Topics in object-oriented databases and object-based representation systems

Amedeo NAPOLI

LORIA – UMR 7503

B.P. 239

F-54506 Vandœuvre-lès-Nancy

Email: napoli@loria.fr

An introduction to object-oriented databases (OODB)

- Features of the conventional database systems (DBS).
- Towards object-oriented databases systems (OODBS) — The core model for OODBS (class, instance, attribute, method, class hierarchy).
- Semantic extensions: composite objects and versions.
- SQL and queries against an OODB.
- Authorization, storage structures, transaction management.
- Object-oriented programming and DB.
- The OMG core model, OSQL, and the CORBA architecture.
- An introduction to JDBC.

Features of the conventional DBS (1)

- Conventional DBS provide DB languages to allow application programmers (or end-users) to define and manipulate the DB.
- A conventional DB language consists of 3 sublanguages:

  (1) a data definition language (DDL),

  (2) a data manipulation language (DML),

  (3) a data control language (DCL).

Features of the conventional DBS (2)

- The data definition language (DDL) allows the specification of the DB *schema*.
- In relational DB, the schema of a DB is the specification of a set of relations (the name, the domain of the relation, and integrity constraints on the domain).
- The data manipulation language (DML) includes facilities to express queries (query language, SQL) and updates (replace, insert, delete) against the DB.
- In relational DB, a query is an expression of a subset of the DB to be retrieved which satisfies user-specified search conditions.

Features of the conventional DBS (3)

- The data control language (DCL) includes facilities for protecting the integrity of the DB and for managing resources of the system.
- The integrity feature of a DCL includes *transactions* (a sequence of reads and writes against a DB, persistency and permanent writings), *integrity constraints* (conditions imposed on the contents of the DB), and *authorization* (security of data).
- The resource management feature of a DCL is the specification of the creation and the deletion of DB schema elements and access methods to the DB.

Features of the conventional DBS (4)

DBS provide various facilities such as:

- *Concurrency control* whose role is to protect the integrity of a DB in face of simultaneous accesses of the same data by multiple users.
- *Recovery* is necessary to restore the DB to a consistent state after crashes.
- *Other utilities*: error reporting, performance monitoring, application program tracing, etc.

Shortcomings of the conventional DB technology (1)

- A conventional data model, i.e. the relational model, is too simple for modeling complex nested entities, such as design and engineering objects, and complex documents (information systems on the Web).
- Conventional DBS support only a limited set of atomic data types. In particular, they do not allow the storage and retrieval of long unstructured data (long data) such as images, audio, and textual documents.
- The conventional data model does not include useful semantic concepts such as generalization and aggregations relationships (programmers must explicitely represent and manage such relationships in their programs).

Shortcomings of the conventional DB technology (2)

- Application programs are implemented in some algorithmic programming language, very different from database languages in both data model and data structures.
- The performance of conventional DBS may be unacceptable for various types of compute-intensive applications, e.g. simulation in CAD systems.
- The model of transactions supported in conventional DBS is inappropriate for long-duration transaction necessary in interactive, cooperative design environments.
- Conventional DBS do not support facilities for representing and managing the temporal dimension in DB (time, version, change notifications, etc.).

Additional features for new generation DBS

- The ability to represent and manipulate complex nested objects.
- The ability to store and retrieve arbitrarily long data.
- The ability to define and manipulate arbitrary data types.
- The ability to represent and manage changes in the DB over time (time, time interval, versions).
- The ability to represent and manipulate various semantic modeling concepts: compound documents, composite objects.
- The ability to specify rules and extended constraints to support inferencing and constraint management which are necessary as basic mechanisms for supporting knowledge-based applications.
- The ability to manage long-duration cooperative transactions.

Transition between old and new DB technologies

- The *extended relational DB technology* starts with the relational model of data and a relational query language, and extends them in various ways to allow the modeling and manipulation of additional semantic relationships and DB facilities.
- The *OODB technology* starts with an OO data model and a database language that captures it, and extends them in various ways to allow additional capabilites.
- The OO technology forms a good basis for a rich data model for the new generation DB applications such as CAD systems, knowledge-based systems, multimedia information systems, and advanced human-interface systems.

Object-oriented database systems

- A *data model* is a logical organization of the real-world objects (entities), constraints on them, and relationships among objects. An object-oriented data model is a set of object-oriented concepts for modeling data (class, instance, hierarchy, inheritance, message passing, ...).
- An *object-oriented database* is a collection of objects whose behavior and state, and the relationships are defined in accordance with an OO data model.
- An OODBS is a database system which allows the definition and manipulation of an OODB.

