Information Systems

Technical Course/Spezialvorlesung

Part I: Object Oriented Modeling and Design

PD Dr. Wolfgang May

Dates: Thursday, 14.15 – 15:45 Uhr, SR 052-02-017
Friday, 9.15 – 10:45 Uhr, SR 051-00-006

Overview (Preliminary)

I. Design of information systems
- object orientation
- object-oriented data model
- object-oriented modeling
- concepts: standardization by the Object Management Group (OMG)
- Graphical representation: Unified Modeling Language

Characterizations ...

- *Elmasri, Navathe; Fundamentals of Database systems; Benjamin/Cummings, 1994*:

  An information system includes all resources within the organization that are involved in the collection, management, use, and dissemination of information. In a computerized environment, these resources include the data itself, the DBMS software, the computersystem hardware and storage media, the personnel who use and manage the data (...), the applications software that accesses and updates the data, and the application programmers who develop these applications.

- *Ramakrishnan; Database Management Systems; WCB/McGraw-Hill, 1998*:

  Today, more than at any previous time, the success of an organization depends on its ability to acquire accurate and timely data about its operations, to manage this data effectively, and to use it to analyze and guide its activities.

  Phrases such as the information superhighway have become ubiquitous, and information processing is a rapidly growing multibillion dollar industry.

Lecture Information Systems

- *Lockemann, Krüger, Krumm; Telekommunikation und Datenhaltung; Hanser, 1993*:

  A system is a set of elements that are correlated by well-defined relationships.

  In case that these relationships are intended to exchange and manage information, it is called an information system.

- *Kappel, Schrefl; Object-Oriented Information Systems; Springer, 1996*:

  Information systems are data-intensive software systems, i.e., they provide the functionality for managing and manipulation of large quanitities of information; thus they are in general based on a database system.

  The following views can be distinguished:

  - Data
  - Events (WHEN is something executed)
  - Processes (WHAT is executed)
    Business Processes, Workflows

Information System = Database + Application Program

Database: statical aspect:

Storing and querying data: `INSERT`, `UPDATE`, `DELETE`, `SELECT`, simple constraints.

- data repository: `INSERT` and `SELECT` predominate.
  Databases that are mainly used for storing large datasets (e.g. measures in biosciences, medicine) that are queried infrequently.
- operative database: data is stored, changed, used, and deleted. Such databases always mirror the current situation of some process (e.g. production processes, client data, banking).
  Applications for *Data Warehouses, OLTP/OLAP* and *Data Mining*.

Integration of dynamic aspects:

Active databases, constraints, procedures, workflows, interaction with the user.

Application programs on top of the above “simple” operative databases in business.

Lecture Information Systems

Information System = Database + Application Program

Integration of application functionality into existing operative databases:

First steps:

- Relational DBMS: Applications are programmes around database systems by using Embedded Cobol/Pascal/C (invocations of SQL from host languages).
- move application logic into the DBMS:
  procedural extensions to relational DMBS (Oracle: PL/SQL): Procedures, functions, and triggers can be programmed in a *procedural* programming language *inside* the DBMS.

Next step:

Object-Orientation in Databases

- object relational databases: *object types* and *object methods* allow for coding object-oriented behavior (still without a class hierarchy) *inside the DBMS*.
- by integration of databases into object-oriented programming languages (JDBC/Java), databases can be integrated directly into object-oriented applications.
- Discovery: difference between “program” and “database” diminishes. There is the need for working with the same objects in the DBMS and in the application programming language.
- Problem: *impedance mismatch* between the data types of the DBMS and the language SQL, and those of the application programming language.

$\Rightarrow$ intersection of software engineering and database topics

Software Engineering

“Software Crisis” 1965

- Programs did not do what they should, or they performed erroneously.
- Software costs – in case that it was completed – a multiple of the planned money.
- Deadlines were rarely met, and later changes of existing programs were nearly impossible.
- *Software designers and programmers were not longer able to handle larger programs with the available techniques and tools.*
- The techniques how small programs have been designed and coded were not sufficient to manage the upcoming problems of developing and maintaining larger programs, following the technological progress.

Small Problems

- abstract problem description (WHAT)

  solution, algorithm (HOW)

  in general done by a single person
- focus of interest: correctness and efficiency
- precondition: well-structuredness of the programs
  - adequate procedural decomposition (top-down approach)
  - structured programming (no GOTO-statements)

Lecture Information Systems

Large Problems

- software engineering as an engineering discipline:
  - Software of high quality
  - cost-efficient, with a given budget
  - completed in-time
- **Life-cycle-model**/**Waterfall-model** of the stages of the software development process

  Analysis and definition

  $\Longrightarrow$ Design

  $\Longrightarrow$ Implementation

  $\Longrightarrow$ Test

  $\Longrightarrow$ Use and Maintenance

Software Crisis (1990s)

Critics against the waterfall-model:

- sequential stages are unrealistic
- each stage leads to feedback to previous stages
- the assumption to have closed and correct product definitions is not realistic
- a usable version is only provided at the end of the development
- each stage uses different modeling concepts.

Alternatives:

- iterated phases model
- spiral model
- generation of a prototype
- evolutionary software development
- transformational software development

... but none of these approaches allows for a unified modeling concept, that especially supports an integrated handling of data and operations

Lecture Information Systems

Object Orientation

- Idea “object orientation”: early 70s
- object-oriented programming: Simula (1970), Smalltalk-76, C++(1985), Eiffel (1988), Java (1995).
- object-oriented analysis/design: early 90s
  Booch, Coad & Yourdon, Jacobsen, Rumbaugh, Shlaer & Mellor, ...

  Abstract description of processes, not only of programs.
- UM (*Unified Method*; Booch & Rumbaugh, 1995), then together with Jacobson (Use Cases) UML (*Unified Modeling Language*).
- UML1.0 submitted 1997 to OMG (Object Management Group) for standardization.

OO Modeling of an (Information) System

- data model (static):
  - object state,
  - relationships between objects,
  - class hierarchy
- behavior (dynamic):
  - OO-programming: objects provide operations (Methods). These are invoked by sending a suitable message *object.operation(...)* to the object.
  - Operations change the internal state of the *host-object*.
  - Operations often also effect the state of other objects ⇒ communication between objects.
- modeling by
  - state transitions of the objects,
  - activities and processes/workflows (dynamic specification)
  - cooperation of several objects in a process.

Object-oriented *modeling* is not the same as object-oriented *programming*!

Lecture Information Systems

Characteristics

- **Integrated view:**

  Instead of separating data and operations, the class concept allows for integrated units of data and operation.
- **Better possibilities for abstraction.**

  In contrast to earlier methods, the modeling is moved from the modeling the solution to modeling the problem/application (“what is modeled” instead of “how is it solved”).
- **Methodical Continuity.**

  The result of some step in the (software)-design-process can be taken over to the subsequent step. The representation formalism is not changed.
- **Evolutionary Development.**

  A complex system is not created in one go. Using object-oriented software development methods, the principle of evolution can be transferred to the software development process.

OO Programming vs. OODBMS

Lifetime of Objects

- Object-oriented programming language: objects are created when an application program is executed, and they are destroyed after its termination.
- if we have a database system:
- persistent: objects are created and stored in the DBS in an activity, they remain when the creating activity terminates.
  Ex.: countries, cities, that are in general forever in the DB; students: are created by a short *registration* activity and remain until another activity removes them from the register.
- transient: objects are needed only temporarily for executing an activity. They exist only as long as a certain application is running. They are managed by the runtime system of the programming language.
  Ex.: during an activity, some object creates an auxiliary object for the result of some evaluation, surrenders it to another object that executes some activities and then destroys the auxiliary object.

Lifetime of Objects

- Relational DBMS: All SQL-types have only persistent instances that are located in the DBMS. All non-SQL-types (i.e., types of the host language) have only transient instances, that are destroyed upon termination of the application programs (= i.e., leaving the host language environment).

  On the other hand, persistent objects can be manipulated only by SQL, whereas transient objects can be manipulated only by means of the host programming language.

  $\Rightarrow$ impedance mismatch
- ODBMS: the object types of the DBMS and of the application are the same. They may have persistent and transient instances at the same time.

  For persistent and transient objects, the same programming language and the same operations are used.

Active and Passive Objects

- An object that exists, but is not loaded (into the runtime environment) by the program is called *passive*. The objects that are loaded during runtime are called *active*.
- for persistent objects, *virtual* data can be defined that is present only when they are active (cf. views).
  if an object of type employee is activated, its tax class is computed based on its family status, and can be queried.

OO-DBMS

Standardization, similar to the standardization of relational databases (SQL standards):

Success of relational database systems:

- not only due to the simple, high-level data model,
- but also due to its standardization: SQL
  - Portability
  - Interoperability

ODMG: Object Database Management Group

- Founded 1991
- Architecture of OODBMS, DDL, Query Language (OQL), C++/Java/Smalltalk Language Bindings
- ODMG-93 (1.0) Standard
- ODMG-2.0-Standard (1997)
- ODMG-3.0-Standard (2000); incremental modifications

Literatur: Cattell et al; Object Database Management

ODMG: Object Database Management Group

- Voting Members: Organisations and companies who offer commercial ODBMS, e.g., JavaSoft, Lucent, GemStone, ObjectDesign, Versant, Unidata, Windward Solutions ...
  Reviewer Members: Organisation who have a material interest in the work of ODMG
- the goal is not to define identical products, but to reach source-code portability (cf. Java, XML, SQL).
- leaving enough freedom to define the own products:
  - Performance, optimization (price)
  - Support of programming languages,
  - predefined functionality and types for certain application areas (Multimedia, CAD, ...),
  - integrated programming environments, design tools ...
- ODMG 1.0-Standard (1993),
  ODMG 2.0-Standard (1997),
  ODMG 3.0-Standard (2000)

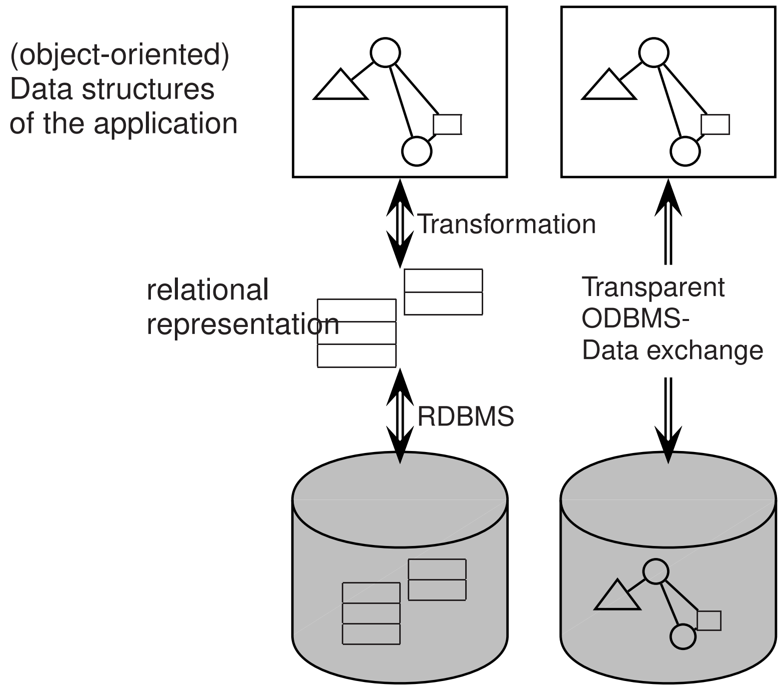
Architecture of ODBMS

- Different from “classical” relational DBMS:
  SQL: High-level language for data manipulation, application must be written in another programming language (cf. embedded SQL, JDBC).
- ODBMS/ODM: transparent integration of DBMS-functionality (Persistence, Concurrency Control, Data Recovery, Associative Queries) into application programming languages (cf. Persistent Java).
  The objects of the application are simply stored in a database.
- no separate DML necessary, the application programming language *is* the DML.
- no transformation between the (logical) database representation and the representation in the programming language (cf. Data type conversion in JDBC).
- Performance advantages by data-caching in the application.

Architectures

ODMG defines two types of products:

- Object Database Management Systems (ODBMSs) store the objects directly,
- Object-to-Database Mappings (ODMs) convert objects and store them in a relational (or something else ... XML) representation.



Literature

- Rumbaugh, Blaha, Premerlani, Eddy, Lorenson: Object-oriented Modelling and Design, Prentice Hall, 1991; German Translation: Hanser, 1993
- Heuer: Objektorientierte Datenbanken, Addison-Wesley 1997.
- Lausen/Vossen: Objekt-orientierte Datenbanken: Modelle und Sprachen, Oldenbourg, 1996 (dt + engl)
- UML1.0: Rumbaugh, Jacobsen, Booch: The Unified Modeling Language, Rational Software Corporation, 1997.
- Oestereich: Objektorientierte Softwareentwicklung (incl. UML); Oldenbourg 1997.
- Hitz, Kappel: UML@Work, dpunkt Verlag, 1999 (in german).
- Cattell et al: The Object Database Standard: ODMG 2.0, Morgan Kaufmann, 1997.
- Cattell et al: The Object Database Standard: ODMG 3.0, Morgan Kaufmann, 2000.

Semantic Datamodels

- Relational data model: resulting schema does not mirror the structure of the application.

⇒ semantic data model (mid-70s):
  - A lot of concepts for a more direct representation of the structure of application objects.
  - DB-design and data modeling profit from such schemas

- most cases: actual data*base* design again relational.

Lecture Information Systems

Principles of "New" Database Models

- separation between internal and conceptual level
- focus on entity or object, not on tuple/record
- richer concepts
  (in the relational model, relations are used for everything: objects, (multivalued) properties of objects, relationships between objects, class memberships)
- integrity constraints on the semantic level are contained in the structure of the objects.
- modeling of static and dynamic properties,
- modeling of relationships between objects,
- modeling of integrity constraints (keys, cardinalities, inclusions)

Requirements on “New” Database Models

Abstraction levels

complex objects can be abstracted stepwise,
hide details (*information hiding*)

Direct modeling of application objects: object type of the application domain must be representable in a *single* concept of the database model,

Expressiveness: as many semantical concepts as possible
Example: n:m-relationships:

- Network data model: not possible
- Relational model: possible, but must be restricted by referential integrity constraints to valid objects
- Semantical data models: specialized constructs for representation of valid n:m-relationships

Exact definition: formal assertion of

- consistency of the concepts in the schema,
- completeness of the concepts wrt. a reference model,
- criteria for the quality of a database design (e.g. avoiding redundancy)

Note: the relational model satisfies this requirement (wrt. the ER-model as reference model; several normal forms).

Lecture Information Systems

Object-Oriented Data Model

Beeri (DOOD89, Data and Knowledge Engineering 1990) proposes to distinguish three steps:

Structure: Static structure of the objects of the application domain (common to both semantic data models and object-oriented principles):

- classes and types
- including patterns: `TUPLE OF`, `SET OF`
- object identity, complex objects
- relationships between classes,
- structural inheritance
- integrity constraints

Operations: Generic operations (extend the functionality of the relational model): querying and data manipulation languages

- SQL-style,
- calculus-style,
- deductive

Behavior: object- and class-specific operations

- definition and inheritance of methods
- abstract data types and encapsulation
- overriding and multiple inheritance

Concepts of Object Orientation

- Base elements are *objects* (Germany, Berlin, Bill Gates) and *literals* (42, 3.1415, “x”, April 1).
- objects have a unique *identifier (OID)*, and a state, and show some behavior,
- objects are grouped by classes,
- OID is independent of the state of an object (is fixed when an object is created).
- literals are simple values (no identifier, no state, no behavior).
- constructs like *tuples*, *sets*, *lists* (also called *collections*) exist in two versions:
  – “only” *values*, no oid, must belong to an object.
  – objects: have behavior of their own

Exercise 1
Compare the concepts *value* and *object*.

Lecture Information Systems

Objects

Structural Aspect

- The state of an object is given by values of its properties. Properties can be given by
  – *attributes* (name, age, area)
    germany[name→“Germany”; area→356910] or
  – *relationships* (*Relationships*, has_capital, boss_of) with other objects
    germany[has_capital→*berlin*] .

