These slides are for use with

Database Systems

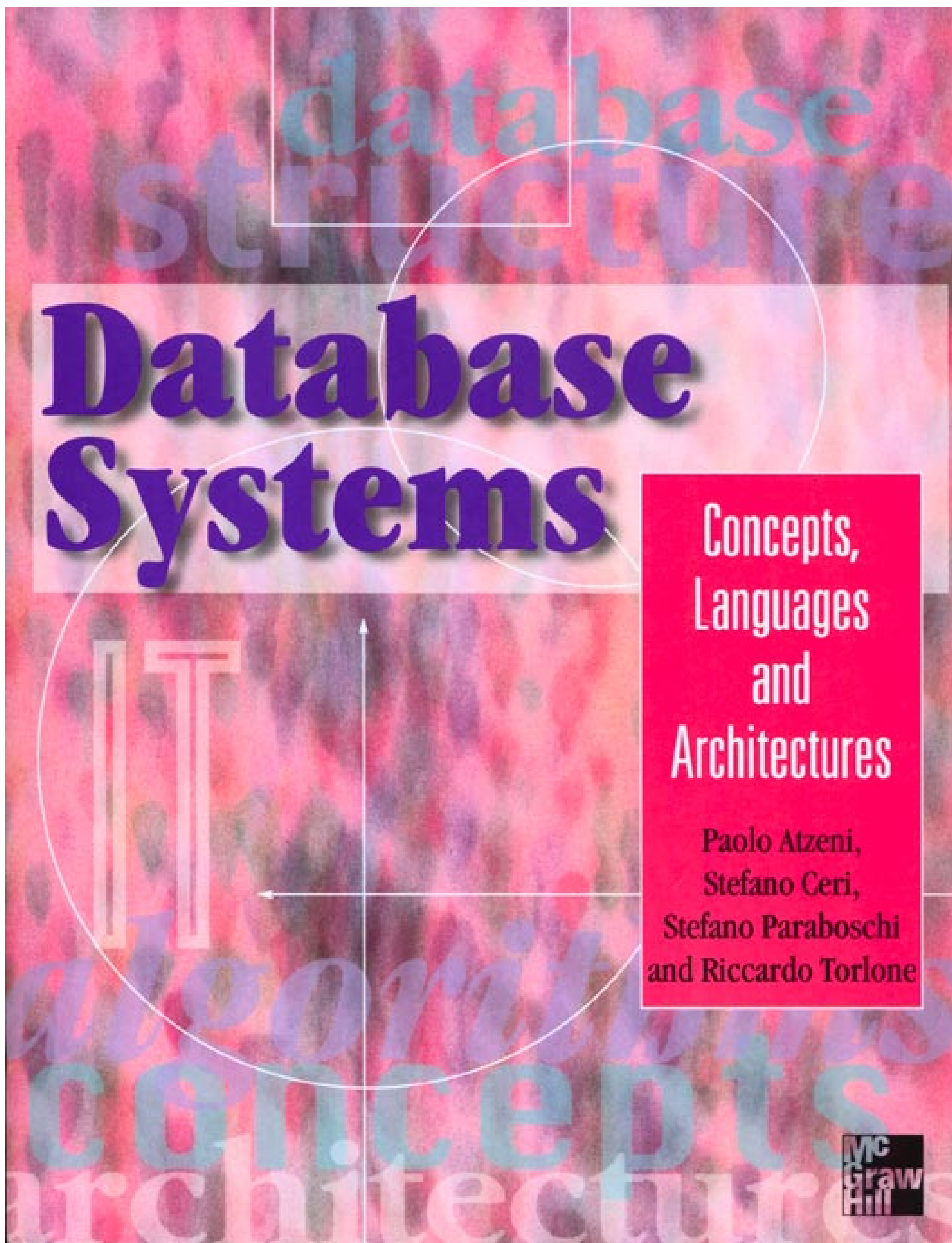
Concepts, Languages and Architectures

Paolo Atzeni • Stefano Ceri • Stefano Paraboschi • Riccardo Torlone
© McGraw-Hill 1999

To view these slides on-screen or with a projector use the arrow keys to move to the next or previous slide. The return or enter key will also take you to the next slide. Note you can press the 'escape' key to reveal the menu bar and then use the standard Acrobat controls — including the magnifying glass to zoom in on details.

To print these slides on acetates for projection use the escape key to reveal the menu and choose 'print' from the 'file' menu. If the slides are too large for your printer then select 'shrink to fit' in the print dialogue box.

Press the 'return' or 'enter' key to continue . . .



Chapter 11

Object databases

Click here for help

McGraw-Hill

Summary

- Motivations
- Object Oriented Databases (OODB)
- Object-relational Databases (ORDB)
- Technology of Object Databases

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Technology of relational DBMSs

- Relational Database Management Systems (RDBMS) permitted an efficient implementation of business applications, offering
  - persistency, sharing, reliability
  - data with a simple structure, for numerical/symbolic information
  - concurrent transactions of brief duration (OLTP)
  - complex queries, expressed by declarative languages with “associative” access
- The technical evolution (improved performance, capacity, and hardware costs) generated new applicative needs — where relational technology demonstrated its inadequacy

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Emerging applicative areas

- Technical applications
  - CASE (Computer-Aided Software Engineering)
  - CAD (Computer-Aided Design)
  - CAM (Computer-Aided Manufacturing)
- Document management
  - documents and office automation
  - hyper-textual data
  - multimedia
- Other
  - scientific and medical applications
  - expert systems, knowledge representation

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Features of new applicative areas

- All these applications need services traditionally offered by DBMSs, like persistency, sharing, and reliability, but also
  - data with a complex structure
    - non-numerical information — images, spatial data, temporal series, ...
    - pre-defined and user-defined types (with reuse)
    - explicit ("semantic") relationships on data (references), complex aggregations
  - complex operations
    - customized on the type of information — e.g., multimedia
    - associated also with user-defined types
    - long-lived transactions

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Object databases

- Some of the above features suggest the introduction into database management systems of concepts typical of the object-oriented paradigm
- Since the mid-'80s, several object DBMSs (ODBMS) have been implemented — research prototypes and commercial products
- Systems have been developed independently from each other, with no standardization on the data model and on the query languages
- After an initial period, a convergence on a common model and language is arising, to permit the interoperability among systems by different producers

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

ODBMS Technology

- The first generation of ODBMS is represented by *persistent* object programming languages, which offer only some of the services of DBMSs, with no support for queries, in a way incompatible with RDBMSs
- The ODBMS of the second generation offer a greater number of DBMS services, and typically offer support for queries
- Two main ODBMS solutions
  - OODBMS (*Object-Oriented*): a *revolutionary* solution, with respect to RDBMS
  - ORDBMS (*Object-Relational*): an evolutionary solution, with respect to RDBMS

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

An object data model

- An object database is a *collection* of *objects*
- Each object has an identifier, a state and a behavior
  - the *object identifier* (OID) characterizes uniquely the object, and permits to build references among objects
  - the *state* is the collection of values associated with the object *properties* — it is generally a value with a complex structure
  - the *behavior* is described by the methods that can be invoked on the object

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Types — static part

- A type describes the object *properties* (static part) and the interface of the *methods* (dynamic part)
- For the static part, types are built on the basis of a set of *atomic types*
  - numbers
  - string
  - OID
  - Boolean
  - enumeration types
  - ...
- To represent the null value, the polymorphic `nil` value is used
- Each type definition associates a name (label) to a type

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Complex data types

- *Type constructors* allow the definition of complex data types
- Typical constructors:
  - *record-of*(A_1:T_1, ..., A_n:T_n)
  - *set-of*(T)
  - *bag-of*(T)
  - *list-of*(T)
- The constructors are orthogonal
- Given a complex data type *T*, an object having as type *T* is an *instance of T*
- Some systems offer a limited choice of constructors