The OO data model – The core model (1)

- *Object* and *objects identifier*: any real-world entity is an object to which is associated a system-wide unique identifier.
- *Attributes* and *methods*: an object has one or more attributes, and one or more methods which operates on the values of attributes.

  The value of an attribute of an object is also an object.

  An attribute of an object may take on a single or a set of values.
- *Encapsulation* and *message passing*: messages are sent to an object to access the values of the attributes and the methods encapsulated in the object.

  There is no way to access an object except through the public interface specified for it.

The core model (2)

- *Class*: all objects sharing the same set of attributes and methods may be grouped into a class.

  An object belongs to only one class as an instance of that class.

  A class may be also an object, instance of a *meta-class*.

- *Class hierarchy* and *inheritance*: the classes in a system form a hierarchy (a rooted directed acyclic graph) called a *class hierarchy*.

  Any class inherit all attributes and methods of its superclasses recursively.

  An instance of a class `C` is also an instance of all superclasses of `C`.

The core model (3)

- An attribute value or a method code may be newly defined in a subclass: this operation is called *overriding*.
- In the standard interpretation of the specialization relation (and inheritance), a class `C` must be in conformance with all restrictions introduced in the superclasses of `C`.

  In this way, an instance of a class `C` is an instance of all superclasses of `C`.

The class Airplane

```
(make-class Airplane
     :superclasses (MotorizedVehicule AirborneVehicule)
     :attributes ((Capacity :domain integer)
                  (Weight :inherit-from nil)
                  (Manufacturer :domain AirplaneCompany
                                :inherit-from AirborneVehicule))
     :methods ((PresentCoordinates AirborneVehicule)) )
```

The class Person

```
(make-class Person
     :superclasses (AnimateBody)
     :attributes ((name :domain String)
                  (surname :domain String)
                  (sex :domain Sex)
                  (address :domain Address)
                  (children :domain Person)
                  (birth-date :domain Date))
     :methods ((ComputeAge)) )
```

The class-composition hierarchy

- The fact that the domain of an attribute may be an arbitrary class has a major consequence on the core model; namely, it gives rise to the *nested structure* of the definition of a class.
- A hierarchy of classes arises from the aggregation relationship between a class and its attributes and from the fact that the domain of an attribute may be an arbitrary class with its own set of attributes.
- This hierarchy, called *class-composition hierarchy*, is "orthogonal" to the concept of a class hierarchy (a class hierarchy captures the generalization relationships between one class and a set of specialized classes).

The role of a class

- The concept of a class captures the instance-of relationship between an instance and the class to which it belongs.
- A class provides the basis on which a query may be formulated.

  The notion of a class is necessary for aggregating a collection of related objects in an OODB as a target for the formulation of a query.
- The concept of a class can be used to enhance the semantic integrity of an OODB (e.g. domain specification for an attribute).
- The notion of a class conveniently factors out the specifications of the attributes and methods from all instances of the class into a single object (namely the class); this results in the saving of storage space.

Message passing and access to objects

- All operations on an object are performed by using the message interface of the object which is implemented with a method.
- A message can be sent to an object (a receiver) by using the following syntax:

  `(Selector Receiver [Arg1 Arg2 Arg3 ...])`

  The expression `(Weight Airplane-1)` will return the value of the attribute `Weight` associated with the object `Airplane-1`.

  The expression `(Weight Airplane-1 1000)` will assign the value `1000` to the attribute `Weight` of the object `Airplane-1`.
- Reading and writing values are operations performed by using message passing.

Two semantic extensions to the core model

- A *composite object* is a heterogeneous set of objects which form a part hierarchy.

  The part-of relationship form a hierarchy which is "orthogonal" to the class hierarchy (inheritance hierarchy).

- A *versioned object* is a set of objects which are versions of the same conceptual object; a versioned object consists of a hierarchy of objects which capture the version-of relationship between an object and another object derived form the object.

Composite objects

- If the domain of an attribute `A` in a class `C` is a class `D`, the value stored for the attribute `A` in an instance `x` of `C` is the object identifier(s) (oid) of instance(s) of `D`, say `y`. Then, the object `x` *references* the object `y`.
- A *weak reference* is the object reference under the core model (without the part-of semantics).
- A *composite reference* is a weak reference augmented with the part-of relationship.

Exclusive and dependent references

- An object may be part of one or more object. Then, an *exclusive composite reference* from an object `x` to another object `y` means that `y` is a part of only `x`, while a *shared composite reference* from `x` to `y` means that `y` is a part of `x` and possibly other objects.
- A *dependent composite reference* from `x` to `y` means that the existence of `y` depends on the existence of `x`; while an independent composite reference does not carry this additional semantics.
- The creation and the deletion of a composite object will trigger respectively the recursive creation and deletion of all objects referenced by the object through dependent composite references.