  The values of properties can change over time.
- there can be multiple objects with the same attribute values at a given timepoint.

Exercise 2
Compare the concepts *object identity* (OO-model) and *key* (relational model).

- first do the gardener=murderer example.

Solution to Exercise 1 (cf. Heuer S.288)

Remark: values can be atomic (literals; integer, string etc.), or structured things (sets, lists, tuples, dates)

- printable vs. not printable: string, integer etc. can be printed, persons and countries not.
- application-dependent abstraction: 42 is 42 – in all applications. For persons e.g., the semantics depends on the application.
- Generation/Definition: values are automatically given with the installation of an ODBMS, objects must be created later (also set etc ...). The properties of objects must also be defined later.
- Information: values immediately carry information, objects contain information only due to their attribute values. Values do not have attribute values, that could describe them in more detail
- values describe objects (but not themselve – 42 alone makes no sense) – on the other hand, objects are only described via their values: a_shoe.size does not yet contain any information, in contrast to a_shoe[size→42].

Solution to Exercise 2 (cf. Heuer S.286)

- Update-operations on key attributes: change the value of the object, or make it a completely different object (what happens to references that used that value?)
- Update-operations of foreign keys: reference another object, or change the key value of the referenced object?
- several objects with the same state are allowed. Tuples may not have the same key values
- Problem: operations that involve tuples with a different number of key attributes (union, intersection)

access to objects/tuples:

- *internally*, only oids are used for representing relationships
- the external users in general select and access objects and tuples via their properties (and thus also via their keys)
  select * from country where name = “Germany” ,
  not select * from country where oid = 1234.
- only from *programs*, there is also an oid-based access (if the program knows an oid from a previous query):
  x :=select oid from country where name = “Germany”
  select * from country where oid = x

Objects

Behavior

- The *behavior of an object* is described by its *operations* (synonymously: *methods, services*). Operations can have parameters.
- operations of an object are called by sending a suitable message to the object:
  square.show(),
  EU.make_member(Germany),
  Microsoft.employ(Bill).

Lecture Information Systems

Generic Operations

- objects are generated by *constructors*.
  - identity, shallow equality, deep equality (see later for details)
  - assignment, shallow copying, deep copying,
  - equating (at the beginning: 3 objects: detective, gardener, murderer; later only 2: detective, gardener == murderer [Heuer])

  Complex types must provide suitable operations (cf. C++):
  - Tuple: access to components, test for equality
  - Set: iterators, membership, set comparisons, set operations
  - List: first, next, tail, last, iterators, concatenation, membership, comparison (equality, containment)
- a class is also a set! ($\Rightarrow$ can be enumerated by an iterator)
- ... see later section on OQL

Classes and Types

- objects and literals are categorized by their *types/classes* (Country, City, Person).
- *both* define hierarchies!

Modeling:

- classification of things, explanation of notions (*taxonomy, ontology*),
- vehicle, motor vehicle, car, truck, ...
- definition of properties by which elements are grouped to classes/subclasses.

Programming (Java):

- derive classes from others for reusing their implementation,
- derived classes extend or specialize base classes,
- only pragmatical reasons.

Classes and Types

What *are* classes and types? Differences?

- a *class* consists of a *domain* (all objects that can belong to that class) and a current set of objects (*extension* – all objects that currently belong to the class).
  Ex.: classes *person, student, employee.*
- the type definition (also called *state type*) defines the *signature (Interface)* of objects. It contains
  - names and types of attributes,
  - names of methods, types of parameters and results

  vehicle[fuel⇒string, producer⇒string]
  lending_business[lend(something)⇒number]

  All elements of an object type can have a certain set of states.
- objects are called *instances* of their types/classes.

Structural Hierarchies: Class Hierarchy

semantical notion: “similar” objects are grouped into classes.

class hierarchy:

At every timepoint, the set of objects of a *subclass* K_{sub} is a subset of *superclass* K_{sup}.
(semantical) integrity constraint: each object of a class is also an element of each of its superclasses:

$$K_{sup} \supseteq K_{sub}$$

Specialization: definition (restriction) of the domain of subclasses based on the domain of superclasses: Student vs. Person.
Not each element of a superclass is necessarily contained in a specialized subclass (Person; Student, Employee).

Generalization: definition of the domain of superclasses based on the domain of subclasses: waters are lakes, seas, rivers

$$K_{sup} = \bigcup K_{sub}$$

Distinction between superclass and subclass, or between different subclasses of a superclass by some criterion (*Discriminator*).

Structural Hierarchies: Type Hierarchy

Type hierarchy: objects have a certain state and behavior.
The abstract state and behavior is defined by the *signature*:
A *subtype* T is more specialized than a *supertype* T', if

- base types: $dom(T) \subsetneq dom(T')$.
- compound types (tuple types): T contains at least the components of T'. Each component of T is a subtype of the corresponding component of T'.
- set/list types: the underlying type of T is more special then the base underlying type of T'.
- T is called an *extension* of T'.

person[name⇒string; birthday⇒date]
young_person[name⇒string; birthday⇒date;
{date $>$ 01.01.1980}]
student[name⇒string; birthday⇒date;
regNr⇒number; subject⇒string]
employee[name⇒string; birthday⇒date;
company⇒string; salary⇒number]

john[name→“John”; birthday→01.01.1990;
regNr→123456; subject→“Math”;
company→“bla”; salary→3000]

Structural Hierarchies: Is-a-Hierarchy

Is-a-Hierarchy: K is more special than K', if both of the following hold
(i.e. the type hierarchy adapts to the semantical hierarchy):

- K is a subclass of K', and
- the type T of K is a subtype of the type T' of K'.

- In the following, we assume that the three hierarchies are identical (clean modeling). Most OODMs require them to be identical.
- *Sometimes* it makes sense to have them uncoupled for the implementation in order to reuse program code. This has nothing to do with *modeling*.

Lecture Information Systems

Encapsulation

Encapsulation:

- Nobody else than an object itself can access its object state.
- Access to the object state from outside is possible only by the operations of an object (*interface*).

The (abstract) state of an object and its behavior is given by the type (signature) of the corresponding class:

state: *vehicle[fuel⇒string, producer⇒string]*

mb600[fuel→"SuperUnleaded",
producer→"Mercedes-Benz"]

behavior:

lending_business[lend(something)⇒number]
lendnr := info-lib.lend(uml@work)
+ side effects on the state of *info-lib* and *uml@work*

Methods without implementation are called *abstract* methods (for classes; implementations are then given for the subclasses).

Is-a Hierarchy

Subclasses refine classes:
vehicle[fuel⇒string, producer⇒string]
lending_business[lend(something)⇒number]

- *truck* is a subclass of *vehicle*
- *car_hire* and *library* are subclasses of *lending_business*
- refined signature by suitable Subtyping:
  additional attributes:
  truck[payload⇒number]
  refined typing:
  car_hire[lend(vehicle)⇒contractNr]
  library[lend(book)⇒pawnTicketNr]
- often also restricted by additional assertions: constraints, preconditions, rules that must be satisfied by the objects of a class.
  car{*maxWeight* $<$ *2.8t*}

Inheritance

properties/operations with the same name can be defined for different classes. They are called *overloaded*.

Structural inheritance: (for types)

- subtypes inherit the signature of their supertypes (structural inheritance; see definition of the type hierarchy).
- properties/operations of a superclass can be *redefined* by a *more special definition* (*overriding*).

Value inheritance (and behavioral inheritance):

- typical attribute values
  truck[fuel•→“Diesel”]
- implementation: the method *lend* is not implemented for general lending_businesses (abstract method). The implementation is then given for the subclasses
  car_hire[lend(vehicle)⇒contractNr]
  and *library[lend(book)⇒pawnTicketNr]* .
- value inheritance can be *monotonic* and *non-monotonic*.

Object-databases

- The *logical schema* of a database is given by its type specifications and by the class hierarchy.
- A *database(instance)* contains objects.
- The *extension* of an object type/class in a database is the set of all instances of the type/class, under consideration of the subclass relationship.
- The notion of an *extension* is similar to the notion of an entity type or a relation in RDBMS (see later: OQL).
- locally to an extension of a type, *keys candidate keys, simple keys, compound keys* are defined as usual based on the properties of the objects.
  - Attributes: Country(Name), Person(Name, FirstName, Birthdate)
  - Attributes and relationships: City(Name, Country); cf. weak entity types in the ER-model.
- there *can* be several different objects with the same attribute values [and relationships] at the same time. (cf. relational model: relation = multiset) Distinction then by their OIDs; not externally visible.
- The value of an object identifier is called a *reference*.

Lecture Information Systems

Object-Oriented Data Model

- describes only the *(database-)state* (*attributes, relationships, class memberships and class hierarchy*), not the *behavior*
- model of the application domain,
- relationships are often treated as object-valued attributes
- more expressive than the relational model/ER-model
- Relational data model: standardized
  - interoperability: not only specification/description, but also allows for data exchange between different (relational) database systems
  - copying of tables (export)
  - remote-access: ODBC/JDBC

⇒ object-oriented data model for exchanging object-oriented databases.

- OIF (Object Interchange Format; ODMG)
- frame-based models
- OEM (Object Exchange Model; TSIMMIS-Project, Stanford)
- XML (Extendible Markup Language)

OIF: Object Interchange Format

Data format of the ODMG-standard.

see later.

(less suitable for describing the pure data model)

Frame-based Models

- objects are represented as *object frames*,
- frame consists of *slots* for storing properties ("signature" of the slots *can* be predefined, but not necessarily ($\Rightarrow$ Semistructured Data))
- slots can be literal- or object-valued (*internal* storage by references), thus they can also describe *relationships*.

Example: F-Logic

- is-a relationship:
  $object : class$
- Subclasses-relationship:
  $class :: class$
- properties:
  $object[property \rightarrow object/value]$ (scalar)
  $object[property \twoheadrightarrow \{set\text{-}of\text{-}objects/values\}]$
  (set-valued/multi-valued)
  Analogously with parameters:
  $object[property\ @(list\text{-}of\text{-}objects/values) \rightarrow object/value]$
  $object[property\ @(list\text{-}of\text{-}objects/values) \twoheadrightarrow \{set\text{-}of\text{-}objects/values\}]$
- inheritable properties:
  $class[property \bullet\!\!\rightarrow object/value]$ (scalar)
  $class[property \bullet\!\!\twoheadrightarrow \{set\text{-}of\text{-}objects/values\}]$ (set-valued)

F-Logic: Example

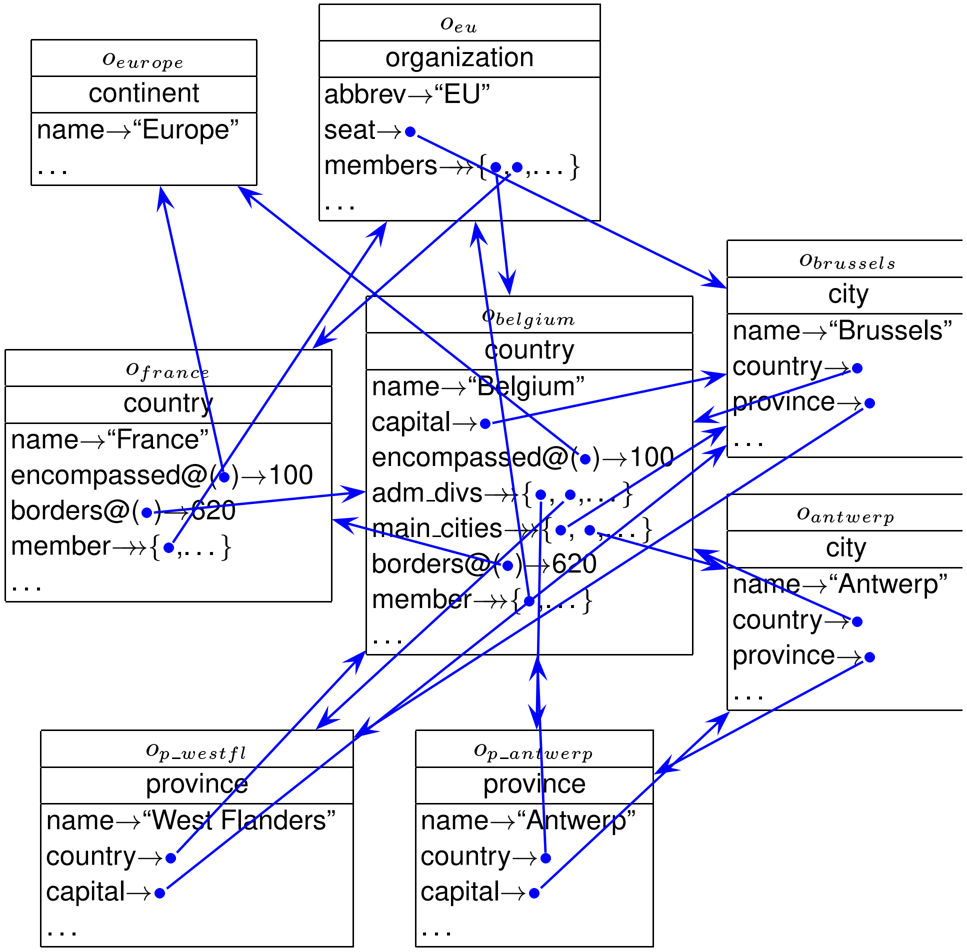
$o_{belgium}$: country[name→ "Belgium"; car_code→ "B";
capital→$o_{brussels}$; independence→ "04 10 1830";
total_area→30510; population→10170241;
encompassed@(o_{europe})→100; pop_growth→0.33;
adm_divs→→{$o_{p\_antwerp}, o_{p\_westfl}, \ldots$};
main_cities→→{$o_{brussels}, o_{antwerp}, \ldots$};
borders@(o_{france})→620; borders@($o_{germany}$)→167;
borders@($o_{luxembourg}$)→148; borders@($o_{netherlands}$)→450].

$o_{brussels}$: city[name→ "Brussels"; country→$o_{belgium}$;
province→o_{p_westfl}; population@(95)→951580].

o_{p_westfl} : prov[name→ "West Flanders"; country→$o_{belgium}$;
capital→$o_{brussels}$; area→3358; population→2253794].

o_{eu} : org[abbrev→ "EU"; name→ "European Union";
established→ "07 02 1992"; seat→$o_{brussels}$;
members@("member")→→{$o_{belgium}, o_{france}, \ldots$};
members@("applicant")→→{$o_{hungary}, o_{slovakia}, \ldots$}].

Representation as a Graph



F-Logic

- Null values are not explicitly stored, slots are simply not used.
- *navigation* instead of *join*:
  - number of inhabitants of the province where the capital of Belgium is located:
    $o_{belgium}$.capital.province.population
  - All names of cities in Belgium:
    $o_{belgium}$..main_cities.name
  - Sum of inhabitants of all provinces of Belgium:
    sum{X | $o_{belgium}$..adm_divs[population→X]}

... this does not only hold for F-Logic, but in general for the object-oriented data model (and also for the object-relational model).