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Example of complex data type

```
Automobile: record-of(
   RegistrationNumber: string,
   Model: string,
   Manufacturer: record-of(
      Name: string,
      President: string,
      Factories: set-of(
         record-of(
            Name: string,
            City: string,
            NoOfEmployees: integer))),
   Colour: string,
   Price: integer,
   MechanicalParts: record-of(
      Motor: string,
      ShockAbsorber: string))
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Example of complex value

- It is possible to define complex values *compatible* with a complex data type

```
V1: ["MI67T891", "Uno",
     ["Fiat", "Agnelli",
      {["Mirafiori", "Torino", 10000],
       ["Trattori", "Modena", 1000]}],
     "blue", 7000,
     ["1100CV", "Monroe"]]
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Objects and values

- The use of complex types and values permits to associate with a single object an arbitrary structure
- On the contrary, in the relational model concepts have often to be represented by multiple relations
- Alas, the representation proposed for `Automobile` in the previous example is not "normalized": we can decompose it using *object references*
- An object is made up of a pair (*OID, Value*), where *OID* (*object identifier*) is a system defined atomic value, generally not visible to the user, and *Value* is a complex value
- The value of an object property can be the OID of another object (implementing a reference)

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Object references

```
Automobile: record-of(
   RegistrationNumber: string,
   Model: string,
   Manufacturer: *Manufacturer,
   Colour: string,
   Price: integer,
   MechanicalParts: record-of(
      Motor: string,
      ShockAbsorber: string))

Manufacturer: record-of(
   Name: string,
   President: string,
   Factories: set-of(*Factory))

Factory: record-of(
   Name: string,
   City: string,
   NoOfEmployees: integer)
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Object references

- A set of objects compatible with the schema

```
O1: <OID1, ["MI67T891", "Uno", OID2,"blue", 7000,
            ["1100CV", "Monroe"]]

O2: <OID2, ["Fiat", "Agnelli", {OID3, OID4}]>

O3: <OID3, ["Mirafiori", "Turin", 10000]>

O4: <OID4, ["Trattori", "Modena", 1000]>
```

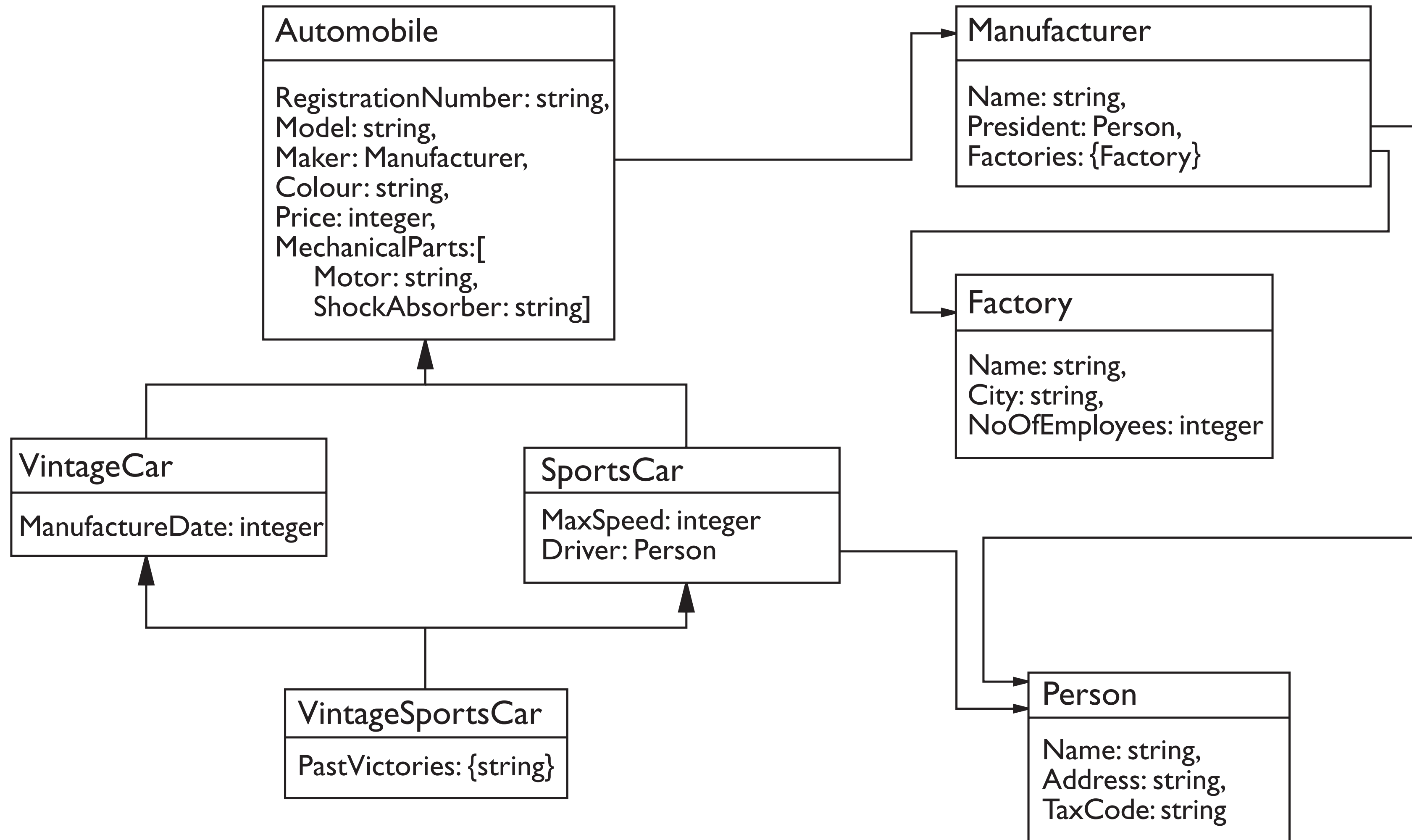
© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Identity and equality

- Two object may have the same state, but differ in their identifier
  - *identity* (O1=O2) — the objects have the same identifier
  - *superficial equality* (O1==O2) — the objects have the same state, i.e., the same value for the corresponding properties
  - *deep equality* (O1===O2) — the objects have the same “reachable” values, as they are obtained navigating the object references (it does not require superficial equality)
    - O1 = <OID1, [a, 10, OID3]>
    - O2 = <OID2, [a, 10, OID4]>
    - O3 = <OID3, [a,b]>
    - O4 = <OID4, [a,b]>

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Database schema for the description of automobiles



© McGraw-Hill 1999

Semantics of references

- The concept of reference has analogies with *pointers* in programming languages and with *foreign keys in* relational systems. However
  - pointers may become corrupt (dangling); references to objects (in a good ODBMS)are automatically made invalid when a referenced object is deleted
  - foreign keys are visible, since they are realized with values; object identifiers are not associated with user viewable values
  - an update on the attributes of a foreign key may cause the loss of the reference; an update on a referenced object maintains the object reference

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Classes

- Objects are associated with a type (*intension*) and a class (*implementation*)
  - the *type* is an abstraction that describes (1) the state and (2) the behaviour of an object
  - the *class* describes the implementation of a type — data structure and implementation of methods in a programming language
- A class definition is composed of two parts
  - The interface describes the type of the objects belonging to the class and the signature of each method
  - The implementation describes the code of the methods and sometimes the physical structures used for the storage of objects

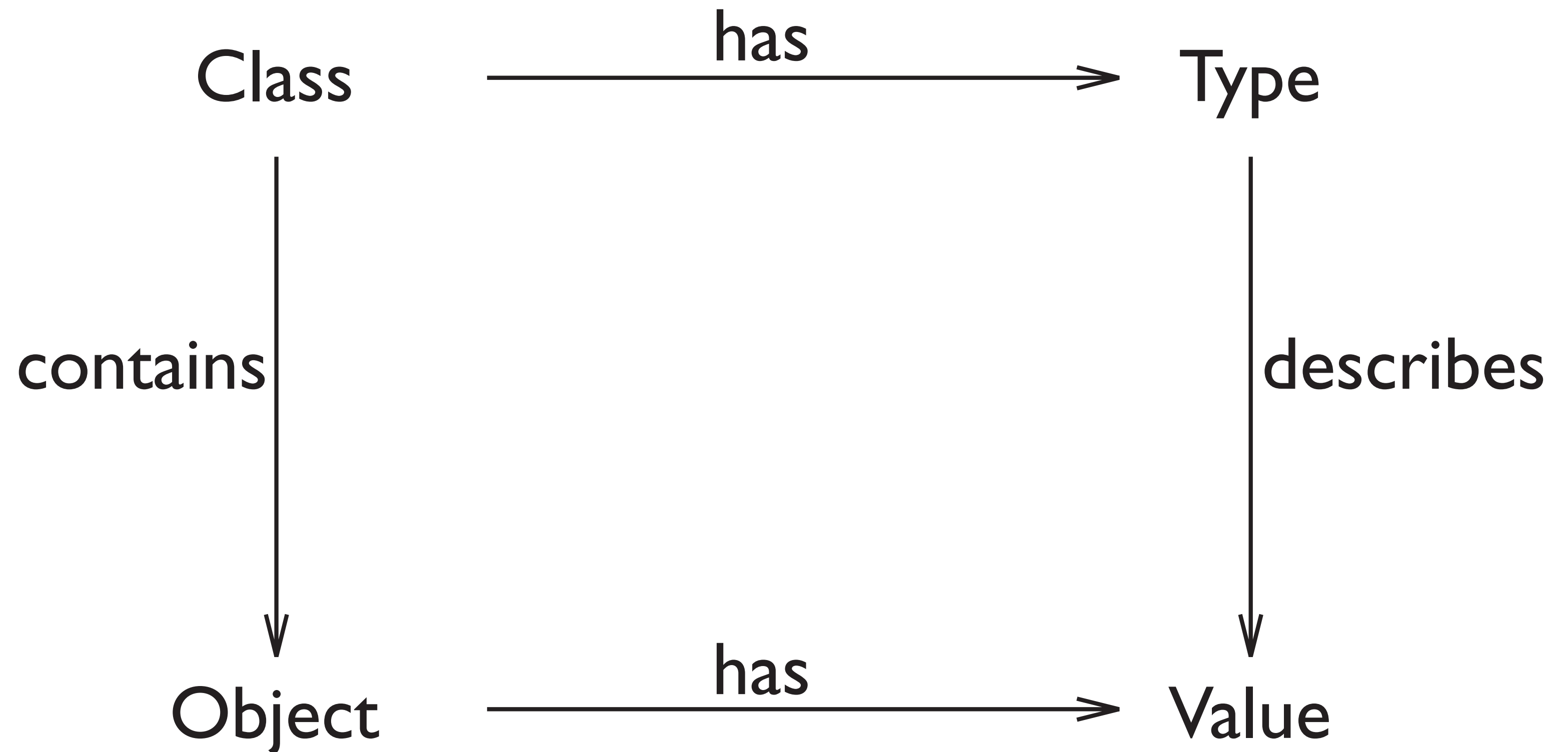
© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

An object data model

- Objects are grouped in collections (*extensions*)
- We make the following simplifying assumptions
  - the concept of class describes both the implementation and the extension of a type
  - types are an abstraction that describes the state and the behaviour
- In our model, then:
  - types and classes are distinct
  - every class is associated with a unique type
  - extensions are not described by an explicit concept

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Relationship between values, objects, types and classes



© McGraw-Hill 1999

Considerations on classes

- We assume that all the objects of a class are of the same type
- Objects may be dynamically added and removed from a class (typically using constructor and destructor methods)
- Since all the objects of a class are associated with the same type, they all have the same properties and accept the same method invocations
- Concepts of class and extension are distinct
- A *class* is an implementation of a type — a type can have several implementations. This is not to be used to associate different semantics to different extension (absolutely not!), but it can be used to implement a (unique) semantics on different architectural platforms

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Example of class definition in O2