Versions

- Version control is one of the most important data-modeling requirements in the new generation DB applications.
- After the initial creation of an object, new versions of the object are derived from it, and new versions can in turn be derived from them, and so on.
- Versions give rise to two new types of relationship: the *derived-from* relationship between a new version of an object and an old version (of the object from which the new version was derived); the *version-of* relationship between each version of an object and an abstract object that represents the object.
- Version support incurs storage and processing overhead, and it is desirable to distinguish between *versionable* and *non-versionable* objects.

Schema modification

- OODBS systems can be (easily) extended on the basis of inheritance and specialization.
- Many users share a single DB and different users often need to *view* the DB schema differently.
- This means that the users need to modify (dynamically) the DB schema (this ability is important during the development phase).
- A *single-schema modification* is the direct modification of a single schema.
- *Schema versioning* is the versioning of a single schema, making different views of the DB visible to the users.

Single-schema modification

- One approach to single-schema modification is to define a taxonomy and a model of changes to the schema.
- The taxonomy defines a set of meaningful schema changes.
- The model provides a basis for specifying the semantics of schema changes.

  The model consists of a set of *invariants* and a set of *rules*.
- The set of invariants provide the basis for the specification of the semantics of schema changes (any schema changes must preserve all invariants).
- The rules are used by the system to select one most meaningful way in which to preserve the invariants.

Single-schema modification (continued)

- There are two types of changes to the schema of an OODB. One is to the definition of a class and the other is to the class-hierarchy structure.
- *Soft changes* are schema changes that do not require existing instances to change their classes (it avoids DB updates).
- *Hard changes* forces existing instances to change their classes (it implies DB updates).

Changes to the content of a class

- Change to an attribute of a class: (1) add a new attribute, (2) drop an existing attribute, (3) change the name of an attribute, (4) change the domain of an attribute, (5) change the default value of an attribute.
- Change to a method of a class: (1) add a new method, (2) drop an existing method, (3) change the name of a method, (4) change the code of a method.
- Change the name of a class.

Changes to the superclass/subclass relationship

- Make a class `S` a superclass of a class `C`.
- Remove a class `S` from the superclass list of a class `C`.
- Change the order of the superclasses of a class `C`.
- Add a new class.
- Drop an existing class.
- Create a new class `C` as a generalized superclass of `n` existing classes.
- Partition a class `C` into `n` new classes.
- Coalesce (fusion) `n` classes into one class.

Introduction to the notion of a query

- In relational DB, the selection operation on a relation `R`, i.e. a single-relation query on `R` with a Boolean combination of predicates on the attributes of `R`, identifies the tuples of `R` which satisfy all the predicates.
- The selection operation on a class `C` retrieves instances of the class `C` which satisfy a Boolean combination of predicates on a subgraph of the schema graph for `C`, called the *query graph*.

The notion of a schema graph

- A *schema graph* `SG` for a class `C` is a rooted directed graph consisting of a set of classes `V` and a set of arcs `E` between pair of classes; `C` is the root of `SG`.
- `E` has two types of arcs: (1) one is between two classes `C1` and `C2` such as `C2` is the domain of an attribute of `C1`, (2) another is between two classes `C1` and `D1` such that `D1` is an immediate subclass of `C1`.

The query graph

- A *query graph* `QG` for a single-operand query (a *single-operand query* retrieves objects from only one target class) on a class `C` is a connected subgraph of the schema graph `SG` for `C`, and `C` is the root of `QG`.
- `QG` includes only those nodes of the corresponding `SG` on which at least one predicate of the query is specified.

An example of query

The query graph corresponds to the following query:

find all blue vehicle manufactured by a company located in Detroit and whose president is under 50 years of age.

```
(select Vehicle (Color = "blue"
                and Manufacturer Location = "Detroit"
                and Manufacturer President Age < 50))
```

The query language
Anatomy of a query

```
SimpleQuery ::= select TargetClause from RangeClause
                       where QualificationClause
```

- The *target clause* is the specification of the attributes to be output.
- The *range clause* indicates the binding of variables, called *object variables*, to corresponding set of instances of classes.
- The *qualification clause*, also called the *value clause*, specifies the qualification conditions as a Boolean combination of predicates.

The notion of a path

- A *path* denotes a nested sequence of attributes of an object; it is a convenient way to express queries along the class-composition hierarchy.
- An object variable can be used to specify a path in a complex object (e.g. `Manufacturer Location`).
- A path specification is called a *path variable*; it is a *scalar variable* if all attributes are scalar attributes.
- A path variable may include set of attributes, and keywords such as `each` (the predicate must hold for each object), `exists` (the predicate must hold for at least one object) or `set-of` may be associated with the path.

