit well within a software engineering environment. Theoretically founded methods have proven to be well-suited for modularization based on abstract data types. The deficiency observed is that concepts in software engineering do not match with database models, or are not suitable for them.

An abstract data type (ADT) is a mathematical structure which fully defines the behavior (semantics and operations) of objects. Abstract data types are specified as an algebra describing what sorts of objects (types) are dealt with, and what kinds of operations they are subject to. A set of axioms determines the effects of the operations. Abstract data types can be combined in layers, where higher-level abstract data types are first described independently (specifications), and it is then shown how this behavior can be achieved using other, hierarchically lower abstract data types (abstract implementation [Olthoff 1985] [Frank 1986]). The implementation of abstract data types leads immediately to modular packages with type definitions and implementations of the operations. This is of particular importance for object-oriented programming where objects consist of a type definition and a collection of operations through which objects can be accessed or manipulated exclusively.

Object-oriented programming of object-oriented applications depends upon the choice of the programming language. We claim that object-oriented applications cannot be achieved without object-oriented tools and concepts. These concepts do not necessarily include a message-passing paradigm [Goldberg 1983] that is often cited as necessary for an object-oriented design. It was outlined that message-passing is a pedagogical rather than a seman-

tical difference to conventional routine calls, and any procedure call in an Algol-like language could be seen as message-passing [Storm 1986]. Currently, only a few languages support object-orientation, but none of them has all the features desired [Edelson 1987].

2.1 PANDA's Software Environment

The software engineering environment of PANDA supports *modularization*, *encapsulation*, and *reusability*, giving rise to simulate inheritance and late binding. Programmers are motivated to write object-oriented code in a standardized way such that other programmers can easily read and correct it. Modularization is achieved by using a Pascal precompiler [Egenhofer 1988b]. Types and routines are provided from one module to another, with type checking across modules. Unlike other object-oriented implementations in Pascal which suppress type checking by using pointers as parameters [Jacky 1987], PANDA is written in a strongly typed Pascal that incorporates object-oriented programming style. The implementation of operations is encapsulated into the module and thus not visible from the outside. A highly modular programming style requires that the modules are managed in a controlled library system providing the user information about existing ADTs, their operations, and their specifications. The precompiler is embedded into a library management system with hierarchical directory structure where several versions of modules can be kept in parallel [Egenhofer 1987]. Compatibility of code among different Pascal compilers and various hardware is guaranteed by the precompiler, because only the precompiler itself, not the particular code, must be adapted to fit specific compiler features.

The full benefits of this layered, object-oriented design are noticed during software maintenance: changes to the data definition require only recompilation of the corresponding modules and relinking of their (shareable) images. The lifetime of PANDA has been influenced by the modular programming paradigm when subparts could be recoded without influencing the other parts.

2.2 PANDA's Architecture

PANDA consists of an implementation-independent database kernel and application-specific object layers on top of the kernel (Figure 1).

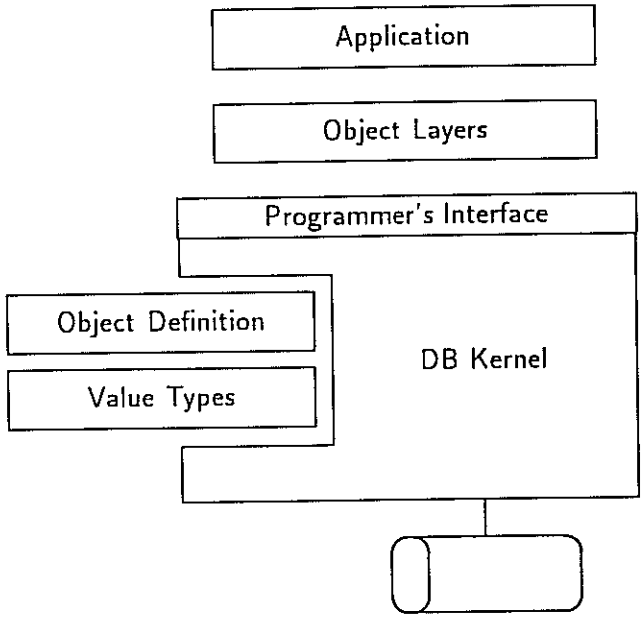


Figure 1: The architecture of PANDA.