Lecture Information Systems

F-Logic

The signature can also be represented in F-Logic:

- properties:
  type[*property*⇒*type*] (skalar)
  type[*property*⇒⇒*type*] (mengen-valued)
  Analogously with parameters:
  type[*property* @(*list-of-types*)→*type*]

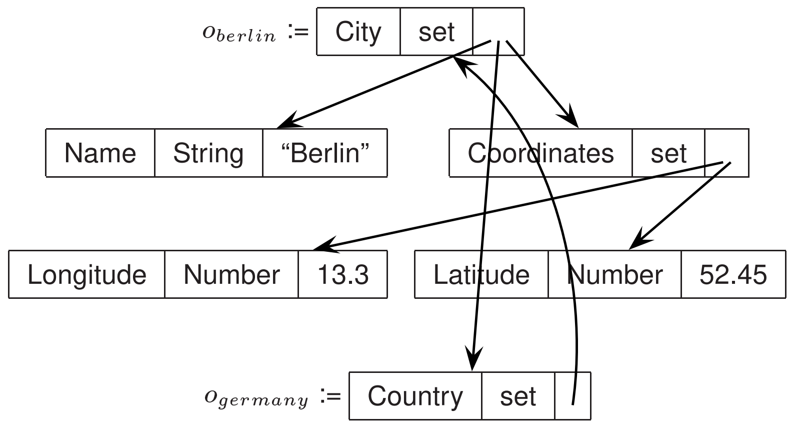
- country[name⇒string; capital⇒city;
  total_area⇒number; population⇒number;
  encompassed@(continent)⇒number;
  adm_divs⇒⇒province;
  main_cities⇒⇒city;
  borders@(country)⇒number]

  city[name⇒string; country⇒country;
  province⇒province; population@(number)→number]

- can also be seen as a graph between data types (where methods that are parametrized with non-literal types do not look very good).

OEM: Object Exchange Model

- A simple object model
- TSIMMIS-Project (Stanford)
- Each object has an Object-ID, a label (corresponding to its class), a type, and a value,
- Values of complex types are sets of objects
- representable as a graph:

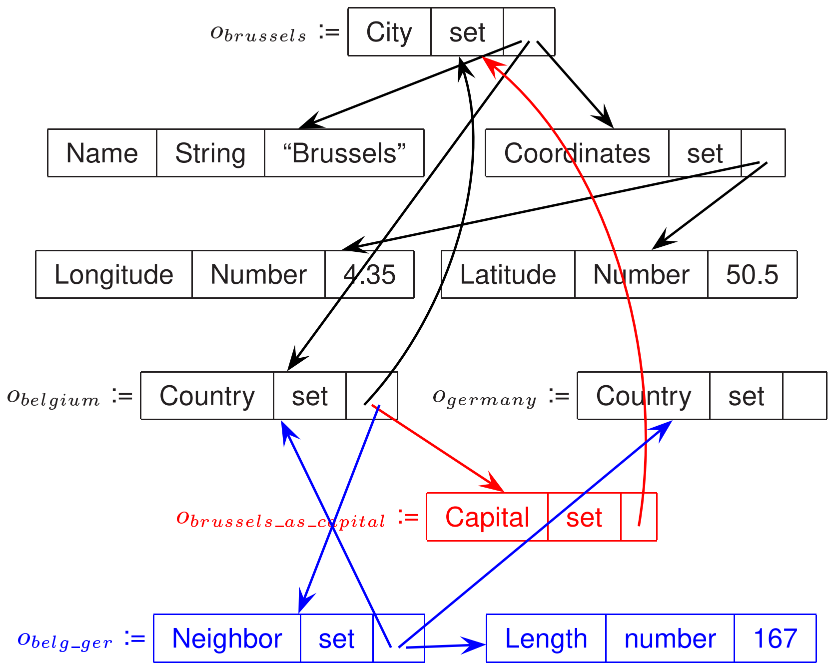


Exercise 3 (OO-Data model – Representation)
Represent the excerpt of the database given on Slide 44 in OEM, and represent the above database in F-Logic.

Solution of Exercise 3

Problems with OEM:

- relationship $o_{brussels}$ is capital of $o_{belgium}$:
  since $o_{brussels}$ has the label $city$, is this not possible
  $\Rightarrow$ substitute-object with label $capital$
- analogously for organization $\rightarrow$ seat $\rightarrow$ city
- parametrized properties: $country.borders(country)$:
  substitute-object for neighbor relationship



Compromise: Object-Relational Model

e.g. in Oracle

- Complex object types with methods:

```
CREATE TYPE GeoCoord AS OBJECT
 (Longitude NUMBER,
  Latitude  NUMBER,
  MEMBER FUNCTION Distance (other IN GeoCoord)
         RETURN NUMBER);
CREATE TYPE City_Type
 (Name VARCHAR(35),
  Country Country_Type,
  Population NUMBER,
  Coordinates GeoCoord,
  MEMBER FUNCTION Distance (other IN City_Type)
         RETURN NUMBER);
```

- Storing objects in tables
  - object tables contain row objects

    ```
    CREATE TABLE City OF City_Type;
    ```

- object-valued attributes vs. *reference attributes*
  - object-valued attributes create *column objects* that do not have an OID (*City.Coordinates*).
  - reference attributes point to an object in an object table (*City.Country*).

Object-Relational Model

Distance-method based on coordinates and lifted to “object level”:

```
SELECT c1.distance(c2)     % City-level
FROM City c1, City c2
WHERE c1.name='Freiburg'
  AND c2.name='Berlin';

SELECT c1.coord.distance(c2.coord)  % coord-level
FROM City c1, City c2
WHERE c1.name='Freiburg'
  AND c2.name='Berlin';

SELECT (DISTINCT)     % relational without object-methods
   distance(c1.longitude, c1.latitude,
            c2.longitude, c2.latitude)
FROM City c1, City c2
WHERE c1.name='Freiburg'
  AND c2.name='Berlin';
```

Object-Relational Model

- Distance between a city and a mountain?
  is not possible on object-level due to
  `City_Type: Distance (other IN City_Type).`

```
SELECT c.coord.distance(m.coord)  % coord-level
FROM City c, Mountain m
WHERE c.name='Freiburg'
  AND m.name='Feldberg';
```

We miss a class hierarchy for writing

```
 CREATE TYPE Geo_Object
  (Name VARCHAR,
   Coord GeoCoord,
   MEMBER METHOD Distance (other IN Geo_Object)
                 RETURNS NUMBER);
 City :: Geo_Object;
 Mountain :: Geo_Object;
```

```
SELECT c.distance(m)  % Geo-Object-level
FROM City c, Mountain m
WHERE c.name='Freiburg'
  AND m.name='Feldberg';
```


Lecture Information Systems

Object-Relational model

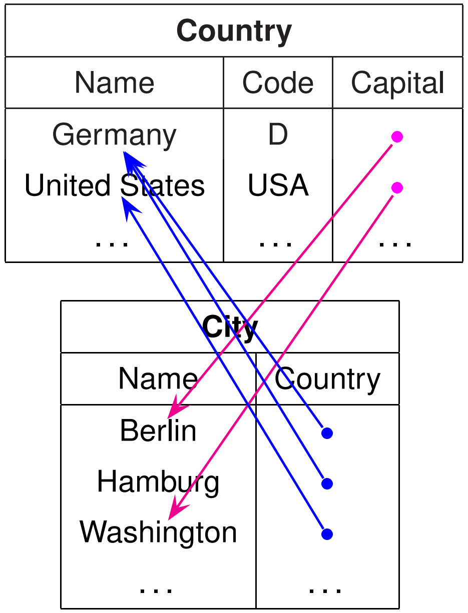
```
CREATE OR REPLACE TYPE City_Type AS OBJECT
    (Name VARCHAR2(35),
     Country REF Country_Type);

CREATE OR REPLACE TYPE Country_Type AS OBJECT
    (Name VARCHAR2(32),
     Code VARCHAR2(4),
     Capital REF City_Type);

CREATE TABLE City OF City_Type;
CREATE TABLE Country OF Country_Type;

INSERT INTO City_ObjTab
  SELECT 'Berlin',REF(c)
  FROM Country_ObjTab c
  WHERE c.Country_Code = 'D';
```

Object-Relational model



Storage: object tables contain an additional column that contains the objectidentifier, which is also used in REF-columns:

country(oid$_{Germany}$, “Germany”, “D”, oid$_{Berlin}$)
country(oid$_{USA}$, “United States”, “USA”, oid$_{Wash}$)
city(oid$_{Berlin}$, “Berlin”, oid$_{Germany}$)
city(oid$_{Hamburg}$, “Hamburg”, oid$_{Germany}$)
city(oid$_{Wash}$, “Washington”, oid$_{USA}$)

Lecture Information Systems

Objects, Object References

- Object-oriented data model can be represented as a network/graph
  $\rightarrow$ *navigation* by path expressions:
  $o_{belgium}$.capital.province.population
- In non-full-fledged object-oriented representations, explicit *references/reference-attributes* are used (cf. object relational model).

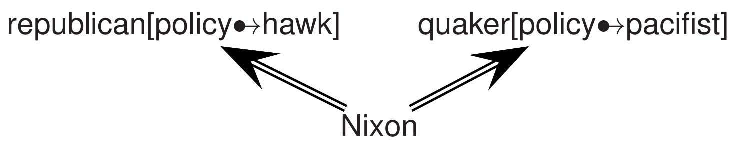
```
SELECT Capital FROM Country
```
% results in an OID, *not* in an object

```
SELECT DEREF(Capital) FROM Country
```
% results in an object of type City_Type

```
SELECT Capital.Name FROM Country
```
% results in a String
% automatical dereferencing in path expressions

Class Hierarchy: Organized Inheritance

- Common concept in programming languages:
  strict trees: each class has a unique direct superclass.
  Inheritance is unambiguous.
- for knowledge representation necessary: directed acyclic graph: a class may have several direct superclasses.
  - cumulative inheritance from all direct superclasses
    AmphiCar is a subclass of *roadVehicle* (attribute wheelbase) and *ship* (attribute grossTonnage).
  - Problem: multiple inheritance, conflicts
    $\Rightarrow$ conflict resolution strategies for multiple inheritance.



What is Nixons *policy* ??

Important, but not possible in all OODMs:

- object can be a member of several classes (student/employee)
- object can change its class (student → employee)

Class Hierarchy

X is a news reader in “News at Noon”.
A news reader at “News at Noon” is a news reader
A news reader is a radio employee.
News reader is a profession.

X is a radio employee.
X is a job.
??

$\Rightarrow$ distinction between “is-a” and subclass relationship!

- Problem: news reader is mentioned as an object, and as a class.

X **:** news reader of “News at Noon”.
news reader at “News at Noon” **::** news reader.
news reader **::** radio employee.
news reader **:** job.

X **:** radio employee.

[in ObjectiF-Manual S.50]

Properties of Classes vs. Inheritable Properties

- object-oriented modeling: distinction between properties of a class (class regarded as an object), and properties that can be inherited to instances of the class.
- statement about a “normal” element of a class:
  usually, a news reader earns 2800DM.
  news reader[income $\bullet\!\!\rightarrow$ 2800DM].
  X:news reader[income $\rightarrow$ 2800DM].
- statement about a class (aggregations etc.):
  news readers earn 3000DM at average.
  news reader[averageIncome $\rightarrow$ 3000DM].
- object-oriented programming (Java):
  `static`-declaration: methods, that are applied to classes vs. “normal” methods that are applied to instances (cf. JDBC: DriverManager).

Equality

- There can be multiple different objects with the same value at a given timepoint.
- distinction between *identity* and *equality* of objects.
  - Two objects are *identical* if they have the same object-identifier.
  - Two objects are *deep-equal* if they yield the same values after recursively dereferencing/navigation along their reference attributes. If they yield the same values already without dereferencing, they are called *shallow equal*.

Deep Equality

Exercise 4 (Deep Equality)

a) Determine for each of the following “databases” the set of objects that are deep-equal (hint: use a graph representation of the objects). Strings are regarded as elementary objects that are equal if they are syntactically equal.

1. objects have only scalar attributes:

   i. o1[name→ “a”,next→o2], o2[name→ “b”,next→o3], o3[name→ “c”], o4[name→ “b”,next→o5], o5[name→ “c”].

   ii. o1[name→ “a”,next→o2], o2[name→ “a”,next→o1].

   iii. o1[name→ “a”,next→o2], o2[name→ “b”,next→o1], o3[name→ “a”,next→o2].

2. objects also have set-valued attributes:

   i. v1[name→ “a”,next→→{v2,v3}], v2[name→ “b”,next→→{v4}], v3[name→ “a”,next→→{v2}], v4[name→ “a”,next→→{w2}], w1[name→ “a”,next→→{w2,w3}], w2[name→ “b”,next→→{w4}], w3[name→ “a”,next→→{w2}], w4[name→ “a”,next→→{v2}].

   ii. Same as (i.), additionally v1[name→ “a”,next→→{v2,v3,v5}], v5[name→ “a”,next→→{v2}].

   iii. Consider a water molecule (H-O-H), which objects of it

are deep-equal?

b) Given for (1) and (2) an (inductive) definition of deep equality of the form “object A and object B are deep-equal, if for all their attributes . . . ”. Use negation. If you have knowledge in Logic Programming: encode the definition by Prolog-style rules (deepEqual(A,B) ← . . . and notDeepEqual(A,B) ← . . .).

Solution Exercise 4

... first, the theory part:

b) for scalar attributes: Two objects o1 and o2 are deep-equal, if

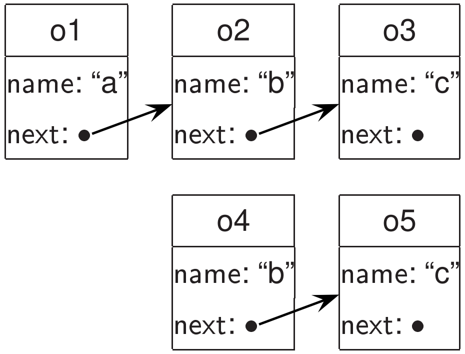
- for all attributes m that result in values of a base domain (string, integer, boolean; i.e., "non-reference-attributes"): $o1[m \to x]$ if and only if $o2[m \to x]$.
- for all reference attributes m: $o1[m \to x]$ if and only if $o2[m \to y]$ and x and y are deep-equal. $\Rightarrow$ recursive definition.

Equivalent: two objects o1 and o2 are deep-equal, if for all sequences m1,m2,m3,...,mn of method applications such that $o1.m1.m2.m3.\ldots[mn \to x]$ results in an atomic value x, also $o2.m1.m2.m3.\ldots[mn \to x]$.

- for set-valued methods, the definition must be adapted: Two objects o1 and o2 are deep-equal, if for each x such that $o1[m \twoheadrightarrow x]$, there is a y that is deep-equal with x and $o2[m \twoheadrightarrow y]$ holds.

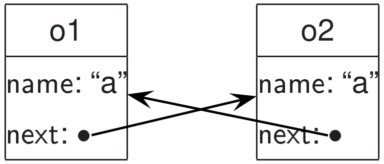
Solution to Exercise 4

a) 1.i.



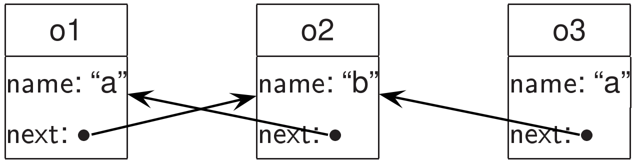
o3, o5 and o2, o4 are deep-equal.

ii.



o1 and o2 are deep-equal.

iii.

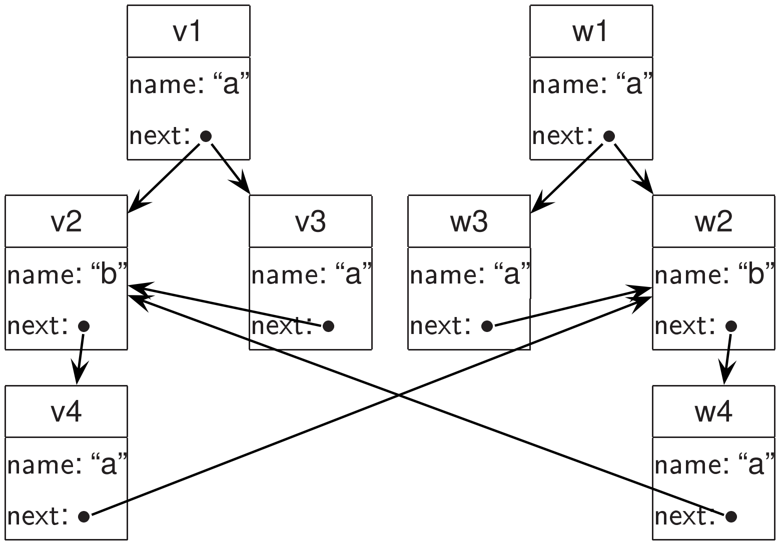


o1 and o3 are deep-equal.