```
add class Automobile
  type tuple(
    RegistrationNumber: string,
    Model: string,
    Maker: Manufacturer,
    Colour: string,
    Price: integer,
    MechanicalParts: tuple(
      Motor: string,
      ShockAbsorber: string))

add class Manufacturer
  type tuple(
    Name: string,
    President: Person,
    Factories: set(Factory))
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Relationships

- References and collections of references permit the representation of *relationships* (*binary*) among two types A and B
  - a one-to-one relationship can be represented by a “reference” attribute in each type
  - a one-to-many relationship can be represented by a reference attribute from A to B, and a set of references from B to A
  - a many-to-many relationship can be represented by a set of references for each type
- This solution introduces two properties, one for each class, which are not independent: each one is the inverse of the other
- Some object models offer an automatic management of relationships

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Methods — dynamic part

- The object-oriented paradigm derives from the concept of *abstract data type*
- A *method* is a procedure used to encapsulate the state of an object and is characterized by an *interface* (or *signature*) and an *implementation*
  - the interface provides all the information needed to invoke a method (the number and type of the input and output parameters)
  - the implementation contains the code of the method
- The *type* of an object describes, together with the properties, the interfaces of the methods applicable on objects of that type
- We make the hypothesis that methods are like *functions*, i.e., they can have several input parameters, but at most one output parameter

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Methods

- Succinctly, methods can be of the following categories
  - *constructors* — to build objects using for the object properties the values of the input parameters, returning the OID of the built object
  - *destructors* — to remove objects, and possibly other objects reachable from them
  - *accessors* — functions that return information on the object content (derived properties)
  - *transformers* — procedures that modify the state of the object, and possibly of other objects reachable from them
- A method can be *public* or *private*

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Methods

```
add method init(                             // constructor
    RegNumber_par: string,
    Model_par: string,
    Colour_par: string,
    Price_par: integer): Automobile
  in class Automobile is public

add method Price(): integer                  // accessor
  in class Automobile is public

add method Increase(                         // transformer
    Amount: integer)          // returns void
in class Automobile is public
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Implementation and method invocation

```
body init(RegNumber_par: string,
          Model_par: string,
          Colour_par: string,
          Price_par: integer):
          Automobile in class Automobile
  co2{self -> RegistrationNumber = RegNumber_par;
      self -> Model = Model_par;
      self -> Colour = Colour_par;
      self -> Price = Price_par;
      return(self); }$

execute co2 {         // in a program
  o2 Automobile X;    // variable declaration
  X = new(Automobile);
  [X init("MI56T778","Panda","blue",12000)]; }$
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Implementation and method invocation

```
body Increase(Amount: integer)
  in class Automobile
  co2{ self -> Price += Amount;}$

execute co2 {
  o2 Automobile X;
  X = new(Automobile);
  [[X init("MI56T778", "Panda", "blue", 12000)]
   Increase(2500)];}$
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Impedance mismatch

- In a RDBMS there is an impedance mismatch between the languages used to write applications (which manipulate scalar objects) and SQL (which extracts sets of tuples); the problem in a RDBMS is solved with *cursors*
- ODBMS are said to solve this problem, because persistent objects can be manipulated directly by the instructions of the programming language (procedural)
- The programming language of an ODBMS must permit the access to the components of a complex value — e.g., records with the dot operator ".", references with operator "->", collections with ad hoc iterators

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Criteria for designing methods

- It is important to exploit the potential for reuse that the object-oriented approach offers
- The OMT (Object Modeling Technique) methodology suggests the following guidelines to method definition
  - conciseness
  - coherence (only one service per method) and consistency (common style among methods)
  - separation between policy and implementation
  - anticipation of future requirements
  - independence (e.g., no variables shared between methods)
  - use of inheritance

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

McGraw Hill

Generalization hierarchies

- Among the types (and classes) of an object database, it is possible to define a generalization hierarchy
- A sub-class inherits the properties and methods of the sub-class and can, additionally, make its state and methods more specific
- All the objects of a sub-class automatically belong to the super-classes
- Hierarchies have the transitive property: if $C1$ is a sub-class of $C2$ and $C2$ is a sub-class of $C3$, $C1$ is a sub-class of $C3$

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Example of hierarchy