PANDA's DB kernel manages storage and retrieval of data from persistent storage devices, buffers pages and records to improve performance, provides facilities for transaction management, incorporates structures for logical access (hashing, B*-trees, Field Tree for spatial access [Frank 1983b]), and provides generic structures for aggregation and generalization. The kernel is built in a layered and modular structure so that it can be easily extended by additional parts, e.g., a multi-user facility. The application specific object definition is plugged into the kernel. This extensible part of PANDA is based on the definition of value types which can be arbitrarily complex acyclic abstract data types.

The programmer's interface of the kernel is a collection of object-oriented manipulation and retrieval operations which can be called from the application programs. These operations are defined for a generalized object class, the superclass of all user-defined object types, and are compatible with any object type. On top of the programmers's interface is a layered structure of model-specific object operations.

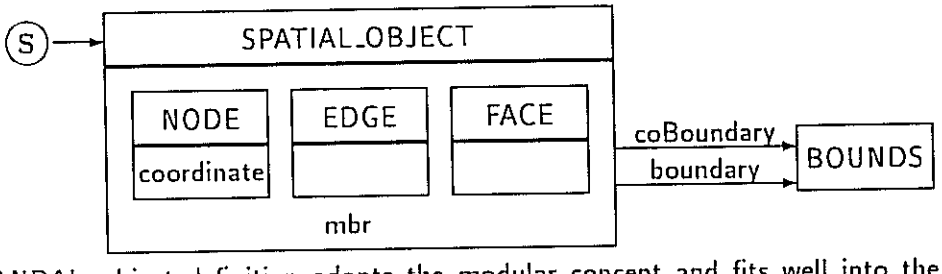
An object-oriented data model gives rise to the formation of arbitrarily complex object types. The majority of object types is composed of a (limited) number of lower structured subparts which occur repeatedly as part of various object types. Instead of redefining these parts for a multitude of objects, PANDA allows the database administrator to define object components which are used to compose the object types. Though this may resemble a traditional entity-attribute type of concept, it is only the object-oriented implementation of several object types with the same components which are defined only once and reused in

every class they are part of. This model exploits the similarities of object types and liberates the database administrator from the task of defining each object type from scratch.

3 Object Definition

The object definition in PANDA is based upon the declaration of a set of names and the definition of how the names are combined to form objects. *Class names* are identifiers to distinguish between different classes; *component names* are identifiers for the components of classes; *value names* refer to the ADT modules of which object types are composed; *link names* are the identifiers that describe aggregations of several object types; and, *path names* identify various logical ways to access objects. The database administrator combines them as follows: (1) a *value name* is assigned to each *component name* (establishing a 1:m-relation), (2) *component names* are assigned to the corresponding *class names* (1:m), (3) the immediate superclass is associated with each subclass (1:m)[1], (4) *classes* and *link names* are combined performing aggregates (m:n), and (5) logical *access paths* are assigned to *classes* (m:1). With these definitions, the schema can be completely described. Transitive closure is applied to propagate properties of superclasses to subclasses, where components, links, and logical access paths are considered as properties.

Graphical tools, similiar to the entity-relationship diagrams [Chen 1976], are useful in the description of the schema. The following conventions are made [Egenhofer 1988a]: A class is represented by a box drawn around the class name; the names of object components are placed in the class box under the class name; generalization is drawn as box around a box, such that the superclass contains the subclass(es); aggregations are indicated as arrows running from one box to another; and, various logical access paths are depicted by specific symbols[2]. The geometric example, described in chapter 1, could be designed graphically as shown in figure 2.



PANDA's object definition adopts the modular concept and fits well into the layered architecture. An application model is defined in standardized operations that are easily integrated into the kernel. The operations, called *ini-routines*, are implemented as short

[1] This model supports only single inheritance; it is a temporary restriction in PANDA.

[2] For instance, a S in a circle describes spatial access.