Solution to Exercise 4

2. objects have also set-valued attributes:

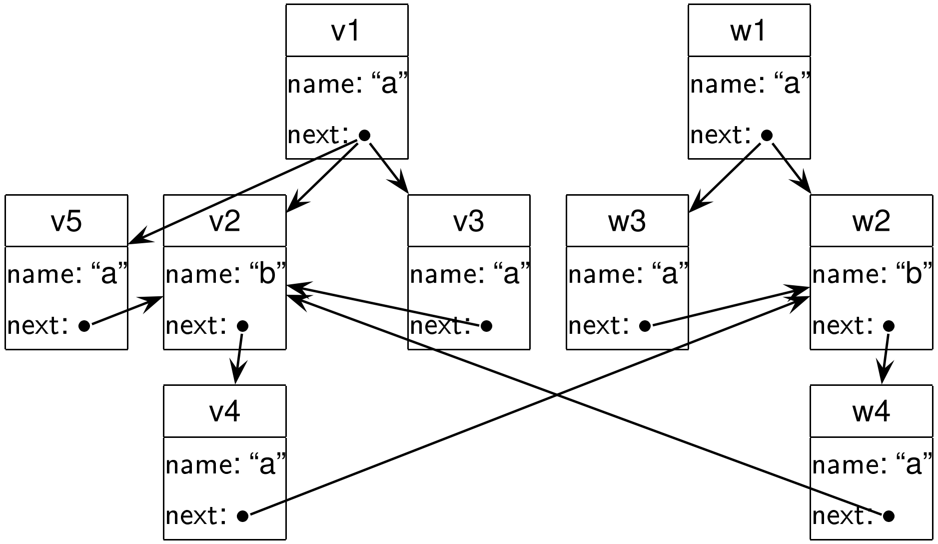
i.



each pair $\{v1, w1\}$, $\{v2, w2\}$, $\{v3, w3\}$, $\{v4, w4\}$ is deep-equal.

Solution to Exercise 4

ii.



each pair $\{v1, w1\}$, $\{v2, w2\}$, $\{v3, v5, w3\}$, $\{v4, w4\}$ is deep-equal.

iii. Both H-atoms are deep-equal. They have the same roles in the atom, and cannot be distinguished.

Solution to Exercise 4

b) cf. *Serge Abiteboul and Jan Van den Bussche: "Deep Equality Revisited"; Proc. DOOD'95 (Deductive and Object-Oriented Databases), Springer LNCS 1013*.

only scalar attributes

```
% different elementary objects:
ndeq(X,Y) :- X:basic, Y:basic, not X = Y.
ndeq(X,Y) :- ndeq(Y,X).

% different class memberships
ndeq(X,Y) :- X:dom, Y:dom, X:C, not Y:C.
ndeq(X,Y) :- X:dom, Y:dom, X::C, not Y::C.

% non-deep-equal attributes (method results)
ndeq(X,Y) :- X:dom, Y:dom, X[M->V], not Y.M[].
ndeq(X,Y) :- X:dom, Y:dom, X[M->V], Y[M->W], ndeq(V,W).

% everything that cannot be proven to be different,
% is deep-equal:
deq(X,Y) :-  X:dom, Y:dom, not ndeq(X,Y).
deq(X,X) :-  X:dom.
```

$\rightarrow$ decidable by *stratifed* negation (first compute `ndeq`, then `deq`).

Solution to Exercise 4

Set-valued Attributes Here it gets more complicated: o1[next$\twoheadrightarrow${a1,a2,a3}] and o2[next $\twoheadrightarrow${b1,b2,b3}]. o_1 and o_2 are deep-equal, if (sets are unordered!!) a permutation $\pi : \{1,2,3\} \rightarrow \{1,2,3\}$ exists, such that a_i and $b_{\pi(i)}$ are deep-equal.

```
% non-equal if for some result there is no
% "matching" result for the other object
ndeq(X,Y) :- X:dom, Y:dom, X[M->>V],
             not X[matched@(Y,M)->>V].

% result matches with deep-equal results
X[matched@(Y,M)->>V] :- X:dom, X[M->>V],
                        Y:dom, Y[M->>W], deq(V,W).
```

These rules depend cyclically (negative) on each other:

$$\mathsf{deq} \xleftarrow{\neg} \mathsf{ndeq} \xleftarrow{\neg} \mathsf{matched} \longleftarrow \mathsf{deq}\ .$$

$\rightarrow$ not expressible with stratified negation (no ordering what must be computed first).

$\rightarrow$ "Well-founded" semantics.

(online in similar form at `http://www.informatik.uni-freiburg/~dbis/florid.html`)

... can we have the same in coloured and clickable?

Graphical Representation of Object-oriented Modeling

UML (Unified Modeling Language)

- UM (*Unified Method*; Booch & Rumbaugh, 1995), then together with Jacobson (Use Cases) UML
- UML1.0 submitted 1997 to OMG (Object Management Group) for standardization.
- June 1999: UML 1.3
- http://www.omg.org/uml

- supported by several CASE-Tools (Computer Aided Software Engineering):
  - ObjectiF/ActiF
  - Rational Rose
  - ...

UML

- graphical modeling language
  - specification, modeling, visualization, documentation
- different abstraction levels
- different *models* that have certain relationships with each other
- diagrams for visualization: *views* of the model(s)
- the modeling is described by several diagrams, each diagram describes one *aspect* of the system to be developed ("sub-plans"):
  - usecase diagram: functionality from the user's point of view (target specification; german: "Pflichtenheft")
  - activity diagram: coarse modeling of dynamic aspects
  - class diagram: static modeling
  - interaction diagram: sequence and collaboration diagram: detailed dynamic modeling
  - state diagram: static + dynamic.
  - implementation diagram: components diagram (implementation dependencies) and distribution diagram (distributed system).

⇒ these techniques are not orthogonal, but generate an (intended) redundancy.

Class Diagrams: Classes and Objects

- classes and objects are represented as rectangles,
- signatures of the attributes, relationships and operations are given,
- for classes, initial values and assertions can be given.

class {**general properties**}
`attr_name[lo..hi]:type = initialValue {assertion}`
<u>`class_attr_name`</u>`:type = initialValue`
`/derived_attr_name:type {formula}`
`...`
`op_name(param: typ = default):type {assertion}`
`...`

- common properties: persistent/transient, abstract class, ...
- attributes and operations of classes (oop: static) are underlined
- derived attributes (/...)
- cardinalities ([interval])
- visibility: + = public

    - = private (only for methods of the class)

    # = protected (only for class and its subclasses)

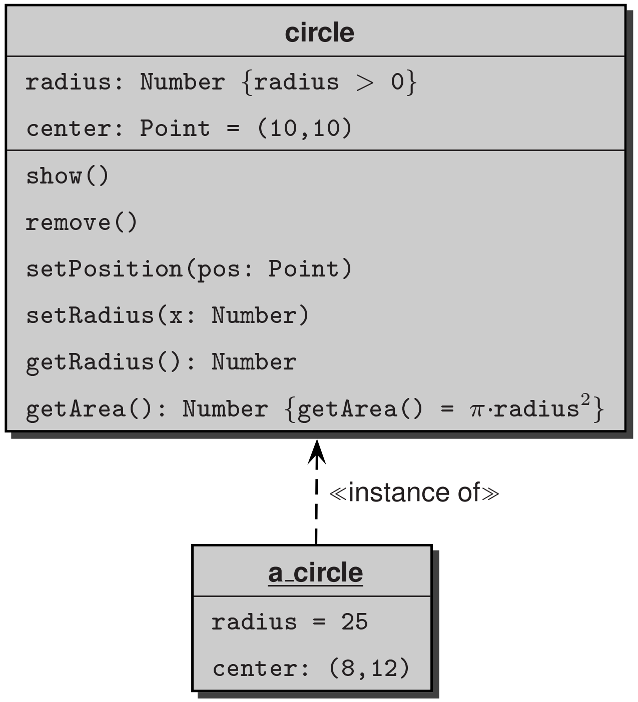
Classes: Example

circle {**persistent**}
`-radius[1]: Number` {`radius` $>$ `0`}
`-center[1]: Point = (10,10)`
`+`<u>`color`</u>`[0..1]`
`+/circumference: Number` {`Number =` $2 \cdot \pi \cdot$`radius`}
`show()`
`remove()`
`setPosition(pos: Point)`
`setRadius(x: Number)`
`getRadius(): Number`
`getArea(): Number` {`getArea() =` $\pi \cdot \mathtt{radius}^2$}

- circles are persistent
- all circles have the same color (method of the class)
- circumference as a derived attribute

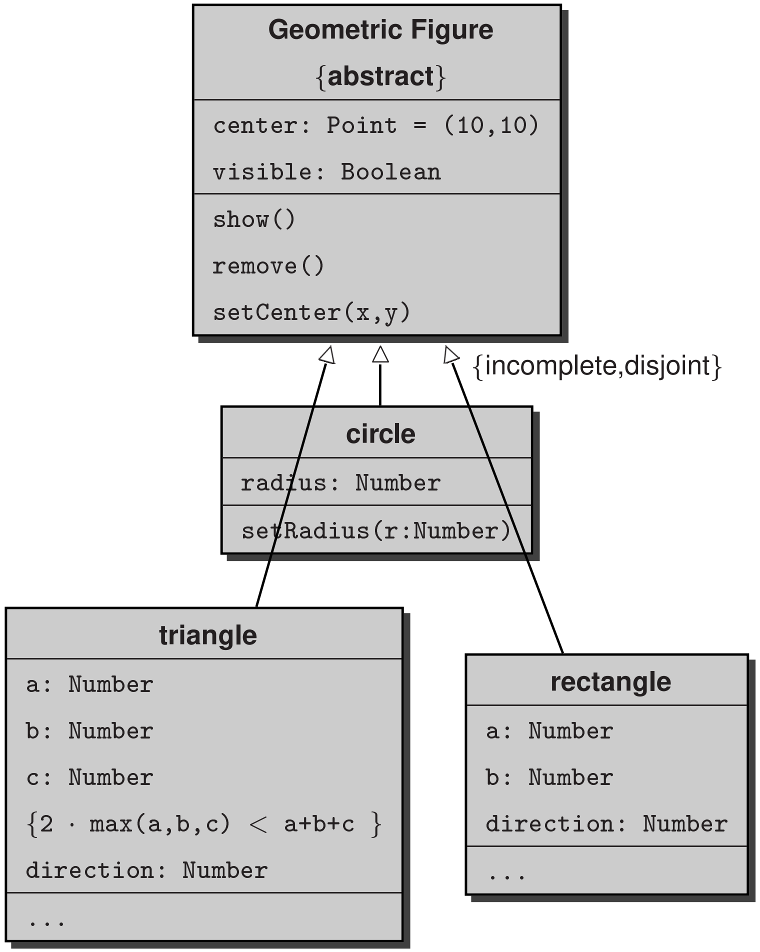
Objects

- for each object, the attribute values are given:
- for each object, its name is underlined;



Short notations: without data, or even only anonymous:

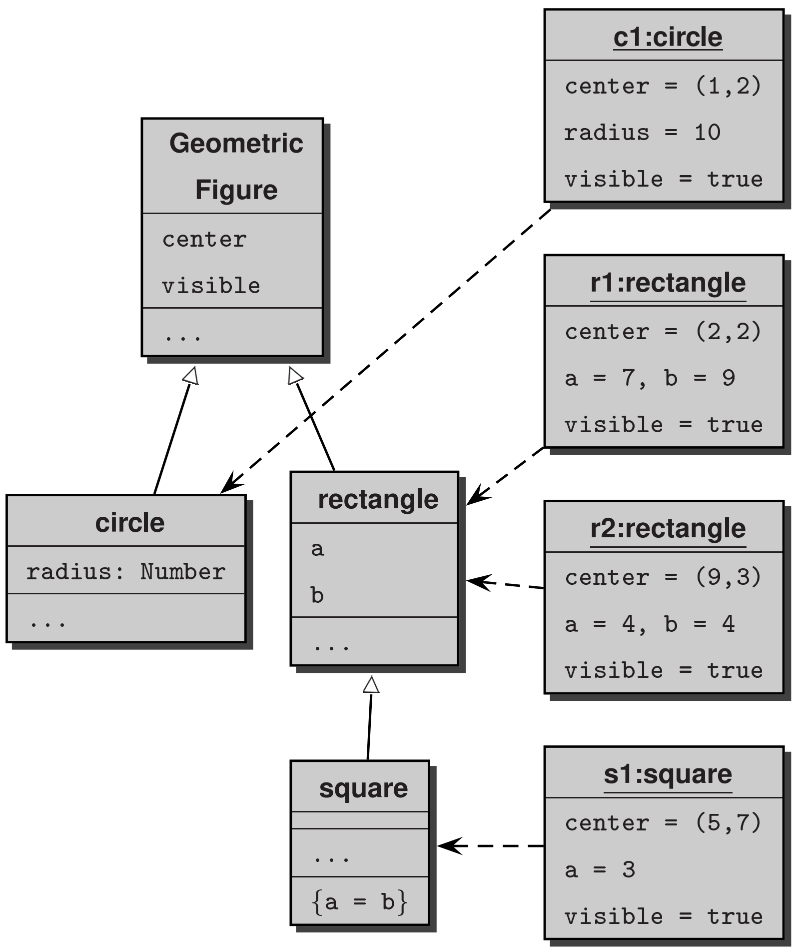
Class Hierarchy



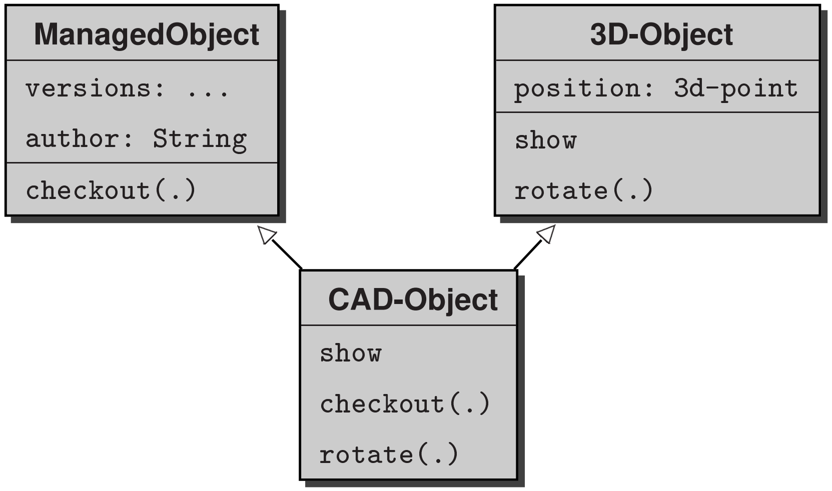
Class Hierarchy

- More detailed specification by keywords:
- *overlapping*:
  - object can be an instance of several classes (only if this is allowed)
  - lower subclasses can be derived from multiple subclasses (only if multiple inheritance is allowed)
- *disjoint*: (default)
- *complete*: all subclasses are completely defined at the beginning
- *incomplete*: during the modeling process, additional subclasses can be defined.
- complete does not mean that each object must at least belong to one subclass!!
  $\Rightarrow$ the distinction between generalization and specialization cannot be described.

Class Hierarchy



Multiple Inheritance



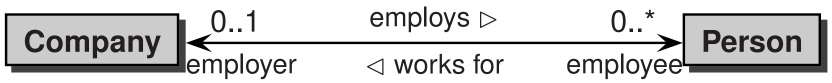
- multiple inheritance required,
- inheritance conflicts,
- allowed in UML, but in general not in CASE-tools due to problematic realization in the implementation.

Relationships

- ER-model: *Relationships*

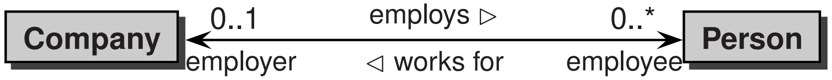
UML: Associations

- An *association* is a *directed* relationship between different objects.
- Often, also the opposite direction gets a name (employs/works_for)
- Associations have a name, and can be augmented by *multiplicities/complexities* and *role names*.