```
add class SportsCar
  inherits Automobile
  type tuple(MaxSpeed: integer,
             Driver: Person)

add class VintageCar
  inherits Automobile
  type tuple(ManufactureDate: integer)

execute co2 {
  o2 VintageCar X;
  X = new(VintageCar);
  [X init("MI56543", "Ferrari", "red", 300000)];
  X -> ManufactureDate = 1957; }$
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Considerations on hierarchies

- The role of generalization hierarchies in object databases is the same of object-oriented programming languages
- There is an important difference: objects of a programming language are short-lived, whereas the objects of a database are persistent, with non-trivial consequences

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Migration between classes

- During its life an object must keep its identity, so it is possible to reference it in a unique way
- But the object can *change its type*:
  - a person becomes a student, then a worker-student, then a worker, then a married person, ...
- It seems necessary to have a mechanism that permits to migrate from one level of the hierarchy to another, using operations of
  - specialization (the object becomes a member of a sub-class)
  - generalization (the object loses the membership of a sub-class)
- Some systems constrain an object to belong only to one most specialized class, others do not set this constraint

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Multiple inheritance

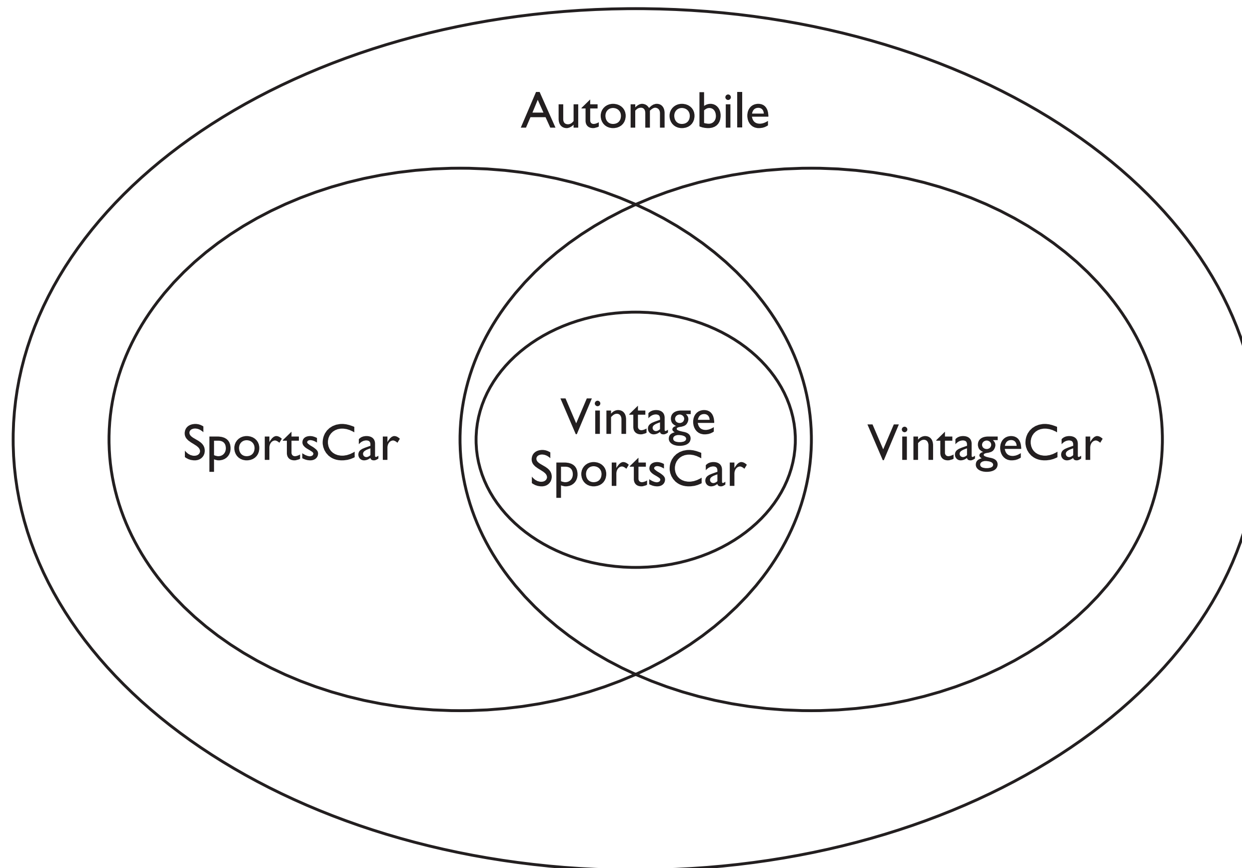
- In some systems it is possible for a class to inherit from several super-classes
- Example:

```
add class VintageSportsCar
  inherits SportsCar, VintageCar
  type tuple(PastVictories: set(string))
```

- Each instance of VintageSportsCar belongs to both SportsCar and VintageCar
- There can be objects that belong to both SportsCar and VintageCar, but that do not belong to VintageSportsCar

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Representation of objects belonging to the classes AUTOMOBILE, SPORTSCAR, VINTAGECAR and VINTAGESPORTSCAR



© McGraw-Hill 1999

Conflicts

- Multiple inheritance may produce *name conflicts*, when two or more super-classes have methods or properties with the same name
- Possible solutions
  - forbid definitions that introduce conflicts
  - define mechanisms for conflict resolution (e.g., static ordering on classes or dynamic choice)
  - force a local redefinition of conflicting properties and methods

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Persistence

- Objects can be *temporary* (as in traditional programs) or *persistent*
- Persistence can be specified in several ways (a system may present only some of them):
  - insertion into a persistent class
  - reachability from persistent objects (it guarantees referential integrity, but it requires a garbage collector to determine erasable objects)
  - denomination, associating a name ("handle") with an object

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Redefinition of methods

- *Overriding* consists in the redefinition of a method in a sub-class (e.g., method *display*)
- Then the system presents several distinct versions of the same method (*overloading*)
- The choice of the method to execute depends on the class to which the object belongs; if the class is not known at compile-time, it is necessary to do a *late binding*

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Refinement of properties and methods

- Considerable attention must be paid to changes on the interface of redefined methods
- When a method is refined in a sub-class, it parameters may be defined in two ways
  - *co-variant*: the parameters are sub-types of the parameters of the super-class
  - *contro-variant*: the parameters are super-types of the parameters of the super-class
- The co-variant solution is common, but it introduces anomalies when input parameters are redefined

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Co-variant definition

```
add class User inherits Programmer

add class File …
method init(Name:string,Owner:User)

add class Source inherits File ...
method init(Name:string,Owner:Programmer)
```

- It is not possible to guarantee at compile-time that the invocation of the method on File is correct (the file can actually belong to class Source, but the `Owner` parameter can be outside of class Programmer)

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

The object oriented database manifesto

(Atkinson, Bancilhon, DeWitt, Dittrich, Maier, Zdonik)

- A list of properties for the definition (and evaluation) of OODBMS.
- It includes:
  - Mandatory properties (the "golden rules")
  - Optional properties
  - Open choices