Pascal routines which are executed during PANDA's startup. For each object type a separate *ini-module* is defined. This approach releases the database administrator from recompiling the entire DBMS code; instead, only the ini-modules and an interface module are compiled, and (shareable) images are linked. A systematic implementation with strict naming conventions allows the generatation of code for the ini-modules. Currently, code is generated partially from definitions in a text file; a more user friendly generation immediately from the graphical schema design is being investigated.

Four phases are distinguished during object definition: (1) declaring the terms of the user's model, (2) implementing value types as Pascal modules, (3) mapping value types on components and composing classes of components and/or other subclasses, and (4) generalizing all classes to a common object class. Each step relies upon the completion of the previous definitions.

3.1 Phase 1: Declaration of User Terms

The kernel was originally initiated with templates for class names, component names, link names, and path names, and these templates must be replaced with the particular names used. The graphical diagram can be systematically translated into the required text form. Each name in the head of a box is a *className*; the remaining names in boxes are components which must be tagged with the corresponding className to form *componentNames*; names related with arrows are *linkNames*. The following name declaration was derived from the grahical schema example in figure 2.

```
classTypes:  node, edge, face, bounds, spatialObject;
componentTypes:  nodeCoordinate, spatialObjectMbr;
linkTypes:  boundary, coBoundary;
```

3.2 Phase 2: Definition of Value Types

A value type is the Pascal implementation of an abstract data type in a single programming unit; therefore, the module for a value type consists of a type definition and the set of pertinent operations. Value types are the implementation of class components and are managed in a library system such that previously defined components are reusable for the design of other databases. Existing implementations can be extended or modified, and new definitions added.

The type definition can be any simple data type, such as integer, string, real, etc., or any structured type, such as array or record, except types for dynamically allocated variables, such as pointers[1]. For example, a one-dimensional interval (ivl) can be expressed as a value

[1] This limitation is with respect to the intended persistency of objects.

type composed of two integers.

```
TYPE iv1Type = RECORD
                 low, high: integer;
               END;
```

Value types can be built upon other value types by using their definitions and methods. Due to restrictions of current compilers, only acyclic combinations of value types are permitted. The following example shows a two-dimensional interval (iv2) that is composed of two one-dimensional intervals:

```
IMPORT iv1 FROM meterv[1];
TYPE iv2Type = RECORD
                 Xiv, Yiv:  iv1Type;
               END;
```

Indexing and hashing structures support the efficiency of database management. Specific object-operations are needed for each object type that is supported by such a structure [Rowe 1987]. For example, hashing requires a function that calculates a hash value, and an operation that compares two values for equality. The implementation of these operations depends upon the structure and the semantics of the object type. Unlike traditional databases that support only a limited, hardcoded set of types, PANDA is extensible. This implies that the operations for the supporting structures are not predefined and must be provided by the application designer. For example, if the data model defines a direct access path for an objectType upon a one-dimensional interval (iv1), then the following two operations—calculating a hash value for an interval and checking two intervals for equalitiy—must be provided for the valueType iv1Type:

```
FUNCTION iv1Hash (i: iv1Type): hashType;
INTENT to calculate a hash value for a one-dimensional interval;
BEGIN
  iv1Hash := hashAdd (intHash (i.low), intHash (i.high));
END;

FUNCTION iv1Equal (i1, i2: iv2Type): boolean;
INTENT to check whether two 1-d intervals are equal;
```

[1]The import statement provides the types of the module iv1 from the library meterv to satisfy the compilation of the module with the iv2Type.

```
BEGIN
  iv1Equal := intEqual (i1.low, i2.low) and intEqual (i1.high, i2.high);
END;
```

The decomposition of objects into components releases the application designer from redundant definitions of specific object operations for the structures. Considerable overhead is removed by defining the operations for the value types and applying them for the class types.