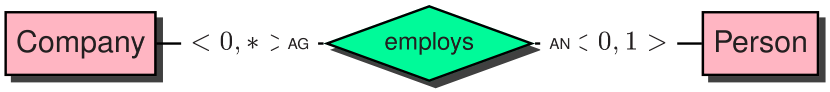


- derived relationships: /name
- arrow: supported navigation direction
- further information: ordered, sorted, frozen, add-only

Relationships

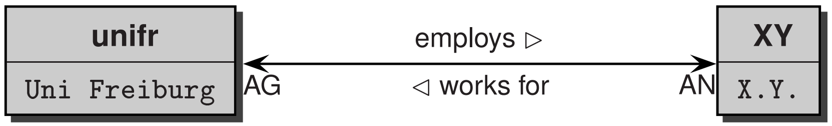


- ... similar to the ER-model.



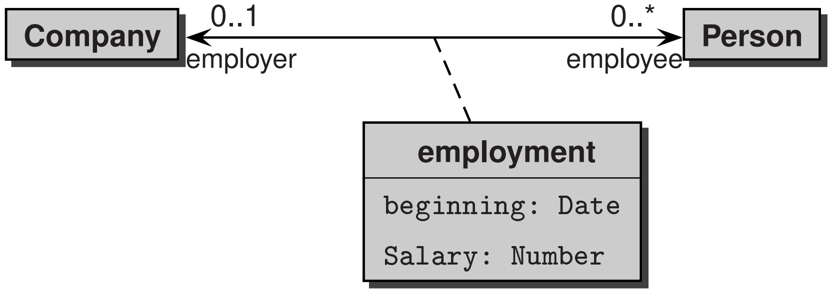
- ... different position of the multiplicities/complexities!
- Bidirectional relationships can be specified,
- these details only for *binary* relationships,
- less support for non-binary relationships (see later).

the same on the object level:



Association classes

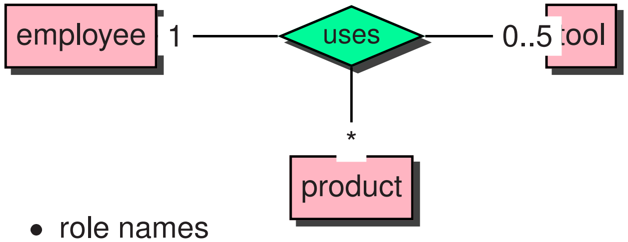
- Associations (relationships) can also have attributes (cf. relationships with attributes in the ER-model)
- Associations can be *class-valued*:



- Association classes can be involved in associations with other classes.
- Generalization can also be applied to associations and -classes:
  employer employs person / university employs staff members ...

Aside: Non-binary relationships in the ER-model

- UML similar to ER-model:

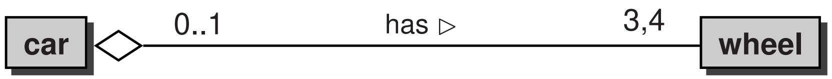


- role names
- Multiplicity: specifies how many objects of an entity type may be involved with each combination of the participating entity types in this relationship:
  - for a given pair (product, tool), only one employee can perform that task.
  - when working at a product, an employee can use up to 5 tools.
  - An employee can work with a given tool at arbitrary many products.
- can also be defined as association classes (entity types).
- no notion of navigation directions (in some sense induced both directions by foreign keys).

Special Types of Relationships

Aggregation

- Part-whole-relationship:

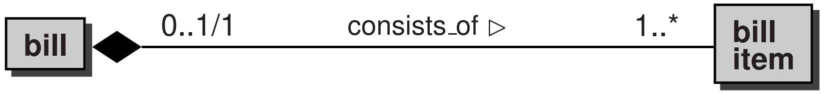


Composition

- Existence of the parts depends on the existence of the whole,

⇒ part can only be a part of *one* whole.

A bill consists of several bill items.

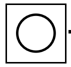


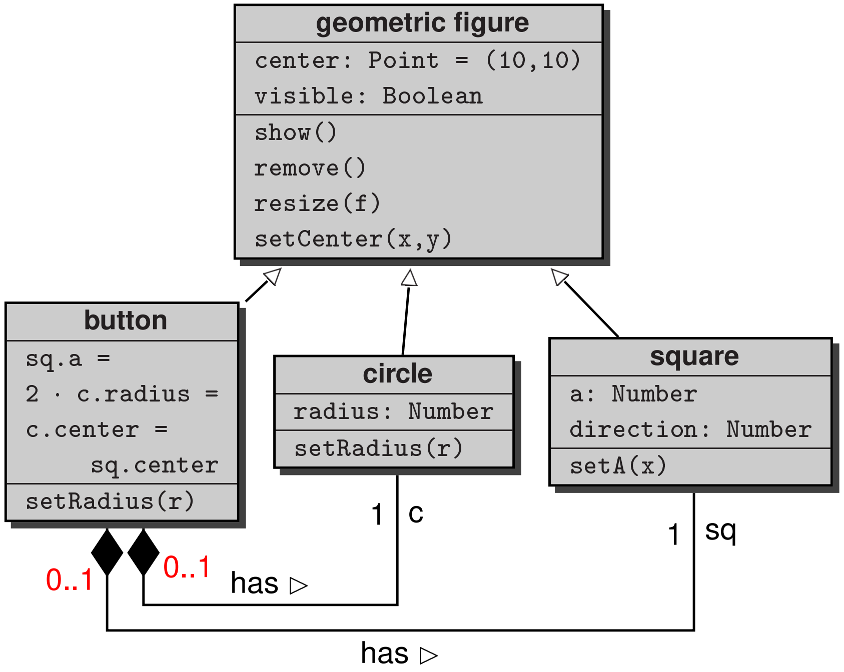
Exercise 5 (Multiplicities for Aggregation/Composition)
What statements can be made about the multiplicities at the side of the superordinate object (i.e., where the diamond is drawn) in case of aggregation/composition?
What does 0..1 mean at this position?

Solution of Exercise 5

- composition: on the side of the superior object there must be a “1” or “0..1” (dependence of the subordinate object from the superordinate one).
  Note that the standard is ambiguous wrt. “0..1”, see [Kappel, S.37].
  - with “0..1”, an object may exist independently after its creation and can be bound later (or never) to a whole. In that case, its existence then depends on the existence of the whole.
- Aggregation: a “1” at the side of the superordinate object makes no sense – then, the relationship is preferably modeled as a composition.
- Aggregation with 1..*: the subordinate object must be associated to a superordinate one, but it is not dependent on a certain one (it can change from one to another).

Example from [Oestereich]

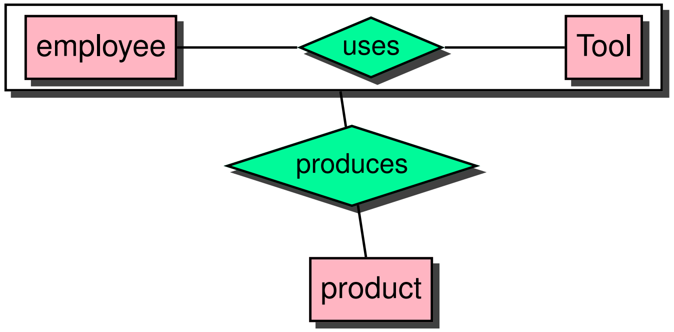
An application uses figures “button”, that look like .



- not every circle/square belongs to a button, thus 0..1
- (cf. Exercise 5)

Aside: Aggregation in the ER Model

UML's Aggregation is not the same as *aggregation* in the ER-model:



This ER functionality corresponds to the definition of *association classes* that can again be involved in associations with other classes.

Aggregrations

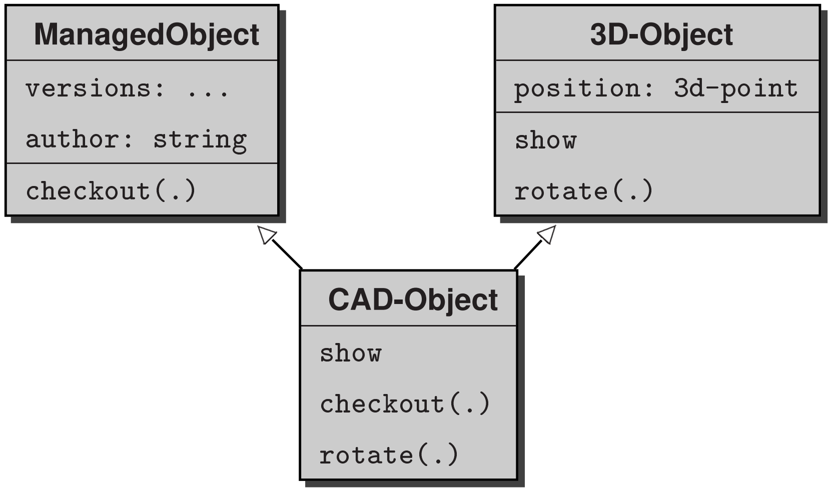
- The "whole" acts as a substitute for its parts, i.e., external objects communicate only with the "whole", that in turn invokes operations of the parts:
  button.show() is mapped to button.c.show() and button.s.show():

```
class button extends GeomFigur
{circle c;
 square s;

 void show()
 {c.show;
  s.show;}

  :
}
```

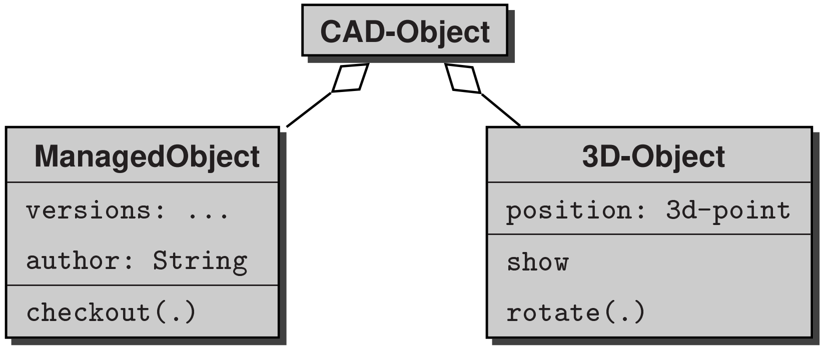
Aggregration instead of Inheritance



- multiple inheritance required

Aggregration instead of Inheritance

Other point of view: a whole consists of several parts, that each has a certain functionality and belong to suitable classes:

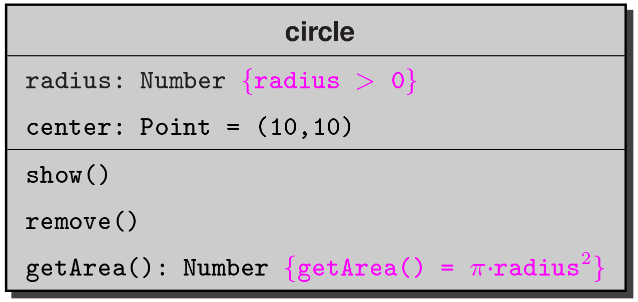


... see later for “message passing” (Slide 131).

Assertions in UML

An **assertion** is an expression that constrains the possible contents, states, or semantics of model elements (class, relationship, . . .).

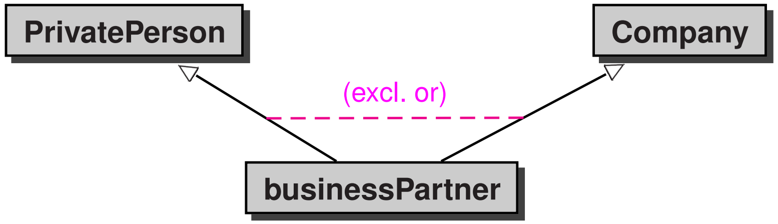
- Assertions, that concern only a single model element are added in curly brackets to the model element.



- Assertions are formulated in *OCL (Object Constraint Language)*, based on first-order logic:
  $\forall t \in$ notification:
  t.receiver $\subseteq t$.appointment.participant
- Invariants, pre- and postconditions, dependencies ...

Assertions

- undirected dependencies between two model elements are represented by a dashed line:

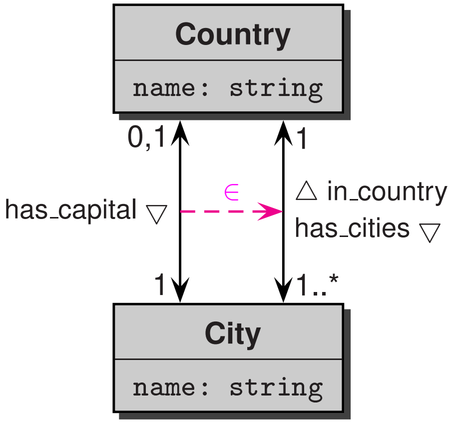


Remark:

- due to the exclusive-or, there is no inheritance conflict!
- the given relationship is not a “true” subclass relationship.
- often not clear whether a “subclass” should be modeled, or whether a simple “is-a” association would be better (see also *delegation*).

Assertions

- directed dependencies between two model elements are represented by a dashed arrow from the dependent object to the indepentent one.



- Assertions are supported only in UML, not in ODMG. There they are covered only by the implementation.

Exercise: ER-model vs. UML

Exercise 6
give a modeling of the following situation

- as ER-model, and
- in UML:

There are several spatial objects: square stone (dt.: Quader), cubes, pyramids, tetraeders, octaeders etc. Each of them has surfaces of certain types. Surfaces are limited by edges; each edge is given by two points (that belong to at least three edges).

Define operations that are needed to represent these objects in a CAD-System. Describe (non-formally) the communication between the participating objects of the modeling.

Exercise 7 (UML: Soccer-Championship)

An information system is required for being able to access the current information. Here, a suitable IS should be modeled that provides “*Online*”-information, i.e., at each timepoint the current state should be accessible:

- immediately when there is a goal, or a yellow card, the corresponding information is added to the database and all data is automatically updated.
- when a game is finished, some data has to be updated.
- after finishing the group stage, the matches for the quarterfinals must be determined from the current rankings.

Discuss what notions must be modeled, which attributes must be modeld, and what relationships exist.

Especially, consider the following things:

Group A,

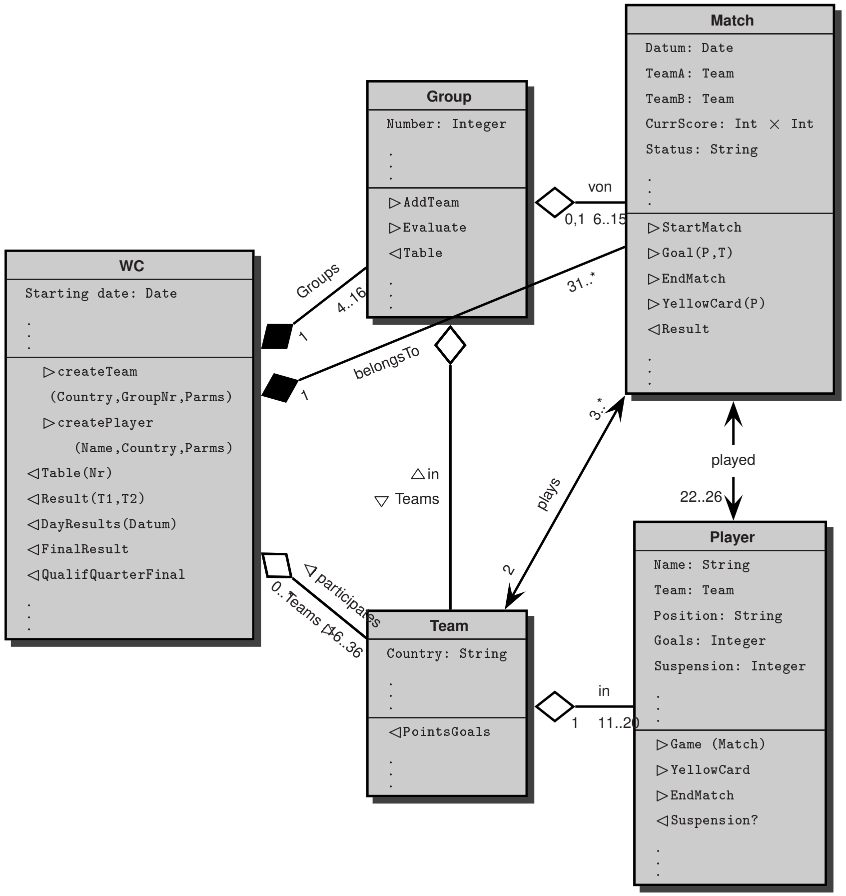
the match Germany-Cameroon,

the national team of Corea,

S.Effenberg,

each of the semifinals.