Examples of queries

All companies which have their division located in the same city where the company headquarters are located:

```
(select Company
          where each division location string-equal location)
```

All companies which have at least one division located in the same city as the company headquarters:

```
(select Company
          where exists division location string-equal location)
```

Examples of queries (continued)

All companies whose divisions are either automobile factories or truck factories:

```
(select Company
        where division function
              is-subset '("AutomobileFactory" "TruckFactory"))


(select Company
        where each division function
                    is-in '("AutomobileFactory" "TruckFactory"))
```

Query result

- In an OODB, the result of a query is a set of objects belonging to a class or a class hierarchy rooted at some class (objects are identified by their object identifier).
- The result of a single-operand query on a class `C` may be the set of instances of the class `C` or the set of instances of the entire hierarchy rooted at `C` which satisfy the query predicates.
- If a query involves a projection operation, only the desired attributes in the instances that satisfy the predicates will be returned.

  This case can give rise to the problem of eliminating duplicates from the query result.

A note on equality of objects

- For OODB, two different criteria exist for determining the uniqueness of an instance from a collection of instances.
- One criterion is the equality of objects identifiers, i.e. *object equality* (denoted by the symbol =).
- Another criterion is the equality of the contents of the instances, i.e. *value equality* (denoted by the symbol ==).

Authorization

- In a DBS, an *authorization* is a specification of an authorization type (read, write, create) on a DB entity (DB, relation, class, attribute, ...) for a user.
- In an OODB, the notion of authorization can be extended to objects, attributes, methods and class hierarchy.
- An authorization on an attribute restricts access privileges to the value of the attribute.

  An authorization on a method may be defined for evoking and changing the method.

  An authorization on the class hierarchy restricts the possible modifications that can be done on a class or on the hierarchical organization of the class (modification of the DB schema).

Storage structures

- The predominant pattern of object-access is the direct access to single objects using the object identifiers as handles.
- Storage structures for OODBS must support efficient access to single objects given their object identifiers, and to sets of objects belonging to one or more class given queries with arbitrarily complex search conditions.
- OODBS must support storage structures to efficiently manage objects in memory once they have been fetched from the DB.
- OODBS require two types of storage structures: one for managing *persistent* objects in the DB, and one for managing objects accumulated in memory.

Structures for instances

- The `UID` is the globally unique identifier of an object.
- The `OBJECT-LENGTH` and `ATTRIBUTE-COUNT` record the total length and the number of attributes stored in the disk format.
- The `ATTRIBUTE-VECTOR` consists of the identifiers `Vi` of all attributes for which the object has explicitely specified values.
- The `VALUES-OFFSET VECTOR` consists of the offsets `Oi`, in the `VALUES` part of the object storage format, of the values of the attributes `Vi`.
- A `VALUE` can be a primitive value or a reference to another instance, namely the `UID` of the referenced object.

Note on the object identifier

- An object identifier consists of a pair `(class-id,instance-id)`, where `class-id` is the identifier of the class to which the object belongs and `instance-id` is the identifier of the instance (either within the class or within the entire DB).

- If the class identifier `class-id` of an object `X` is changed, all objects which contain references to the object `X` have to be changed; note that this operation can be very expensive.

Structures for class objects

- A class object maintains two types of information.
- One is the specification of the attributes and methods shared by all instances of the class (*instance attributes* and *instance methods*).
- Another type of information maintained in a class is the specification of the attribute and methods which apply to the class object itself (*class attribute* and *class method*).

  This last type of information can be considered as a *meta-information* on the class object.

Structures for DB schema

- The schema of an OODB can be represented as a set of system-defined classes.
- This set of classes maintains information about all classes in the DB, about attributes and methods defined for each class, about aggregation (between a class and its attributes) and generalization relationships (between classes).
- The meta-information about classes, attributes and methods is defined in the three classes `Class`, `Attribute` and `Method`.

Disk and page layout

- The disk and page layout is the standard layout used in conventional DBS.
- A raw disk may be divided into a set of partitions.
- Each partition consists of a number of segments; and each segment in turn consists of a number of blocks or pages.

Using a class-hierarchy index

Find all blue vehicle manufactured by Renault Company:

```
(select Vehicle
        where Color = "blue"
          and Manufacturer Name = "Renault Company")
```

- All instances of the class `Company` are identified which have the string `"Renault Company"` in the `Name` attribute.

  A class-hierarchy index on the attribute `Name` can be used to expedite the search on the class `Company`.

- The `UID` of these instances are then looked up in the `Manufacturer` attribute of the class `Vehicle`.