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

The Golden Rules

- Thou shalt support complex objects
- Thou shalt support object identity
- Thou shalt encapsulate thine objects
- Thou shalt support types or classes
- Thine classes or types shalt inherit from their ancestors
- Thou shalt not bynd prematurely
- Thou shalt be computationally complete
- Thou shalt be extensible
- Thou shalt remember thy data
- Thou shalt manage very large databases
- Thou shalt accept concurrent users
- Thou shalt recover from hardware and software failures
- Thou shalt have a simple way of querying data

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Quasi-mandatory properties ("no consensus")

- Derived data and view definition
- DBA services
- Integrity constraints
- Services for schema update

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Optional properties

- Multiple inheritance
- Type checking and type inference
- Distribution
- "Design transactions" (long and nested transactions)
- Version management

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Object Database Management Group and ODMG-93

- ODMG is a committee with academic and industrial participants, among them the main producers of OODBMSs: SunSoft, O2, HP, TI, AT&T, Versant, ONTOS, Objectivity, Itasca, ...
- ODMG-93 is a proposal of a standard for OODBMS, that the members of the consortium intend to implement and support
  - an object data model
  - ODL, a definition language based on IDL (OMG)
  - OQL, a query language based on SQL3
  - Language bindings for C++ and Smalltalk

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

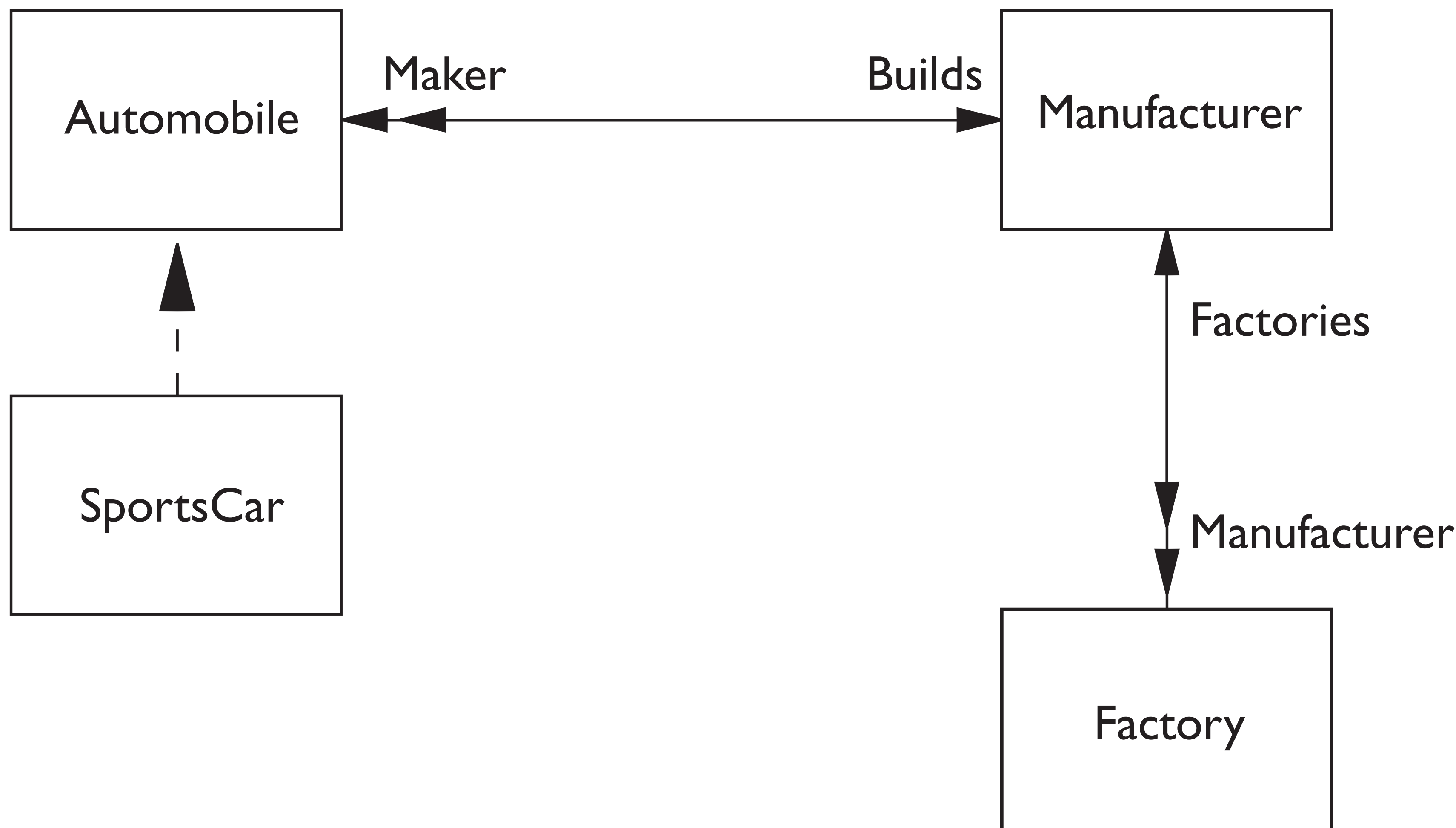
Example of ODL schema

```
interface Automobile
  {extent Automobiles key RegistrationNumber}
  { attribute string RegistrationNumber;
    attribute string Model;
    attribute string Colour;
    attribute integer Price
    attribute structure MechanicalParts
    { string Motor,
      string ShockAbsorber};
    relationship <Manufacturer> Maker
      inverse Manufacturer::Builds;
    Automobile init (in string RegistrationNumber_par,
                     in string Model_par,
                     in string Colour_par,
                     in integer Price_par);
    void Increase (in integer Amount) raises(ExcessivePrice); }

interface SportsCar: Automobile
  {attribute integer MaxSpeed}
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Object-oriented database schema for the description of cars according to the ODMG-93 model



© McGraw-Hill 1999

ODMG-93 data model

- It is possible to define a type hierarchy; some members can be *abstract* (without instances)
- The *extent* of a type is the set of all its instances
- A type has one or more *implementations* (*classes*) — for instance, a class can decide to implement a property of type **Set** as a **Set_as_Btree**
- Objects can have a *name*, to permit explicit references to them in the programs

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL — Object Query Language

- OQL is a SQL-like language for object databases, initially developed for O2, adopted (with a few changes) by ODMG, and based on the following principles
  - it is not computationally complete, but it can invoke methods, and methods can include queries
  - it permits a declarative access to objects
  - it is based on the ODMG data model
  - it has an abstract syntax *similar* to SQL
  - it has high-level primitives for collections
  - it does not offer statements for modifying objects
  - it does not offer control structures
  - it can be optimized, as it is declarative

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL by examples (1)

- Find the registration numbers of the red cars:

```
select distinct X.RegistrationNumber
from X in Automobile
where X.Colour = "red"
```

- Find the registration numbers of the red cars that won the 1954 Italian Grand Prix:

```
select X.RegistrationNumber
from X in VintageSportsCar
where X.Colour = "red"
  and "Italian GP 1954" in X.PastVictories
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL by examples (2)

- Find the registration numbers of the vintage cars built at Maranello and driven by Fangio:

```
select x.RegistrationNumber
from x in VintageSportsCar
where x.Driver.Name = "Fangio"
  and "Maranello" in x.Maker.Factories.Name
```

- Find the persons who are both drivers and manufacturers of the same sports cars:

```
select x.Driver.Name
from x in VintageSportsCar
where x.Driver = x.Manufacturer.President
```

different from:

```
select x.Driver.Name
from x in VintageSportsCar
where x.Driver.Name = x.Manufacturer.President.Name
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL by examples (3)

- Find the Ferrari sports cars that were constructed at Maranello and have a maximum speed of over 250 Km/h:

```
select a.RegistrationNumber
from a in SportsCar, c in Manufacturer, s in Factory
where c = a.Manufacturer and s in c.Factories
  and s.City = "Maranello" and c.Name = "Ferrari"
  and a.MaxSpeed > 250
```

- Find the distinct models and colours of the sports cars that won the 1986 Le Mans 24 Hours:

```
select distinct struct(Model: x.Model,
                       Colour: x.Colour)
from x in VintageSportsCar
where "LeMans86" in x.PastVictories
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL by examples (4)

- Find the names of the manufacturers who sell sports cars at a price higher than 200000; for each of them, list the city and number of employees of the factories:

```
select distinct struct(
  Name: x.Maker.Name,
  Fact: (select struct (Cit: y.City,
                        Emp: y.NoOfEmployees)
          from y in Factory
          where y in x.Maker.Factories))
from x in SportsCar
where x.Price > 200000
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL by examples (5)

- Find the number of models of cars built by manufacturers that have a global total of employees, in all factories, higher than 4500:

```
select count(
  select distinct x.Model
  from x in
    (select y
     from y in Automobile, z in Manufacturer
     where z = y.Maker
       and sum(z.Factories.NoOfEmployees) > 4500))
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL by examples (6)

- Find the list of registration numbers in the class of cars:

```
sort x in Automobile by x.RegistrationNumber
```

- Find the number of cars grouped according to their manufacturers:

```
group a in Automobile
by (constr: a.Maker)
with (AutoNumber: count(
        select x
        from x in partition))
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL by examples (7)

- Classify the sports cars into low, medium and high according to the price:

```
group a in SportsCars
by (Low: a.Price < 50000,
    Medium: a.Price >= 50000 and
            a.Price < 100000,
    High: a.Price >= 100000)
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

OQL by examples (7)

- Classify the sports cars into low, medium and high according to the price and count the number of elements present in each partition:

```
select struct(Low: x.Low,
              Medium: x.Medium,
              High: x.High,
              Total: count(x.partition))
from x in
  (group a in SportsCar
   by (Low: a.Price < 50000,
       Medium: a.Price >= 50000 and
               a.Price < 100000,
       High: a.Price >= 100000))
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Object-Relational databases (ORDBMSs)

- Data model
- Query language

- SQL-3 is the most recent version of SQL (SQL-99 is the official name)
- It maintains backward compatibility with SQL-2 (i.e., SQL-2 statements are valid for the SQL-3 syntax)

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

SQL-3 data model

- It is possible to introduce types of two families:
  - tuple types, possibly with a complex structure and with hierarchies:
    - can be used to define tables with the same schema
    - can be used as components
    - can be used in relationships
  - abstract types

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Tuple types

```
create row type PersType(
  Name varchar(30) not null,
  Residence varchar(30),
  TaxCode char(16) primary key)

create table Person of type PersType
create table Industrial of type PersType
```

- Tuples correspond to the objects
- Relations correspond to the classes

- Identifiers are updateable
- It is possible to use references and to include objects

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Tuple types

```
create row type FactoryType(
       Name varchar (25),
       City varchar (7),
       NoOfEmployees(7))

create row type ManufacturerType(
       ManufID ref(ManufacturerType),
       Name varchar (25),
       President ref(PersType),
       Factories setof(FactoryType))

create row type CarPartType(
       Engine char(10),
       SchockAbsorber char(5))

create row type AutoType(
       RegistrationNumber char(10) primary key,
       Model varchar (30),
       Maker ref(ManufacturerType),
       MechanicalParts ref(CarPartType))
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Tuple types

```
create table Automobile of type AutoType

create table Manufacturer of type ManufacturerType
values for ManufID are system generated
scope for President is Professor


create row type VintageCarType(
  Manufactureyear integer)
  under AutoType

create table VintageCar of type VintageCarType
        under Automobile
```

- Alternatively (but with a “non reusable” type):

```
create table VintageCar(
  Manufactureyear integer)
  under Automobile
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Abstract types

- Abstract types can be used as components in the definition of tuple (the abstract type may have a complex structure)
- Abstract types can also be associated with functions (this introduces the “extensibility” of the type system), defined in SQL-3 or in an external language

```
create type CarPartType(
    Engine char(10),
    Power integer,
    Cylinders integer,

    equals EqualPower,
    greater than GreaterPower)

create function GreaterPower(:p1 CarPartType, p2: CarPartType)
    returns boolean;
    returns ((:p1.Power > :p2.Power) or
             ((:p1.Power = :p2.Power and
              (:p1.Cylinders > :p2. Cylinders)))
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Queries in SQL-3

- SQL-2 queries are correct in SQL-3
- Moreover:
  - it is possible to “chase” references
  - it is possible to refer to the OIDs (if visible)
  - it is possible to access the internal structures
  - it is possible to nest and unnest

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Queries in SQL-3

```
select President -> Name
from Manufacturer
where Name = 'Fiat'


select Name
from Manufacturer, Industrial
where Manufacturer.Name = 'Fiat'
  and Manufacturer.President = Industrial.ManufId

select Maker -> President -> Name
from Automobile
where MechanicalParts..Motor = 'XV154'
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Unnesting and nesting in SQL-3