Soccer-WC: Coarse Modeling

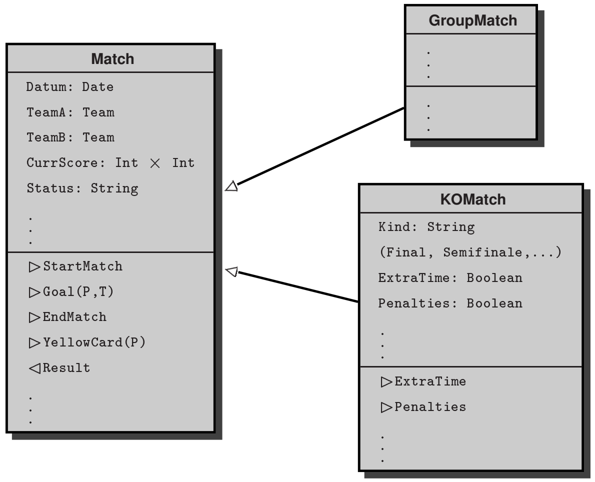


Soccer-WC: Coarse Design – Comments

- Existence of groups and matches depends on the existence of superordinate objects. In contrast, a team may exist without being associated to any WC or group.
- distinguish between GroupMatches and FinalMatches (Quarterfinal etc) [see next slide]
- persons: players are persons, additionally there are trainers, referees, functionaires etc.
- assertion: a player can only play in a match if his team participates in this match.
- assertion: a team can only play in a GroupMatch if Team and GroupMatch belong to the same group.
- Assertions about cardinalities: a team plays at least against each other team of the same group.

Types of Matches

There are two kinds of matches: GroupMatches whose result contributes to the group ranking, and final matches (from the QF on).



Kinds of matches

- the relationship Group-Match exists only for Groupmatches, other matches are Q/S/Final matches; can be specified by an exor-assertion.
- the "behavior" of these matches is different:
  - final matches can be extended and then come to a decision in a penalty shootout,
  - The method "▷Goal" is implemented differently in both classes of matches (effects on the group ranking, or on subsequent finals).
- the set-valued relationship Group-Teams is implemented as a set-valued relationship by a suitable collection. Here, an ODMG-class can be defined based on *Set* that outputs a ranking table of its elements.

Lecture Information Systems

Access to Objects

Objects must provide access methods for set-valued methods for selecting an item:

- Group A: WC1998..Group("A").
- the national team of Cameroon: WC1998..Teams("Cameroon")
- GroupMatch D-CAM: WC1998..Team("D").GroupMatch_vs("CAM")
- S.Effenberg: WC1998..Player("Effenberg")
- A semifinal: WC1998..Semifinals yields both semifinal matches.

So far, everything is static. This information can be stored in an object-oriented or object relational database system (e.g. Oracle).

ODMG-Standard

A standard for the specification of object-oriented systems that consists of different sublanguages that are close to actual implementation languages.

Components of ODMG 3.0

- object specification languages/data model
  - Object Definition Language (ODL),
  - Object Interchange Format (OIF)
- Object Query Language (OQL) – based on ideas of SQL
- C++/Smalltalk/Java Language Binding
  specifies the handling of persistent objects in the target languages.

Lecture Information Systems

ODL: Object Definition Language

- Data Definition Language for object types:
- no programming language, but a language for definition of object specifications,
- extends IDL (Interface Definition Language) from the ODMG/CORBA-standard,
- characterizes object types (class hierarchy, properties and relationships)

Data Types: Literals

- literals have no OIDs.

Atomic Literals

- long, short, unsigned long/short, float, boolean,
- char, string,
- enumeration {...} ("Type generator")
  Z.B.
  enum Weekday {Sunday, Monday, ..., Saturday}
  enum Month {January, February, ..., December}

Structured Literals

- predefined types: date, interval, time, timestamp
  (in addition to *full object types* Date, Interval, Time, Timestamp)
- user-defined structure types, e.g. address

```
struct  GeoCoord {
        real         Longitude;
        real         Latitude; }
```

Collection Literals

- set<t>, bag<t>, list<t>, array<t>, dictionary<t>
  (in addition to *full object types* Set, Bag, List, Array, Dictionary)

Classes

... are used for defining complex object types and to classify objects.

Classes in the object model define the *signature* of their instances (see later for details) (the implementation does not belong to the object model):

```
class <name> {
      <attribute-defs>;
      <relationship-defs>;
      <operation-defs>;
      }
```

<attribute-def> ::= attribute <domain-type> <attribute-name>

```
class City {
  attribute string name;              % attributes ...
  attribute number population;
  attribute GeoCoord coordinates;
  relationship Country in_country;   % ... and relationships
  }
```

Relationships

- relationships are specified with the definition of classes.
- in UML/ODMG only *binary* relationships allowed.
- *bidirectional* relationships can be specified.
- one-to-one / one-to-many / many-to-many-relationships.

class <name> {
<attribute-defs>;
<relationship-defs>;
<operation-defs>; }

<relationship-def>::= relationship <target_of_path>
<relationship-name>
inverse <domain-type> :: <relationship-name'>
<target_of_path>::= <domain-type> |
<collection type> <<domain-type>>

- <collection type> for -to-many-relationships
- der <domain-type> der inverse-relationship is the same as the basic-<domain-type> in <target_of_path>.

Relationships

class <name> {
<attribute-defs>;
<relationship-defs>;
<operation-defs>; }

<relationship-def>::= relationship <target_of_path>
<relationship-name>
inverse <domain-type> :: <relationship-name>
<target_of_path>::= <domain-type> |
<collection type> <<domain-type>>

```
class Country {attribute string name;
               relationship City capital
                 inverse City::is_capital_of;
               relationship set<City> has_cities;
                 inverse City::in_country; }

class City {attribute string name;
            relationship Country in_country;
              inverse Country::has_cities; }
```

Relationships

- relationships define *access paths*, e.g. Germany.capital.
- ODBMS is responsible for maintaining referential integrity:
  when an object is deleted, all access paths to it must be deleted.
- note: ORACLE8 does not do this automatically, but allows to check for lost references by

  ... WHERE <attr> IS DANGLING.

ODMG: Object-Valued Attributes

Attributes can also be object-valued:

```
class Country {
    attribute string name;
    relationship City capital;
    }
```

or

```
class Country {
    attribute string name;
    attribute City capital;
    }
```

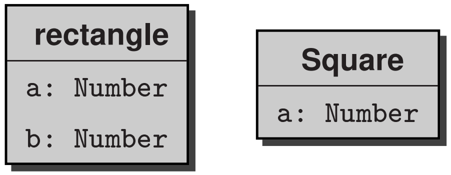
- object-valued *attributes* create a reference, *without* defining an access path. In this case, referential integrity is not guaranteed by the OODBMS.
- only applicable for *unidirectional* relationships,
- ODMG: object-valued attributes are not regarded as “true relationships”.

Discussion

- Attributes and relationships are no *"first-class"*-things in the modeling, i.e.,
  - attributes are no objects, and thus have no object identity,
  - it is not possible to define attributes of attributes, or relationships between attributes,
  - ... would sometimes be useful for storing meta information about attributes.

Structural hierarchies (cf. Slide 33)

- class hierarchy: *semantical* hierarchy,
  - subset relationship: subclasses represent specialized concepts, compared with their superclasses: every square is a rectangle.
  - point of view of the modeling.
- *type hierarchy*: *refinement* or *extension* of attributes and behavior:



  Here, "rectangle" is an "extension" of "square" by the attribute *b*; "rectangle" *extends* the behavior/signature of "square".

- *implementation hierarchy* (OO-programming languages):reuse of the implementation of a class for another.

In the *type hierarchy* and *implementation hierarchy* it makes sense to use the definition of "square" as a base for the definition of "rectangle".

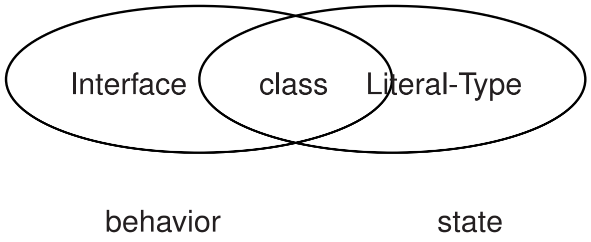
$\Rightarrow$ different hierarchies!

Structural Hierarchies in ODMG

The ODMG-Standard is closer to the implementation than UML. In some sense, both the class hierarchy and the type hierarchy are supported and distinguished.

Class/Type Specification:
abstract, implementation-independent description of properties, operations (and exceptions) (*signature*).

In addition to the usual "material" classes (City, Country, Person) and literal types there are *interface classes*. These specify *only* the *behavior*:



important: implementation-classes of OO-programming languages are something different than classes of the object model!!

The presentation in [Heuer, p.445] does not coincide with the standard!

Interfaces

- *Interface definition*: describes only the behavior (operations):

  ```
  interface <Name>: <supertypes-list>
        { <interface specification>}
  interface i_Person {
        number age()}
  interface i_Employee : i_Person {
        number calculate_paycheck(in Date date)}
  ```

- it is not possible to create actual instances of interfaces (obvious, since interfaces specify only the signature of a behavior – a potential instance could not have any state).
- instances are instead generated of (sub)classes that extend the declarations of the interfaces with a state signature.

Classes

- *class definition*: description of behavior (operations) and state (attributes and relationships):

  class <Name>
  extends <superclass >
  : <supertypes-Liste>
  { <specification> }

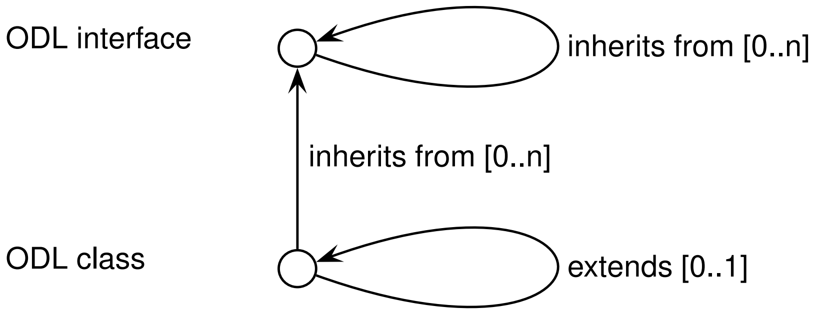
  ```
  class Person : i_Person {
      attribute string name;
      attribute Date birthDate;
      void marry (IN Person partner); }
  ```

- classes extend interfaces with further operations and an (abstract) state ([ODMS schema-definition], possibly inherited from a superclass)
- classes can then be instantiated directly.
- the implementation of methods can either be given at the interface level (if they are simply mapped to other methods of the objects) or at the level of the specialized subclasses (if they need state information, e.g. circle.show()).
- methods without implementation are called *abstract* methods. Implementations are then given for the subclasses.

  Geometric_Figure.show().

Classes and Interfaces

- a class inherits only from one superclass (extends),
- a class can inherit from multiple interfaces,
- interfaces can also define *derived attributes* (they are not part of the state, but “simple methods”; e.g. age based on birthday)



ODL-classes are mapped directly to instantiable classes of the programming language by the language binding.

Example

```
interface i_Person {
        number age()}

class Person : i_Person {
        attribute string name;
        attribute Date birthDate;
        void marry (IN Person partner); }

interface i_Employee {
        number calc_paycheck()}

class EmployeePerson extends Person : i_Employee {
# read: class EmployeePerson extends class Person
#       with i_Employee-Behavior and ...
        attribute Date hireDate;
        attribute Currency payRate;
        relationship ManagerPerson boss
                        inverse Manager::subordinates}

interface i_Manager {
        ...}

class ManagerPerson extends EmployeePerson : i_Manager {
        relationship set<EmployeePerson> subordinates
                        inverse Employee::boss}
```


Example (Cont'd)

- each ManagerPerson has the attributes name, birthDate, hireDate, payRate, a boss, and subordinates. The behavior is inherited from the interface i_Employee to the subclass EmployeePerson:Employee (is-a), and is then propagated through extends to ManagerPerson.

⇒ only "reuse" of the behavior (signature and possibly implementation).

ODMG: Subclasses/Inheritance of Behavior

- class hierarchy is transitive
- whenever an object of a class/type is required, any object of an arbitrary subclass/subtype is allowed.
- subclasses/subtypes inherit the signature of their superclasses (state and behavior) and supertypes (behavior).
- subclasses/subtypes can redefine signatures of their superclasses/supertypes (*overriding*).
  class Salaried_Employee {...}
  class Hourly_Employee {...}
  redefine *calculate_paycheck*.
- operations that have the same names, but different parameters can be inherited from different superclasses, or can be defined by the class itself.
  ⇒ polymorphism:
  alarmclock.set(13:00);   alarmclock.set(1h);

ODMG: Further Concepts around Classes

- *Extension*: set of all instances of a class.
- keys (multiple keys are allowed)
- assign a name to an extent. This is e.g. used in OQL-queries:
  SELECT ... FROM <extent>

Such things are defined with the class declaration:

```
class <name>
(extent <extent_name>
 keys <set of keys>)
{  <attribute-def>;
   <relationship-def>;
   <operations-def>; }
```

```
class City
(extent Cities
 keys (Country,Name), (Country, zip))
{  attribute string name;
   relationship Country country;
   attribute struct { real Latitude, real Longitude } coordinates;
   . . . }
```

- subset-relationship of extents (transitive)

Class Hierarchy ODMG vs. UML

- The distinction between *interfaces/classes* and is-a/extends on made only in the *ODMG-model* (since this is then directly mapped to an implementation).
- In *UML* there is only the concept *class* with super/subclasses and inheritance; also for relationships.
- The structural hierarchy in ODMG is an interface hierarchy, no class hierarchy in the intuitive sense.

Type Hierarchy vs. Implementation [*class*-]Hierarchy

- C++, Eiffel: only sub*classes* can be sub*types*,
- Java: clean distinction between both hierarchies,
- Smalltalk: no notion of type hierarchy (only a class hierarchy).

Lecture Information Systems

Implementation

The implementation of *object types* contains the state and the behavior:

- Representation: *data structure*, derived from the abstract state specification (according to the chosen *language binding*),
- Methods: *procedure bodies*, derived from the abstract specification of behavior (again, according to the chosen *language binding*),
  - a method can have several implementations: one in each programming language.
- The procedure body can change the state of the host object, and it can invoke operations of other objects.

Literal types do not have a behavior:

- implementation of *literal types* is also given by the *language binding*,
- either to the directly corresponding type in the target language, or by defining new things (e.g. structs: in C++ to structs, in Smalltalk and Java to instances of suitable newly defined classes).

- How are set-valued relationships, n-m-relationships etc. stored?
- what does e.g.
  class Country {relationship set<City> has_cities;}
  mean?