- The result of the query is the set of instances of `Vehicle` which have the string `"blue"` in the `Color` attribute and which contain in the `manufacturer` attribute a `UID` that is in the list of `UID`s for `Renault Company` instances.

Transaction management

- A *transaction* is a sequence of reads and writes against a DB.

  A transaction has two properties: atomicity and seriability.

- The *atomicity* property means that the sequence of reads and writes is regarded as a single atomic action against the DB.

  It ensures that if a transaction cannot complete, the system backs out all writes which may have been recorded, else the system guarantees that all writes are recorded in the DB.

- The *seriability* property means that the effect of concurrent execution of more than one transaction is the same as that of executing the same set of transactions one at a time.

Transaction management (continued)

- The objective of *transaction management* is to ensure the atomicity and seriability properties of transactions.
- Transaction management consists of two components: concurrency control and recovery.
- *Concurrency control* refers to automatic control of simultaneous accesses to a common part of the DB by more than one transaction.
- *Recovery* refers to the ability to restore the DB to a state which existed at some point before the failure of any transaction.
- Most systems, including OODBS, use a locking protocol to implement concurrency control, and a logging protocol (memorization of sequences of primitive actions) to implement recovery.

Integrating object-oriented programming and DB

- The programming language must be extended with DB primitives: it needs to be extended with constructs to express queries, integrity specification (including transactions), access authorization, access methods, versions, etc.
- The DBS must be designed and implemented to support the requirements of the programming language: fetch page frames from the disk into page-buffer,copy objects from the page-buffer-pool, send objects to the workspace application, etc.
- The mismatch between data models and data structures: a mapping between the DB format (e.g. storing format) and the programming language format (e.g. manipulation in-memory) is necessary.

Architecture

- A network-based DBS in which the entire DB is managed by one node (processor) is called a *client/server DBS*: a dedicated server manages the entire DB and all other nodes (called *clients*) communicate with the server to access the DB.
- A network-based DBS in which the DB is managed by more than one node is called a *distributed DBS*.
- Mots of operational DBS are client/server systems.

The subsystems of a client/server architecture

- The message handler receives all message sent to the DBS.
- The object subsystem provides high-level data management functions, including query optimization, schema management, long data management (including text search), and support for versionable objects, composite objects, and multimedia objects.
- The transaction management subsystem coordinates concurrent object accesses and provides recovery capabilities, using locking and logging techniques.
- The storage subsystem manages persistent storage of objects and controls the flow of objects between the secondary storage device and main buffer memory.
- The communication subsystem is responsible for opening, closing, and controlling connections; receiving and delivering messages between client machines and the server machine.

A brief list of OODBS

- ONTOS (associated programming language: C++).
- ObjectStore (C++).
- O2 (CO2, C++ and Common Lisp).
- Gemstone (OPAL, a dialect of Smalltalk).
- ORION and ITASCA (Common Lisp).
- Objectivity/DB (C++).
- VERSANT (C++ and Smalltalk).
- POET (C++).
- ...

The OMG object model
The Object Management Group (OMG)

- The OMG is an international software industry consortium (about 350 companies) with two primary aims :

  (1) Promotion of the OO approach to software engineering in general.

  (2) The development of command models and a common interface for the development and use of large-scale distributed applications (open distributed processing) using OO technology.

- OMG is developing standards, and these standards are and will be based on existing commercially available products (as far as possible).

The object management architecture

- The OMG technology is performed around the general reference architecture called the object-management architecture (OMA).
- An application that is OMA compliant consists of a set of inter-working classes and instances that interact via the Object Request Broker (ORB), a distributed "software bus" that enables objects to make and receive requests and responses.

The object management architecture

- Every application in a distributed system is modeled as an object whose categories are as follows.
- Objects services (OS) objects are collections of services with object interfaces that provide basic functions for realizing and maintaining objects of any categories.
- Common facilities (CF) objects are collections of classes and objects that provide general-purpose capabilities useful in many applications.
- Application objects (AO) are specific to particular end-user applications.

The current work of OMG

The current work of OMG is along four major lines:

- Definition of object request broker (ORB) and extensions.
- The definition of objects services and common facilities.
- The definition and extension of a common core object model.
- A consensus building among subgroups of OMG membership in such areas as analysis and design methodologies (such as UML, *Unified modeling language*).

The OMG core model: topics

- Basic concepts.
- Object identity and object identifiers.
- Object types (and non object types).
- Operations.
- Subtyping and inheritance.
- Argument passing and results.
- Interface of a type.
- Implementation.