```
select C.Name, S.City
from Manufacturer as C, C.Factories as S
```

```
select City, set(name)
from Manufacturer
group by City
```

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

The Third Generation Database System Manifesto

(Stonebraker, Rowe, Lindsay, Gray, Carey, Brodie, Bernstein, Beech)

- An answer to the OODMS manifesto
- Main claim: ”Next generation DBMSs should be obtained evolving current (relational) DBMSs"

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

The principles of the counter-manifesto

- Third generation DBMSs should be a (compatible) generalization of second generation DBMSs
- Beyond the offer of traditional data management services, they should permit the definition of complex objects and rules
- They should be open to the integration with additional subsystems

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Manifesto of 3GDBMS: details

[1.1] rich type system
[1.2] inheritance
[1.3] functions and encapsulation
[1.4] OIDs only if there are no keys
[1.5] rules and triggers

[2.1] non procedural, high level access languages
[2.2] specification techniques for collections
[2.3] updateable views
[2.4] transparency of physical parameters

[3.1] multiple high level languages
[3.2] persistent x, for many x's
[3.3] SQL is a standard (even if you don't like it)
[3.4] queries and their results are the lowest level of communication

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Multimedia databases

- Multimedia data types
  - *Images*: several formats, compressed and not; considerable memory occupation
  - *Audio*: an audio signal is segmented in a big number of temporal frames and the average strength of the audio in the frame permits the reconstruction of the original audio; they also require a considerable memory occupation, even using compression
  - *Video*: sequence of images (video frames); the memory requirements are still bigger
  - *Documents*: texts, with possibly included images, with a variable degree of structural richness
  - *Annotations*: free-form user-generated information, typically associated with multimedia information

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Queries on multimedia data

- Queries on multimedia data often integrate two different approaches
  - conditions on the structured part of the information (e.g., the value of a field on the cards associated with a document)
  - “similarity” conditions, that require ad-hoc techniques and normally produce probabilistic answers (a multimedia object satisfies a given similarity criterion with a certain probability)
- Adaptive techniques, that guide users in the search of the required information, can be very useful

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Document search

- Search of documents is the most typical case of query on multimedia data (e.g., Internet search engines)
- Typical techniques
  - exclusion of stop words
  - reduction of similar words to a single keyword (stemming)
  - consideration of word frequencies
- Implementation depends on the particular language (English, Italian, etc.)
- Two indicators of search quality
  - precision: percentage of relevant documents among those extracted
  - recall: percentage of extracted relevant documents among all the relevant documents in the database
- Physical techniques: matrices, inverted indexes, signatures

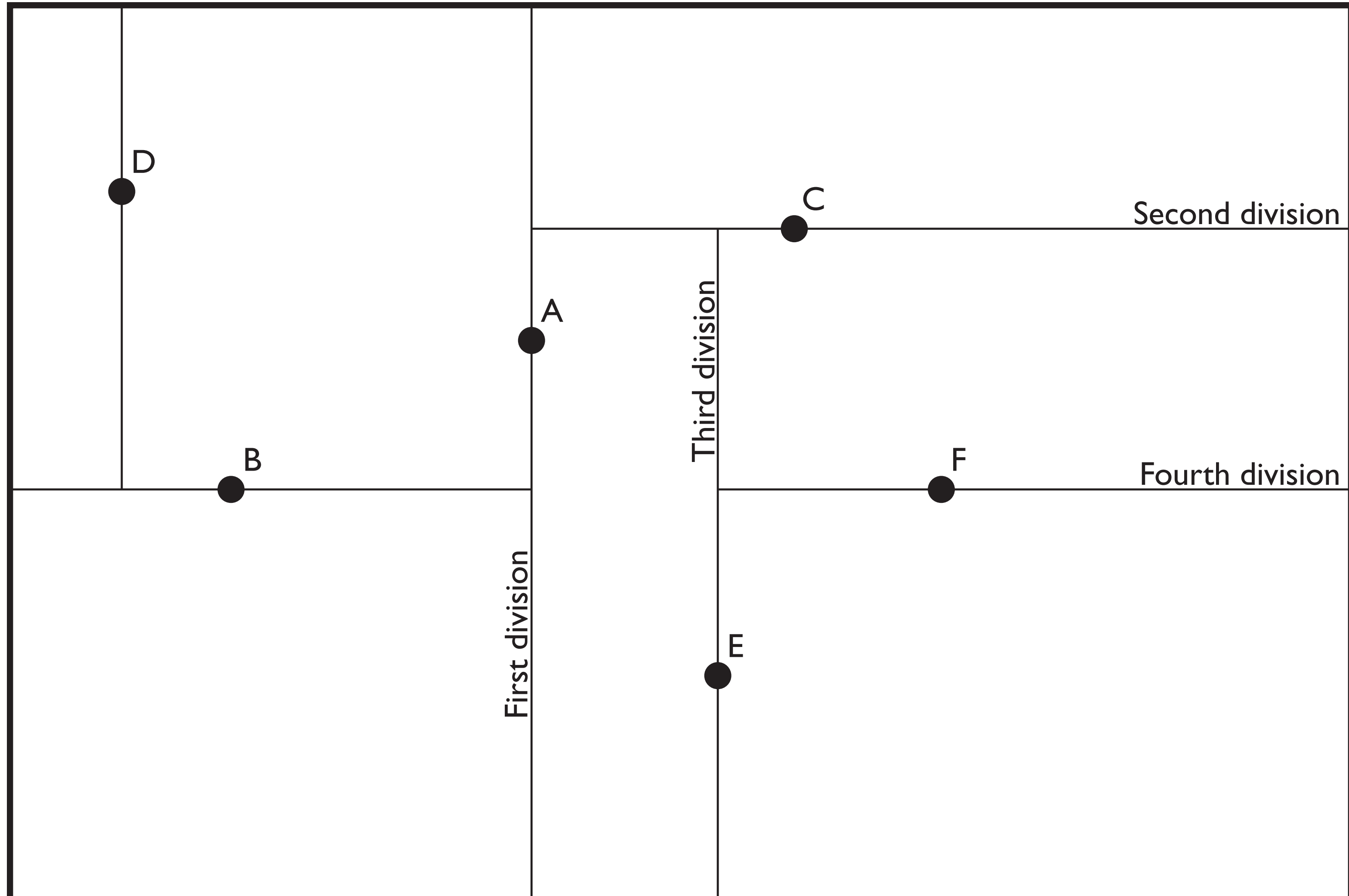
© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Representation of spatial data

- Geographical Information Systems (GIS) are systems dedicated to the management of geographical and spatial information
- The main problem is the choice of a data structure that permits to answer queries efficiently
- 2d-tree e quad-tree structures build a tree representation of points in the space

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

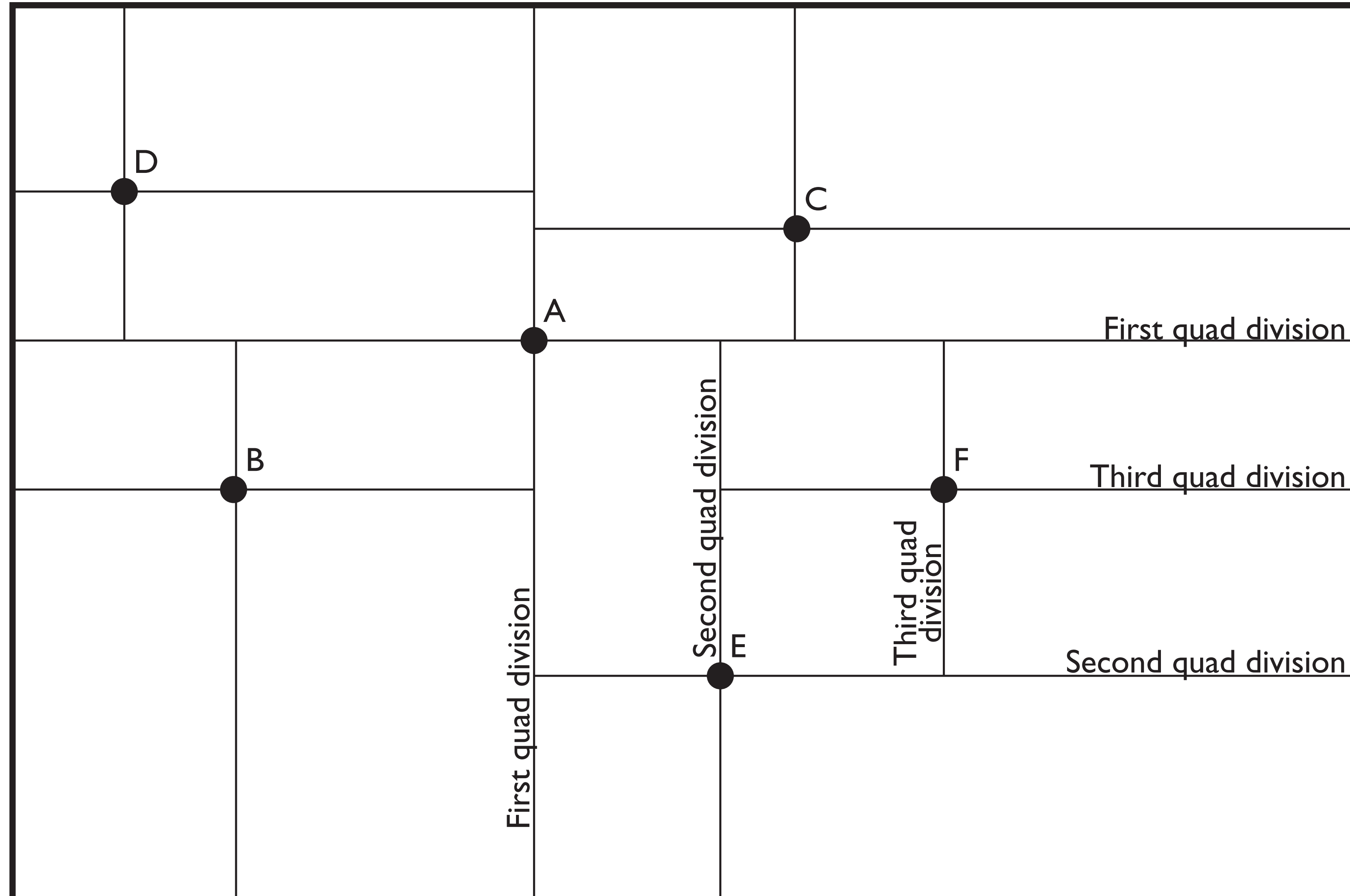
A 2-d tree



© McGraw-Hill 1999

McGraw Hill

A quadtree



© McGraw-Hill 1999

Technology of object databases

- Implementation of object databases presents a number of specific technological problems
- Some of the issues
  - representation of data and identifiers
  - complex indexes
  - client-server architecture
  - transactional model
  - distributed object architecture

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Representation of objects

- We can design a relational representation of an object database — for a class hierarchy there are two main alternative solutions:
  - *horizontal approach* — each object is stored adjacently in the most specific class to which it belongs
  - *vertical approach* — objects are divided into their components (properties), which are stored adjacently
- The horizontal approach favours access to the complete object, the vertical approach the search of objects based on their properties
- BLOBs are represented in specific files

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Representation of identifiers

- There are several solutions for the representation of objects identifiers
  - use of a *physical address*, that is, an explicit reference to the block of secondary memory containing the object representation
  - use of a *surrogate*, that is, a symbolic value uniquely associated with an object. Adequate access structures must then be designed, to guarantee an efficient access to the object starting from its identifier
- In the (frequent) situation of distributed objects, it is necessary to guarantee the uniqueness of identifiers

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Complex indexes

- Query languages permit the use of *path expressions*, to navigate the structure of objects and to access related objects
- Operators "." and "->" are used to access the properties of a record type, to follow references, and sometimes to access the members of a collection