Further Object Types: Collection Types

- also called *container classes*,
- are used for collecting and handling sets of objects,
- typical operations: inserting and removing of objects, membership test, cardinality of the collections.
- are e.g. implicitly used when defining relationships of cardinality "many":

```
class Country {
        attribute string name;
        relationship City capital;
        relationship set<City> has_cities;
            inverse City::in_country;
        }

class City {
        attribute string name;
        relationship Country in_country;
            inverse Country::has_cities;
        }
```


Collection Types

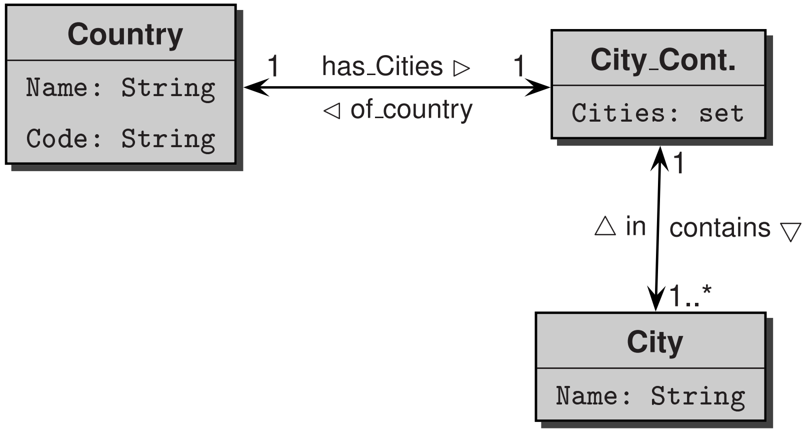
- Collection types (*Collection Objects*):
  All elements of a collection must be of the same type (often an abstract type)

  Depending on the intention, this can be implemented in different ways ... the *idea* of a *container object* is always the same.
- Distinction between *sequential* and *associative* containers.
- *Sequential containers* store the objects in a linear structure;
- *Associative container* store objects together with an identifying key.
- Differences: possibilities to sort, handling of duplicates
  - Set
  - Bag (Multiset)
  - List
  - Array
  - Dictionary

Collections

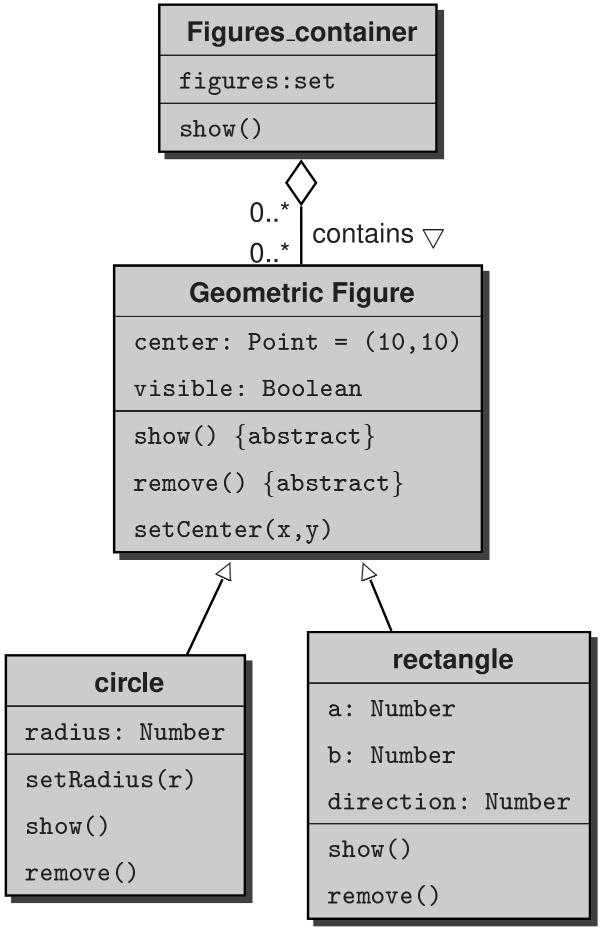
- implicit usage: “x-to-many”-relationships:

- explicit modeling by a container object:



Collections

Often: Collections over abstract classes, that contain elements of different subclasses:



Object Interchange Format (OIF)

- store the database state in one or more files (cf. *export* in ORACLE)
- specification language for persistent objects and their states
- OIF-File contains for each object its type, its attribute values, and its relationships with other objects;
- the database schema (class definitions *and* class hierarchy) is *not* stored in OIF!

Object Interchange Format (OIF)

- Simplest Form: only class membership

  <object> <class> {}
  Germany Country {}
  Berlin City {}

- attribute values are enumerated in the brackets:

  Germany Country {Name “Germany”, Area 356910, ...}

- Structured attributes: nested parenthesized structures

```
struct GeoCoord {
        real  Longitude;
        real  Latitude; }

class City {
        attribute string Name;
        attribute GeoCoord Coordinates;
        relationship Country in_country; }

Berlin City
    {Name "Berlin", in_country Germany,
     Coordinates {Longitude 13.3, Latitude 52.45} }
```

Object Interchange Format (OIF)

- Collections, multi-valued relationships:

```
class Country {
    attribute string Name;
    relationship set<City> has_cities;}
```

```
Germany Country
  {Name “Germany”, Capital Berlin,
   has_cities {Berlin, Frankfurt, Freiburg, ... } }
```

- cyclic references: no problem.

Exercise 8 (Data formats)
Represent the data given in Exercise 3 in OIF.

Solution of Exercise 8

Class Definitions in ODMG

```
class City {
    attribute string Name;
    attribute struct {number longitude;
                      number latitude } coordinates;
    relationship Country in_country; }

class Country {
    attribute string Name;
    relationship set<City> has_cities;
    relationship City capital;
    relationship set<Organization> memberships
      inverse Organization::members;
    Neighborhood relationship here also problematic:
    (no relationship ...)
    attribute set<struct{country neighbor;
                         number length }> borders }

class Organization {
    attribute string Name;
    relationship City seat;
    relationship set<Country> members
      inverse Country::memberships}
```

Solution of Exercise 8

Data in OIF

$o_{brussels}$ City {
  Name "Brussels";
  coordinates { longitude 4.35, latitude 50.5};
  in_country $o_{belgium}$; }

$o_{belgium}$ Country {
  Name "Belgium";
  has_cities $\{o_{brussels}, o_{antwerp}, \ldots\}$;
  capital $o_{brussels}$;
  memberships $\{o_{eu}, o_{nato}, \ldots\}$;
  neighbor {{country o_{france}, length 620}
      {country o_{gemany}, length 167}}}

o_{eu} Organization {
  Name "EU";
  seat $o_{brussels}$;
  members $\{o_{belgium}, o_{france}, \ldots\}$ }

- Up to now: static modeling of an application domain: classes, subclasses, relationships, aggregations, collections ...
- Behavior sketched only indirectly ("what operations are provided by the classes")
- Simply a follow-up model to ER-diagrams ??
- Not yet described: execution of operations, communication/cooperation between objects, message passing.

Behavior

- behavior of objects
- behavior of the systems

Object Behavior

- objects provide operations for performing actions.
- operations are invoked by sending a message.
- message passing is different from function- and procedure calls:
  objects *encapsulate* operations and data; objects can be contacted only via their operations.
- cooperation is organized as *communication* via *messages* between objects (sender and receiver).

Object Behavior

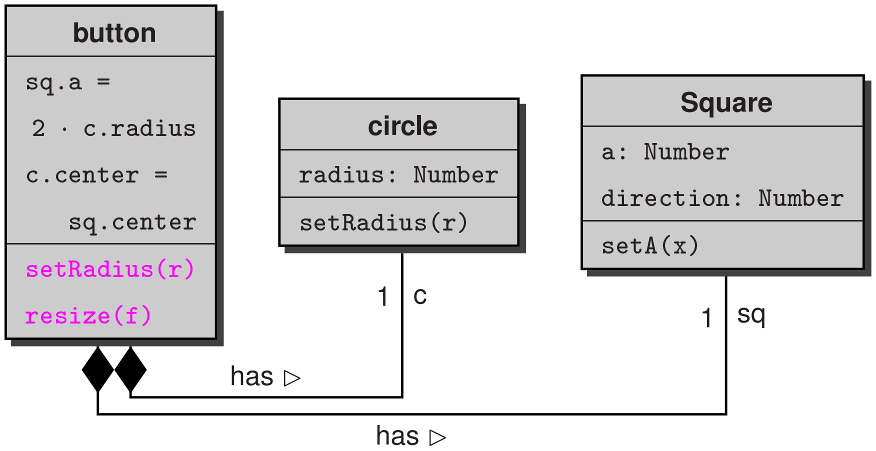
- ODMG: the signature of operations is specified with the classes/interfaces.

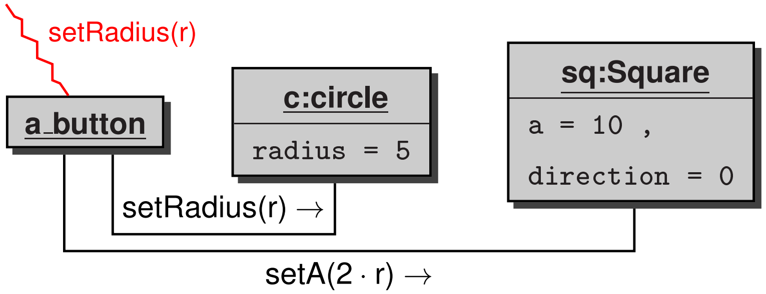
```
class <name>{
      <attribute-def>;
      <relationship-def>;
      <operation-def>;
      }
<operations-def>::= <result-type> <operation-name>
                          (<list-of-parameters-def>)
<parameters-def>::= in|out|inout <domain-type> <variable-name>
```

```
class button {
      void setRadius(in int radius);
      number getRadius();
      void resize(in float factor);
      void show();
      }
```

Object Behavior

- UML: the signature of operations is specified with classes/instances. Cooperation is - among other diagrams – specified in the class diagram (see later).
- Cooperation is based on *associations* and *aggregation relationships*.

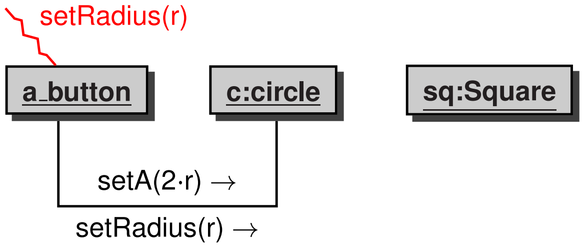




Communication: Delegation

In case that messages to some object are passed without change to another object, this strategy is called *delegation*:

An object does not perform a task by itself, but delegates it to another object (or to several objects).




```
class button extends GeomFigur {
    public void setRadius(int radius)
    {c.setRadius(radius);
     sq.setA(2 · radius); }
    public void resize(float factor)
    {radius := c.getRadius();
     setRadius(radius · factor); }}
```

- often with compound objects
- especially for . . .

Inheritance by Delegation

Delegation can be used for simulating inheritance (cf. Slide 86)

Idea:

- an object belongs to two semantically different superclasses (in most cases, very common ones)

⇒ different direct superclasses needed.
often not allowed `:-(`

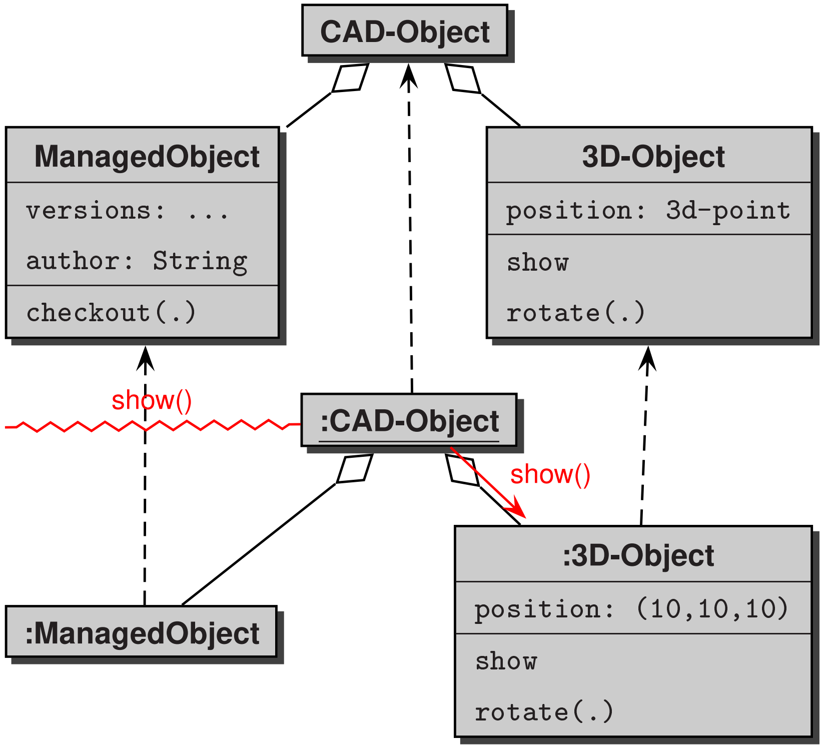
- Das object belongs to only one of these classes, and is substituted [for certain tasks] by another object (that belongs to the other class).
- The second object then plays the role of the first one.

or

- an object consists (by aggregation) of on object of each of the classes and delegates each task suitably
- both subobjects play certain roles of the original object.

Inheritance by Delegation

Example from Slide 86:



- against the user, there is a CAD-object,
- that is substituted internally by the subobjects (*Delegation*),
- 3D-subobject plays the role of the CAD-object.

Object-Oriented Analysis and Design

The notions and circumstances of the application domain are translated into an *abstract model* to be realized in the software project:

- identifying and describing the objects of the application
- relationships and common properties of objects
- grouping of objects that have characteristic properties in common into classes
- describing relationships between classes
- definition of a class hierarchy
- definition and implementation of the functionality of each class (communication, methods)
- combination of the application program by modules

Object-Oriented Analysis and Design

- *Analysis* and *optimization* of (business-)processes.
- relevant workflows are analyzed and described by *use cases*.

Use Cases

- a use case describes *requirements* to the system, but does not describe *how* these should be implemented. A use case can consist of different variants, called *scenarios*.
- connections between use cases described by *use case diagrams*; these can be structured *hierarchically*.
- exceptions and special cases are described in subordinate *secondary use cases* of the normal cases.
- use cases support the description of the *as-is-situation* and make the developers familiar with the application. Terminology and language of the application domain must be explored and fixed in a *glossary*.

Use Case Diagram

- very abstract level,
- use cases describe the relationships (e.g. interaction) between *actors* and the behavior of the system from the point of view of the user.
- structural connections,
- describes activities that should be supported by the software to be developed together with the surrounding environment,
- especially, also activities are described that are not performed by the software system (e.g. an inspection by an expert (that must be organized) where the result is then stored in the system).