Object types

- Objects support only certain operations and the operations applicable to an object collectively characterize its behavior.
- *Types* describe theses operations and thus characterize the behavior of objects.
- Each operation has a signature and the set of operation signatures is the *interface of the type.*
- A type is distinct from its set of instances (extension of the type).

  Both the interface and the extension of the type can change over time without changing the identity of the type.

Subtyping and inheritance

- Subtyping is a relationship between types based on their interfaces.

  It defines the rule by which objects of one type are determined to be acceptable in contexts expecting another type.

  if `T1` is a subtype of `T2`, then any object of `T1` cab be used in any context that expects an object of `T2` (objects of type `T1` are also objects of type `T2`).

- Inheritance is a mechanism for reuse.

  It allows a type to be defined in terms of another type: with the notation above, the operations defined for for `T2` are also defined for `T1` 9and cab be used by `T1`).

- Subtyping is a relation between interfaces and inheritance apply both to interfaces and implementations.

- Subtyping and inheritance are related in the core model: if `T1` is a subtype of `T2`, then `T1` inherits form `T2`.

The object model for CORBA

- The basic service provided by CORBA (*Common request object broker architecture*) is delivery of messages (requests) from one piece of code to another and delivery of response (return or exception) to the caller.
- The sender is often called the *client* and the receiver the *server*.

The architectural parts of CORBA

- A single specification language IDL (*interface definition language*) is used to specify interface independent of programming language.
- A fully dynamic representation of available interfaces in an *interface repository* (IR) representing the interfaces of all objects available in the distributed computing system.
- A fully dynamic calling extension allowing run-time discovery of objects, discovery of available interface from an IR, construction of message requests, and sending of those requests.
- A *context* extension allowing passing of optional named parameters, for explicit control over execution.
- An abstraction mechanism (*object adaptator*) for removing the details of object implementation from the message substrate.

Object SQL

- Object SQL (OSQL) is a database (programming) language that combines an expression-oriented language with a high-level, declarative, and optimizable query language.
- OSQL provides many of the advantages of object orientation, including more intuitive model, code reuse and extensibility, together with all the features of the current DB technology such as query optimization, integrity constraints, multiuser access, authorization, and security.

Object SQL

The design goals for OSQL are that it should:

- be based on a simple orthogonal OO model and type system,
- be computationally complete and independent of specific application programming languages,
- provide constructs for specifying declarative queries and allow such queries to be compiled and optimized,
- be extensible, i.e. allow the user to (dynamically) define new types and operations,
- create no artificial distinctions between meta-data objects and user-defined objects,
- allow separate definition of the interface of an object and the corresponding implementation.

JAVA and OODBS: An Introduction to JDBC

- The *Java database Connection* (JDBC) is a set of Java classes that provide an interface to DB.
- JDBC can be used a Java program to access almost every kind of DB.
- JDBC provides two major sets of interfaces:

  (1) an *application interface* that allows to access DB services in a DBS-independent manner,

  (2) a *driver interface* that DBS vendors must adapt to their particular DB.

JDBC

- JDBC is not a query language; it is simply a Java interface to SQL.
- JDBC neither adds nor subtracts from the power of SQL; it is simply a mechanism for submitting SQL statements.
- JDBC handles details such as connecting to a DB, fetching query results, committing or rolling back transactions, and converting SQL types to and from Java programs variables.

JDBC and CLI

- JDBC is a native Java implementation of the SQL CLI (CLI stands for *Call Level Interface*); it uses Java classes and interfaces to represent the basic abstractions and concepts defined in the SQL CLI, including drivers, connections, SQL statements, transactions, etc.
- The CLI requires the use of intelligent DB drivers that accept aCLI call and translate it into the native DB server's access language.

  With the proper driver, any data source application can function as a CLI-server and can be accessed by the front-end tools and client programs that use the CLI.
- The CLI requires a driver for each DB to which it connects.

  Each driver must be written for a specific server using the server's access methods.
- CLI is not a new query language; it is simply a procedural interface to SQL.

The architecture of JDBC

- JDBC allows the user to write DBS-independent Java code.
- JDBC provides two major sets of interfaces:

  (1) an *application interface* that let the user access SQL services in a DBS-independent manner,

  (2) a *driver interface* that DBS vendors must adapt to their particular DB.
- JDBC drivers must provide implementations of the abstract classes and interfaces defined in the `java.sql` package.

  A JDBC driver must implement the core JAVA.SQL interfaces and classes.

Bibliography (elements)

- M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik, *The Object-Oriented Database System Manifesto*, in Deductive and Object-Oriented Databases. Proceedings of the First International Conference (DOOD'89), edited by W. Kim, J.-M. Nicolas, and S. Nishio, North-Holland, Amsterdam, Netherlands, page 223-240, 1990.