3.3 Phase 3: Composition of Class Types

A fundamental class is composed of zero to many components. Details about the implementation of a class are hidden, such that modules outside of the class definition are not aware of the decomposition into object components. For example, the class type *node* consists of the coordinates which are implemented as the value type *ptType*. The data type for the node class is written as follows:

```
TYPE nodeType = RECORD
                   coordinate:  ptType[5];
                END;
```

A generalized class is composed of zero to many components that are inherited to all subclasses. While some programming languages, such as Smalltalk-80 [Goldberg 1983] or C++ [Stroustrup 1986], support strict inheritance directly, Pascal does not. With the help of variant records, hierarchical inheritance of types can be simulated. For example, the class *spatialObject*, being the superclass of the classes *node*, *edge*, and *face*, with the component *mbr* is implemented as follows:

```
TYPE spatialObjectType = RECORD
                            mbr:  iv2Type;
                            CASE classTypes[6] of
                              node:  (nodeF: nodeType);
                              edge:  (edgeF: edgeType);
                              face:  (faceF: faceType);
                            END;
                         END;
```

Unfortunately, these type definitions are not yet sufficient for the object definition. Today's programming languages of the FORTRAN/Algol type show a major limitation. Compilation

[5] ptType is a value type which was defined previously.
[6] classTypes is an enumerated type over all class names used.

and execution of code are separated into two phases which provide different levels of information. During compilation, variables and types are named, and types may be composed from other types. During executing, the names and the relation between a variable and its type is not accessible by hte user code. Likewise, it is not possible to find out whether a type is part of another structured type.

In PANDA, the database administrator has to write a separate routine for each class type that defines the class, its components, and its immediate superclass. These routines will be executed by the kernel during startup, initializing the user's model. For example, the class *node* is defined in the following ini-routine.

```
PROCEDURE nodeIni;
INTENT to define the class node;
BEGIN
  classPut[7] (node);
  classPutComponent[8] (node, nodeCoordinate, 'ptType');
  classPutSuperclass[9] (node, spatialObject);
END;
```

Besides the particular composition of the class, possible aggregations and logical access paths are defined the same way. For example, a *spatialObject* can be accessed spatially, and it can be part of the two aggregates *boundary* and *coBoundary*.

```
PROCEDURE spatialObjectIni;
INTENT to define the class spatialObject;
BEGIN
  classPut (spatialObject);
  classPutSpatialAccess[10] (spatialObject);
  classPutComponent (spatialObject, mbr, 'iv2Type');
  classPutLink[11] (coBoundary, spatialObject, bounds);
  classPutLink (boundary, spatialObject, bounds);
END;
```

For each class type, the composition of the type and the corresponding ini-routine are combined in a separate module.

[7] classPut defines a class type.
[8] classPutComponent assigns a component implemented as value type to a class.
[9] classPutSuperclass defines a generalization between the subclass and the superclass.
[10] classPutSpatialAccess defines spatial access for a class.
[11] classPutLink defines an aggregate.

3.4 Phase 4: Definition of Database Objects

Conceptually, all classes are specializations of the superclass *objectType* from which all pertinent DB-operations are inherited. The objectType is implemented as a Pascal record with varying parts for each specific classType. The generalized objectType combines all the specific classTypes into a single, compatible type, and common (system) component, such as tuple identifiers and pointers, are added. The following example shows the implementation of the *objectType* with the immediate subclass *spatialObject*.

```
TYPE objectType = RECORD[12]
                    recnr:  felpType[13];
                    links:  objectPointerType[14];
                    CASE[15] class:  classTypes OF
                      ...
                      spatialObject:  (spatialObjectF: spatialObjectType);
                      ...
                    END;
                  END;
```

All database operations, such as *store*, *delete*, and *update* are inherited to the classes. The following chapter presents these operations and their implementation as object operations.

4 A Layered Structure of Object Operations

The object-oriented approach requires the definition of complex objects *and* their pertinent operations. While the knowledge about the composition of complex object types is essential to the database kernel, the object operations are the application programmers' tools to manipulate objects. This is the location to implement consistency checks. In PANDA, these operations are built on top of the kernel. A layered structure of object operations has been developed which is generally applicable for any object-oriented application. Based upon the fundamental object operations to *store*, *delete*, *modify*, and *retrieve* an individual object, more complex operations can be defined. By restricting the operations to a single task, the code for the routines stays small and correctness can be verified more easily.