```
select X.Maker.Factories.City.Name = "London"
from X in Automobile
```

- On the most used paths it is possible to define indexes, which permit both a forward (find the names of the cities of the factories of a car maker) or backward access (find the cars whose maker has a factory in London)
- There are specific index mechanisms for the distinct data types (multimedia, spatial and temporal data)

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Client-server architecture

- In the relational world, the client-server paradigm is a well known approach to application design, with the server responsible of executing SQL statements
- In object systems a different approach is typical, based on the use of check-out and check-in operations on objects that permit clients to import objects and to manipulate them locally
- In general, new issues arise— for instance, if the object identifier is a physical pointer, it is necessary to convert it into a value adequate for the new context (*pointer swizzling*)

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Transactional model

- Design applications require more complex models than those based on locking
  - long transactions (with a duration of minutes, hours, or even days)
  - complex transactions:
    - nested transactions
    - saga: transactions with compensating transactions
- Common solutions
  - check-out and check-in of objects
  - object versions,
    object collection versions,
    object schema versions

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

Distribution and interoperability

- Several distributed object architectures have been defined: CORBA, DCOM by Microsoft, Enterprise Java Beans by Sun/IBM, ...
- CORBA (*Common Object Request Broker Architecture*) is a standard proposal by the OMG (*Object Management Group*) with the goal to guarantee interoperability among distributed objects, using an ORB (*Object Request Broker*)
- The ORB is a *software bus* for distributed objects, responsible of the exchange of messages and invocations between objects
- In general, the goals of an ODBMS and an ORB are different, and each one can offer services to the other

© **McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999**

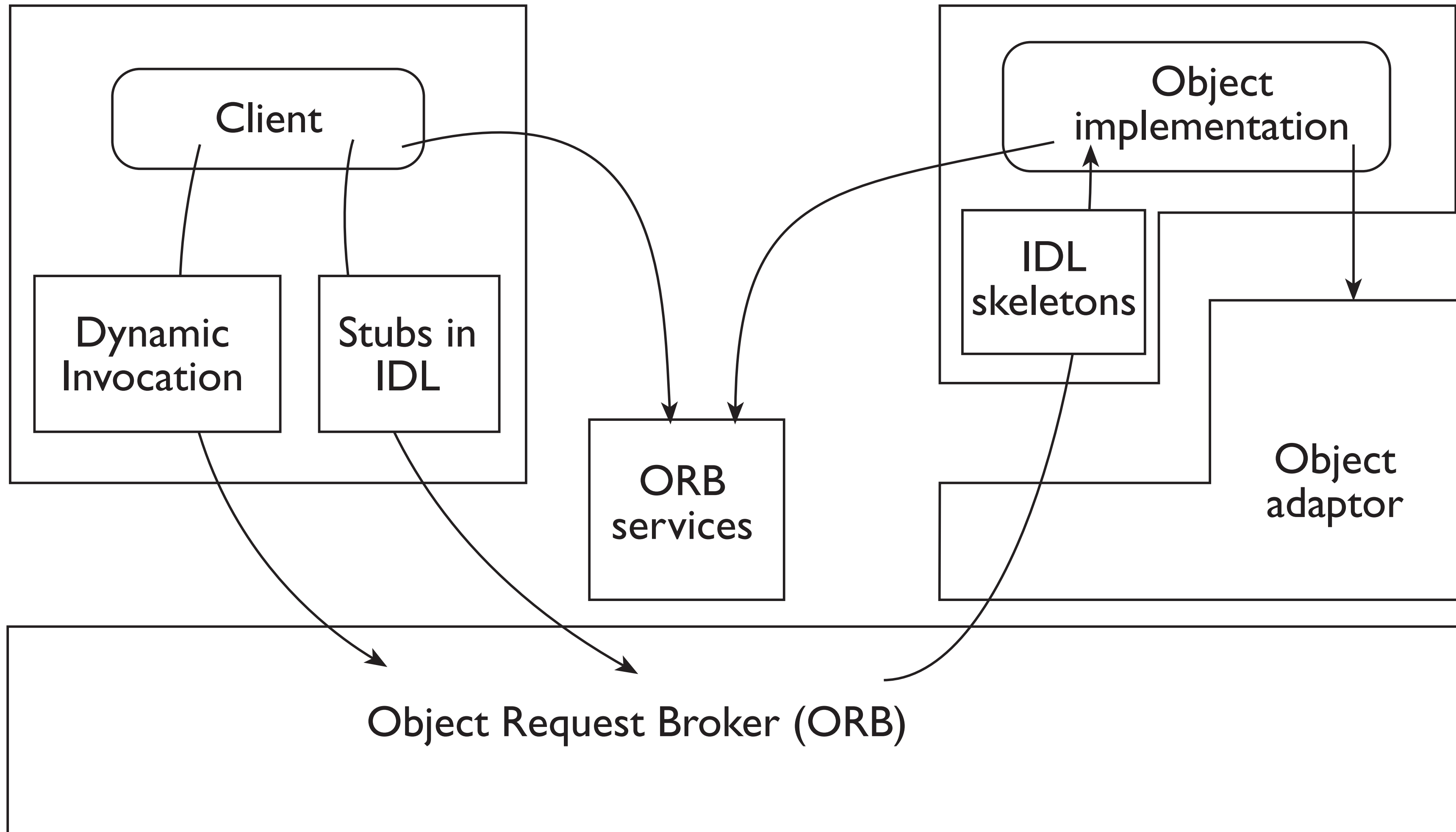
CORBA

- Each object has an *interface* (in a common language, IDL), an implementation (in any language), and is located on a node of the distributed system
- A client can invoke a method of an object in a *static* way, referring to the object interface. The invocation occurs with a mechanism similar to a Remote Procedure Call (RPC), based on a static link with the object *stub*
- This mechanism makes invisible to the programmer the object distribution, its actual location, the mechanism used for object selection, and all the issues related to format conversion of the parameters
- Method invocation can also be *dynamic*

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999

McGraw Hill

Interaction among the four components of a CORBA architecture



© McGraw-Hill 1999

CORBA architecture

- The *object adaptor* offers services to register objects and their implementations, and to access object implementations
- The architecture is based on four main components
  - ORB
  - *object services* — some basic functions (standard) for the objects (e.g., security, persistence, transactions)
  - *common facilities* — some optional services for clients, (e.g., user interfaces, information management, system management, task management)
  - *applicative objects* — defined by the user for the application

© McGraw-Hill and Atzeni, Ceri, Paraboschi, Torlone 1999