- *Object-Oriented Concepts, Databases and Applications*, edited by W. Kim and F. Lochovsky, Addison-Wesley, Reading MA, 1989.

- W. Kim, *Introduction to Object-Oriented Databases*, The MITP Press, Cambridge, MA, 1990.

- *Modern Database Systems – The Object Model, Interoperability, and Beyond*, edited by W. Kim, ACM Press, New York and

Addison-Wesley Publishing Company, Reading, MA, 1995.

- R.G.G. Cattell, *Object Data Management*, Addison-Wesley, Reading, MA, 1994.
- R.G.G. Cattell, *The Object Database Standard – ODMG 2.0*, Morgan Kaufmann Publishers, San Francisco, CA, 1997.
- O. Deux et al., *The Story of O_2*, ACM Transactions on Transactions on Knowledge and Data Engineering, 2(1):91-108, 1990.
- R. Orfali and D. Harkey, *Client/Server Programming with JAVA and CORBA*, John Wiley & Sons, New York, 1997.
- R. Orfali and D. Harkey and J. Edwards, (1) *The Essential Client/Server Survival Guide (Second Edition)*, and (2) *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, 1996.
- ...

Object-based representation systems

- What are the needs for object-based representation systems? The duality programming vs. representation.
- A model of object-based representation systems (OBRS) and its variations.
- Reasoning services: the classification process.
- Resyn, Troeps, and description logics.
- Using OBRS and OODBS for data mining (knowledge discovery) in databases.

The duality programming vs. representation

- Declarative – procedural controversy: knowledge representation is not (only) programming.
- The logical approach: adequation between syntax and semantics (correctness and completeness of OBRS).
- Knowledge Representation using Objects = class definition + class organization + class instantiation + reasoning.

  Classification of classes = constructing a hierarchy of classes.

  Classification of instances = reasoning upon the hierarchy (for identification, query, etc.). classes.

What is a hierarchy? $\mathcal{H} = (\mathcal{X}, \preceq, \omega)$

What is a class? A structure with attributes (and methods).

An example of a knowledge base

A **Man** is a **Human**.

A **Woman** is a **Human**.

No **Man** is a **Woman**, and *vice versa.*

A **Team** is (defined as) a **Set** with at least 2 **members** which are all **Humans**.

A **Small-Team** is (defined as) a **Team** with at most 5 **members**.

A **Modern-Team** is (defined as) a **Team** with at most 2 **members**, with at least 1 **leader**, which is a **member**, and all **leaders** are **Women**.

Towards a model of OBRS

- **Knowledge Description**
  $\rightarrow$ class = data + behavior,
  $\rightarrow$ objects = class instances.
- **Class organization**
  $\rightarrow$ specialization + inheritance,
  $\rightarrow$ property sharing,
  $\rightarrow$ subsumption, composition and viewpoints.
- **Programming**
  $\rightarrow$ message sending,
  $\rightarrow$ access-oriented programming.
- **Reasoning**
  $\rightarrow$ inheritance-based inferences,
  $\rightarrow$ matching and classification,
  $\rightarrow$ constraints and truth maintenance.

Variation on specialization

Composition and composite objects

- A composite class is defined through a set of composing classes –the components– (a component can be a composite class).
- The relations between a composite class and its components can be represented by a composition graph whose vertices denote components and edges denote composition links (components–components and components–composite links).
- The composition relation is a complex relation that may be interpreted in several ways.

Variation on instantiation
Multi-instantiation and viewpoints

- From an object-oriented point of view, an object may be an instance of only one class.
- Multi-instantiation is used in OODBS and in OBRS in order to take into account different points of view on an individual (that may be represented by several instances).

Variation on specialization: subsumption

- A class `C` subsumes a class D iff the extension of `C` –the set of instances of `C`– necessarily includes the extension of `D`.

  The extension of a class is considered as the *interpretation* given to the class.

- Class extensions are nested: the extension of more general classes include the extension of most specific classes.

- The inclusion of extensions implies *monotony* and *overriding* is not allowed: if `C` subsumes `D`, then an instance `i` of `D` is necessarily an instance of `C`, i.e. `i` verifies the constraints defined by the class `C`.

Reasoning services in an OBRS

- *Subsumption*: is a class `C` more general than a class `D`? (**Notation**: $\mathtt{D} \sqsubseteq \mathtt{C}$).
- *Classification*: where is situated a class in a class hierarchy?
- *Satisfiability*: is a class `C` defined in a meaningful way or does it denote the *empty class*?
- *Consistency*: is the represented knowledge consistent?
- *Entailment*: what facts are deducible from the knowledge?
- *Instantiation (test)*: which are the classes an individual —an *object*— is instance of?
- *Retrieval*: which are the instances of a given class?