The structure of these operations is the same for every application: First, the operations are defined to make a specific object, and to assign values to and access them from an object. Then, unary object operations for storing, modifying, deleting, and accessing individual objects are defined based upon the generic DB-operations offered in the programmer's

[12] The first two fields of the objectType are common to all object types.

[13] `felp` stands for file element pointer and is the unique tuple identifier of an object.

[14] Pointers are used to establish aggregate structures.

[15] Here begin the individual objectType definitions.

interface. Another layer treats all binary operations manipulating aggregates. These operations are exploited in the next layer to form complex object operations, including complex consistency constraints.

Conventional database management systems try to integrate consistency constraints into the DBMS kernel. PANDA embeds them in this layered structure, a more desired treatment, because many constraints for complex objects are too complicated to be managed internally. Subsequently, the different layers, or *levels*, are introduced.

4.1 Level 1: Make, Get, and Put Operations

The first layer is a collection of modules with the basic operations to manipulate the individual description of a single object: creating an instance (*make*), assigning values to (*put*), and extracting values from an instance (*get*). These operations hide the implementation of the object types from the user, preventing uncontrolled access. For example, the get and put operations for the component *coordinate* hide the internal structure of the coordinate. The same operation could be used if the coordinates represented a triple of x, y, and z values for a three-dimensional spatial model.

Since the properties of a superclass are propagated to all subclasses, *get* and *put* operations are compatible with the subclasses as well. For example, the object type *node* has get and put operations for the *mbr*, a property inherited from the *spatialObject* class. The following example shows the put operation for the component *coordinate* of the class *node*.

```
PROCEDURE nodePutCoordinate (VAR o: objectType; c: ptType);
INTENT to assign coordinates to a node;
BEGIN
  o.nodef.coordinate := c;
END;
```

For each object type a separate module contains these operations. Their implementation is trivial, and the code can be generated with the knowledge of the object description. Simple consistency constraints, such as checking whether a value lies within a range, can be added to the put operations.

4.2 Level 2: Unary Object Operations

The second layer covers all database operations to store, update, delete, and access a single object. These operations are inherited from the common superclass *object type*. Their implementation is straight forward because the generic object-operations are part of the programmer's interface and can be immediately applied to each object class. Depending on the definition of the class, different access methods are supported, such as access with a key value and spatial access.

For each class, a separate module with the specific object operations is implemented. The module for the class *node*, for instance, contains the operations *nodeStore*, *nodeUpdate*, *nodeDelete*, and *nodeAccessSpatially*. The implementation of *nodeStore* is shown in the following example.

```
FUNCTION nodeStore (VAR o:  objectType; VAR err:  errorType):  boolean[16];
INTENT to store a node;
BEGIN
   nodeStore := pNew[17] (o, node[18], err);
END;
```

4.3 Level 3: Binary Object Operations

The third layer comprises aggregate operations that always envolve two objects. Standard operations are adding a part to an aggregate, removing a part from an aggregate, canceling an entire aggregate, and accessing parts of an aggregate. The operations which estabilsh links among objects require that the corresponding objects have been loaded into the database before. Reversely, remove and cancel dissolve the links without deleting the previously linked objects from the database. For each aggregation, a separate module with the aggregate operations is implemented. The following example shows the implementation of the aggregation *boundary*.

```
FUNCTION boundaryLink (VAR spatial0, bounds0: objectType;
                       VAR err:  errorType):  boolean;
INTENT to link a spatial object and a boundary;
BEGIN
  boundaryLink := pLink[19] (spatial0, bounds0, boundary[20], err);
END;
```

Three types of access operations for aggregates are distinguished: (1) iterating over all aggregate components, similar to a FOR EACH loop in CLU [Liskov 1981], (2) getting a specific aggregate component with a certain value, and (3) getting the composite object part of an aggregate. The second access method is only efficiently supported if a sorted access path was defined.

[16] The database operations are Boolean functions with side effects, being true if the operation was successfully completed, otherwise being false with an error identifying the failure.

[17] pNew is the generic object operation in the PANDA programmmer's interface to store an object.

[18] node is the class name.

[19] pLink is the generic object operation in the PANDA programmmer's interface which makes an aggregate.

[20] boundary is the type of aggregate.

4.4 Level 4+: Complex Object Operations

The fourth and later layers combine operations of the lower levels to form more complex operations. For example, connecting an edge to a node involves fulfillment of several operations: (1) a *bounds* object must be stored, (2) the edge must be linked to the bounds as boundary, and (3) the node must be linked to the bounds as coboundary. The three object operations *boundStore*, *boundaryLink*, and *coBoundaryLink* which were defined in the lower object layers, are used to implement this complex operation. The following example shows the code for the operation *edgeNodeLink*:

```
FUNCTION edgeNodeLink (VAR edge0, node0: objectType;
                       VAR err: errorType): boolean;
INTENT to link an edge with its start or end node,
       both objects must have been stored before;
VAR bound0: objectType;
BEGIN
  edgeNodeLink := false;
  IF boundStore (buond0, err) THEN
    IF boundaryLink (edge0, bound0, err) THEN
      edgeNodeLink := coBoundaryLink (node0, bound0, err);
END;
```

Note that this operation assumes that node and edge are already stored. A higher-level operation can be implemented on top of this level performing more complex consistency constraints.

The level structure is open and can be extended according to the complexity of the application. Entire applications have been written in this highly structured form. The advantage of the object layers is that very complex operations can be implemented by combining other object operations. Following the rule that no object operation may use other operations of a higher level, a well-structured application package can be designed.

5 Conclusion

The close relation between the implementation of object-oriented databases and object-oriented software techniques has been explained. For an object-oriented database it is of vital interest to tie into the software environment of the application. This paper described the implementation of the data model of the PANDA database. PANDA consists of a DBMS kernel with an object-oriented programmer's interface. The object model is based on classification, generalization, and aggregation. Inheritance is used to model the composition of complex object types.

Object types are defined as Pascal routines which are integrated into the kernel and executed at PANDA's startup. An example demonstrated the object definition. Value types,

the implementation of abstract data types (ADT), are the basic building blocks from which class types are composed. Value types can be any structured, acyclic (Pascal-) data type. This technique liberates the database administrator from redundant definitions of similar object parts. Classes can be used as the components of more complex classes. Aggregates of classes allow the implementation of part_of relations.

All classes are generalized to an *objectType* from which each class inherits the database operations. The last chapter presented how object operations are defined on top of the DBMS kernel. This layered structure is an open architecture and can be extended with the complexity of the application. The four basic layers were introduced: Level 1 contains the operations to assign and access data from objects. Level 2 is the implementation of the unary database operations for each class. Level 3 deals with aggregates and provides all binary object operations. Levels 4^+ combines these object operations to more complex ones integrating additional consistency constraints.

Conventional programming languages do not easily support the implementation of object-oriented databases and often, methods must be simulated to match the model. Using a language that supports multiple inheritance, clearer designs and more condensed implementations become possible.

6 Acknowledgement

Thanks to Bud P. Bruegger and Jeff Jackson who helped prepare this paper.

References

[Batory 1984] D.S. Batory and A.P. Buchmann. Molecular Objects, Abstract Data Types, and Data Models: A Framework. In: 10th VLDB conference, Singapore, 1984.

[Brodie 1984a] M.L. Brodie et al., editors. On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages. Springer Verlag, New York, NY, 1984.

[Brodie 1984b] M.L. Brodie. On the Development of Data Models. In: M.L. Brodie et al., editors, On Conceptual Modelling, Springer Verlag, New York, NY, 1984.

[Cardelli 1984] L. Cardelli. A Semantics of Multiple Inheritance. In: G. Kahn et al., editors, Semantics of Data Types, Springer Verlag, New York, NY, 1984.

[Chen 1976] P.S.S. Chen. The Entity-Relationship Model: Towards a Unified View of Data. ACM Transactions on Database Systems, 1(1), March 1976.