Classification-based reasoning

- *Classification*: where exactly is a class situated in the class hierarchy?
- The classification process of an object `X` –a class or an instance– relies on three main steps:
  - `(i)` Search for the most specific subsumers of `X` (MSS),
  - `(ii)` Search for the most general subsumees of `X` (MGS),
  - `(iii)` insertion of `X` in the class hierarchy, under its MSS and over its MGS.

An algorithm for classification

```
function subsumers(C* X)
   if C* is marked
     then consider the next class
     else mark C*
          if X ⋢ C*
             then mark all descendants of C*
                  consider the next class
             else if C* has no descendant (C* is a leaf)
                     then return {C*}
                     else SPS ← {}
                          for every descendant D* of C* do
                              SPS ← SPS ∪ subsumers(D* X)
                          if SPS = {}
                             then return {C*}
                             else return SPS
```

Principles of description logics (1)

- Description logics are based on concepts and role description languages.
- Concepts describes classes of individuals while roles model relations between classes.
- Concepts and roles are syntactic structures to which a semantics is associated (using an interpretation function).
- Concepts (and roles) are organized in hierarchies using the subsumption relation.

Example

```
HUMAN :< TOP
SET :< TOP
MAN :< HUMAN
WOMAN :< (and HUMAN (not MAN))

member :< anyrole
leader :< member

TEAM := (and SET (all member HUMAN)
                 (atleast 2 member))

SMALL-TEAM := (and TEAM (atmost 5 member))

MODERN-TEAM := (and TEAM (atmost 4 member)
                         (atleast 1 leader)
                         (all leader WOMAN))
```

Principles of description logics (2)

- A set of *assertions* describes the facts in which individuals are implied (assertions describes the data associated with a particular problem).
- Two kinds of assertions: a concept instantiation: `(C o)` and a role instantiation: `(R o o')`.
- The classification and the instantiation test processes are the basis of *terminological reasoning*.
- An example of instantiation test: if `(D o)` is true and if $\mathtt{D} \sqsubseteq \mathtt{C}$, then `(C o)` is true.
- DLs are object-based and theoretical studies on DLs have been used for the study of OBRS.

OBRS meet OODBS for data mining

- The goal of *data mining* is to discover *patterns* in data that can be interpreted as *units* of knowledge in an OBRS.
- **Object-based representation of domain knowledge**:

  Units: classes, attributes, relations, instances.

  Procedural representation: methods.
- **Manipulation of data**:

  Data are translated and represented as instances of classes representing domain knowledge.

  Data are manipulated as individual objects by reasoning and exploration processes.

Elements for object-oriented data mining

- OBRS – OODBS interface and service delegation: what are the services that must be performed respectively by the OBRS and by the OODBS?

OBRS	OODBS	Mining techniques
Representation of	Data storage	Machine learning techniques
domain knowledge	Data management	Classification, clustering
Reasoning processes	(queries, views, updates,	(symbolic, numerical, neuronal)
...	integrity checking, ...)	Segmentation techniques
	...	(decision trees, graphs, lists...)
		...

Discussion on the connections between OBRS and OODBS

- Conceptual model of the studied domain:

  no realistic data mining process can be performed without a *model* of the studied domain.

- Data validation for incomplete or corrupted data.
- Multiple viewpoint organization of objects and data (multi-categorization).
- Object-based queries and result reification: intension vs. extension.
- Classification and clustering to discover correlations between data.
- Problems: bridges between OBRS and OODBS.

Bibliography (elements)

- *Langages et modèles à objets — État des recherches et perspectives*, editeb by R. Ducournau, J. Euzenat, G. Masini and A. Napoli, INRIA, Collection Didactique D-019, Le Chesnay, 1998 (in French).
- G. Masini, A. Napoli, D. Colnet, D. Léonard and K. Tombre, *Object-Oriented Languages*, Academic Press, London, 1991.
- B. Nebel, *Reasoning and Revision in Hybrid Representation Systems*, Lecture Notes in Computer Science 422, Springer, Berlin, 1990.
- *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, edited by J. Sowa, Morgan Kaufmann, San Mateo, CA, 1991.
- P.D. Karp, *The Design Space of Frame Knowledge*

Representation Systems, Technical Note 520, SRI International, Menlo Park, 1993.

- *Advances in Knowledge Discovery and Data Mining*, edited by U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, AAAI Press / MIT Press, Menlo Park, CA, 1996.
- A. Borgida, *Description Logics in Data Management*, ACM Transactions on Transactions on Knowledge and Data Engineering, 7(5):671-682, 1995.
- ...