Elements:

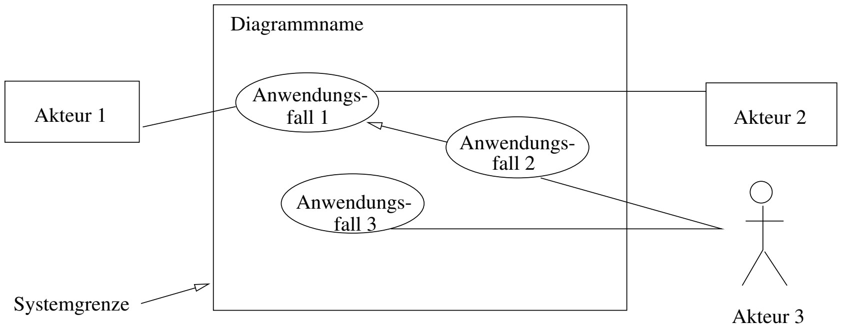
- use cases 

  for each use case, a textual description should be added

- actors 

  inspector

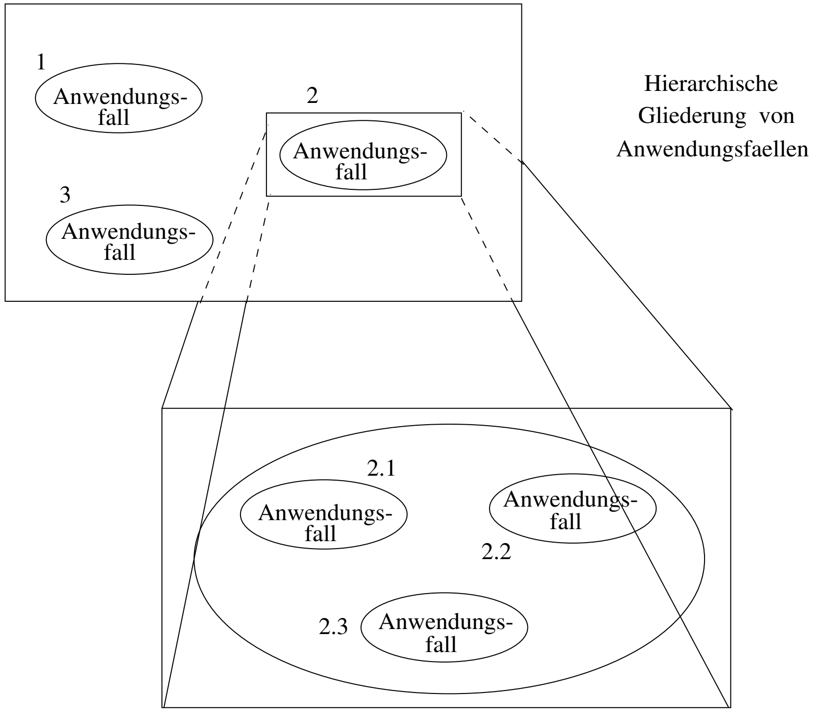
Use Cases



- the system is represented by a rectangle.
- actors are outside of the system
  - do not represent objects, but *roles*;
  - use the system, or
  - are used by the system (then they are sometimes also represented by a rectangle).
  - not necessarily persons, can also be other things (coffee automata);

Use Cases

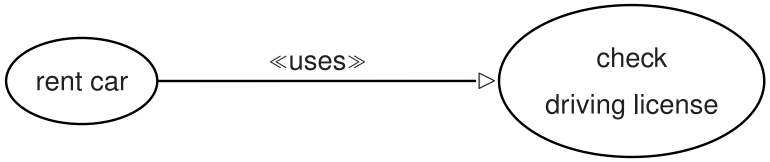
... can be structured hierarchically



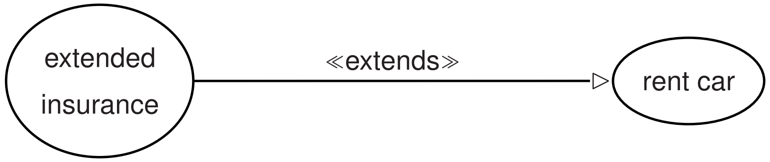
Use Cases

Possible relationships between use cases:

- A ≪includes/uses≫ B: a sub-use-case B is part of several use cases (amongst them A):

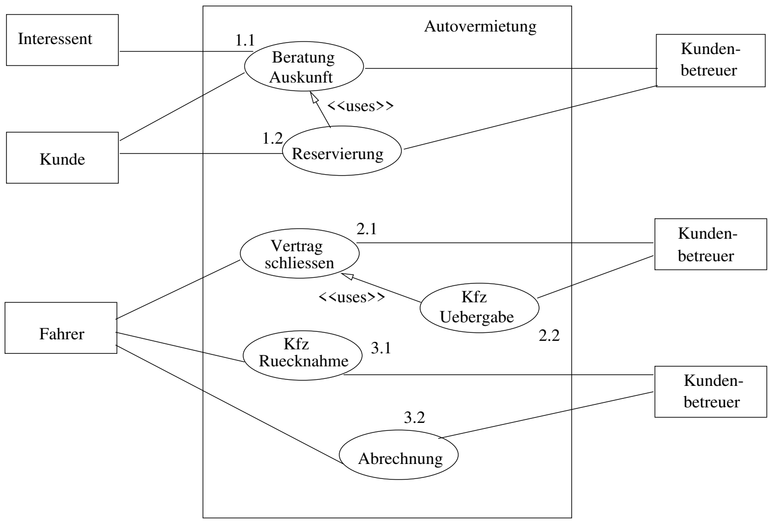


- A ≪extends≫ B:
  behavior of A can be inserted into behavior of B.



- extends: in a more detailed description, *extension points* are given where A is inserted.
- distinction often unclear [Kappel]

Use Cases



Modeling of the Behavior

The behavior of an (information-)system consists of cooperation between different objects that change their states over the time.

- specification of the coarse flow of activities (→ *activity diagrams*),
- specification of the state changes of an object as a reaction on events (messages) (→ *state diagrams*)
- specification of the cooperation between objects for performing an activity,
  - temporal sequence (→ *sequence diagrams*),
  - relationships between objects (→ *collaboration diagrams*)

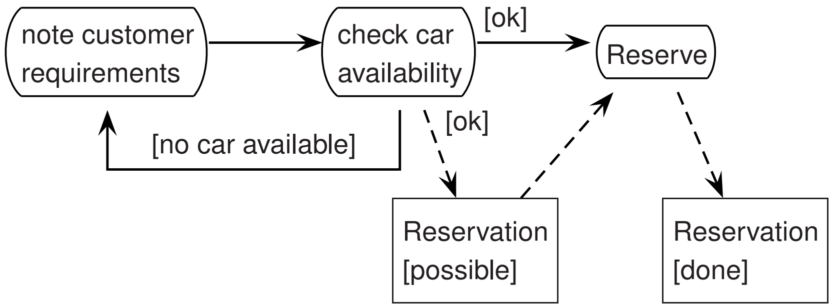
Activity Diagrams

- An *activity* represents a (part of) workflow ("of a use case"). It consists of internal actions and state transitions.
- *activity diagrams* are similar to procedural flowcharts, but all activities are unambiguously associated to objects.
- developed from data flow diagrams (OMT).
- activities and activity diagrams are associated with a class or an operation, or they are subordinate to one or more use cases. They describe the internal possibilities how a process can proceed.
- diagram syntax is derived from the syntax of state diagrams (see later).
- in comparison with state diagrams they are on a higher abstraction level: classes and objects are not yet known.

Activity Diagrams

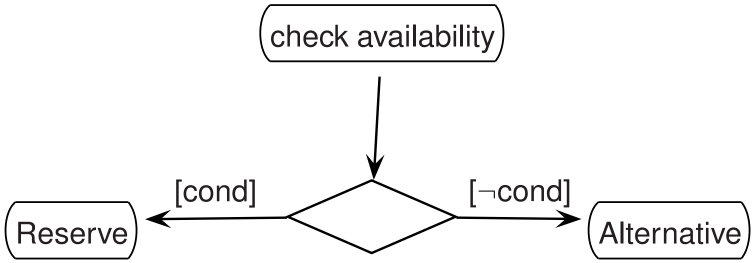
Elements:

- flow of control:
  - activities: Reserve
  - connected by transitions (actions); possibly with boolean constraints.
- object flow:
  - states of objects: Reservation [done]

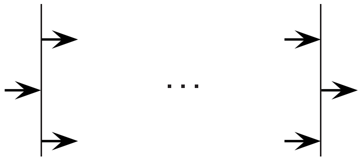


Activity Diagrams

- Decision nodes:



- Synchronisation bar
- Splitting (fork)
- Join



Example: Producer-Consumer-System [Kappel, Abb. 2-90]

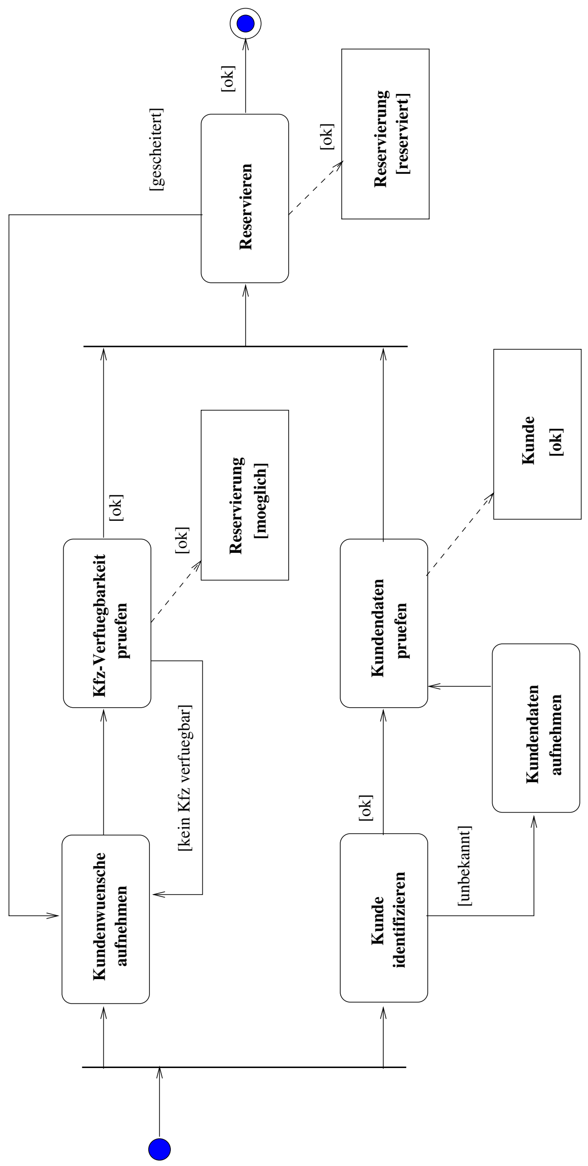
Activity Diagrams

Example: Car rental

- Customers are added to the customer file, and their data is validated,
- The needs of the customers are noted and – as far as possible – satisfied.
- If the customer is accepted and the required car is available, it is reserved.

```
develop
example diagramms
at the blackboard
```

Activity Diagrams



State Diagrams

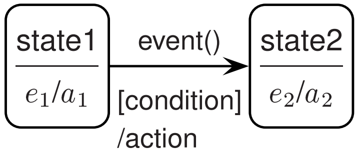
- describes the states that an object can have during its lifetime,
- and the state transitions that are caused by events/messages (intra-object behavior).
- base: finite automata
- more detailed than activity diagrams, but similar symbols

Elements:

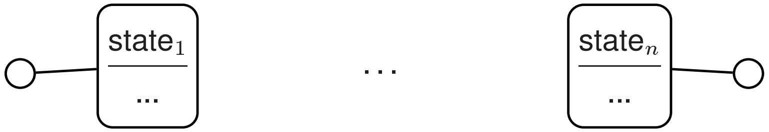
- states

- state transitions that are caused by events,

  possibly with a condition



- start- and end node:



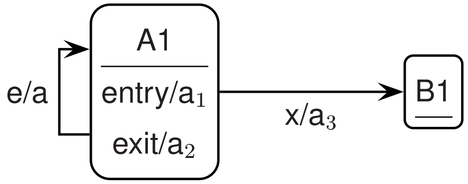
State Diagrams

- Internal transitions:
  - event/action-pairs inside of the states
  - Pseudo events entry/ and exit/, that are executed when entering/leaving a state.

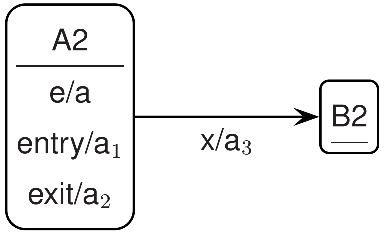
Example

The both simple state diagrams below describe different behavior:

1.



2.

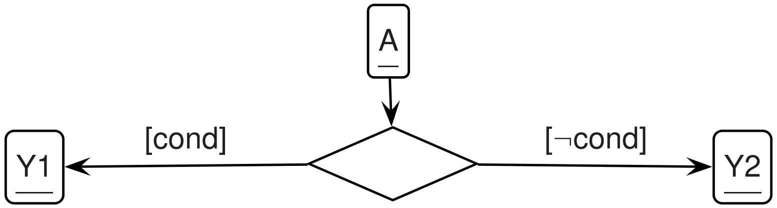


Consider the event sequence “eeeex”:

1. a_1 a_2 a a_1 a_2 a a_1 a_2 a a_1 a_2 a a_1 a_2 a_3
2. a_1 aaaa a_2 a_3

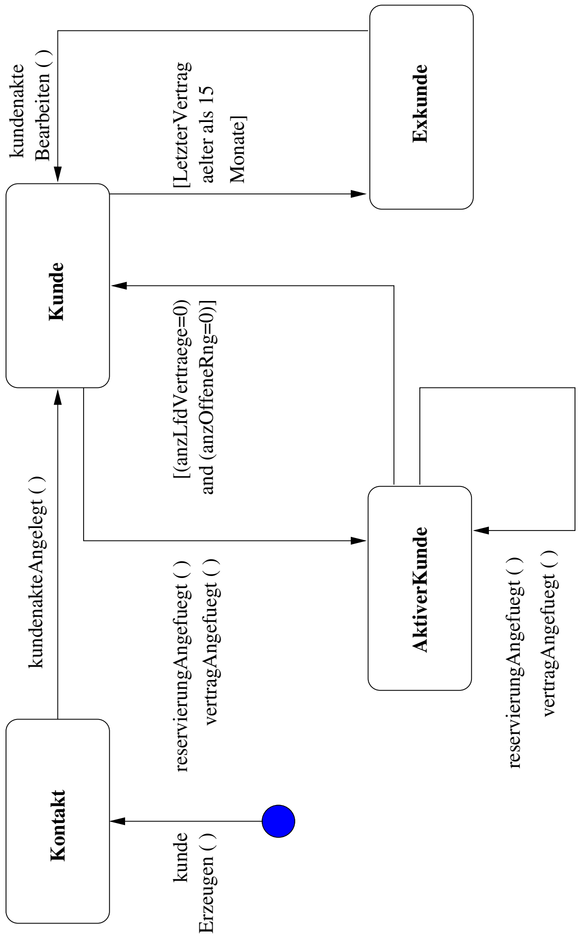
State Diagrams

- Decision nodes:



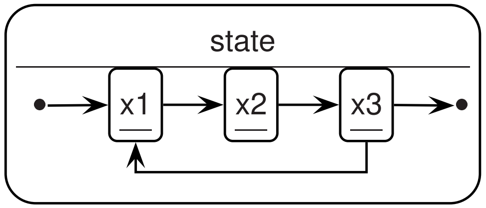
- event-types:
  1. SignalEvent: LeftButtonDown
  2. OperationEvent: object.operation
  3. ChangeEvent: check condition: when n$>$100
  4. TimeEvent: after 5 seconds since leaving state x
- note: ChangeEvent (ongoing test) not the same as Event+Condition (ECA: Event-Condition-Action)

State Diagrams

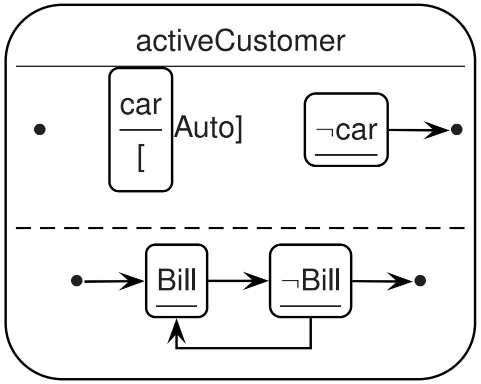


State Diagrams

- nested state diagrams
- OR-Refinement



- AND-Refinement
  - true concurrency
  - orthogonal descriptions:



  - can be represented by a product state [Blackboard].
- Complex transitions: Synchronisation with fork and join.
- Combination of state diagrams (Sync-states) [Kappel, p.148]

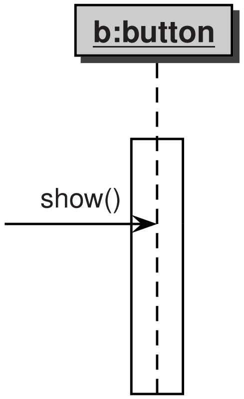
Sequence Diagrams

Sequence diagrams describe the temporal aspect of the communication of objects.

- originally developed for modeling of telecommunication systems (Message Sequence Charts, CCITT 98, SDL).
- exemplary description of a workflow.

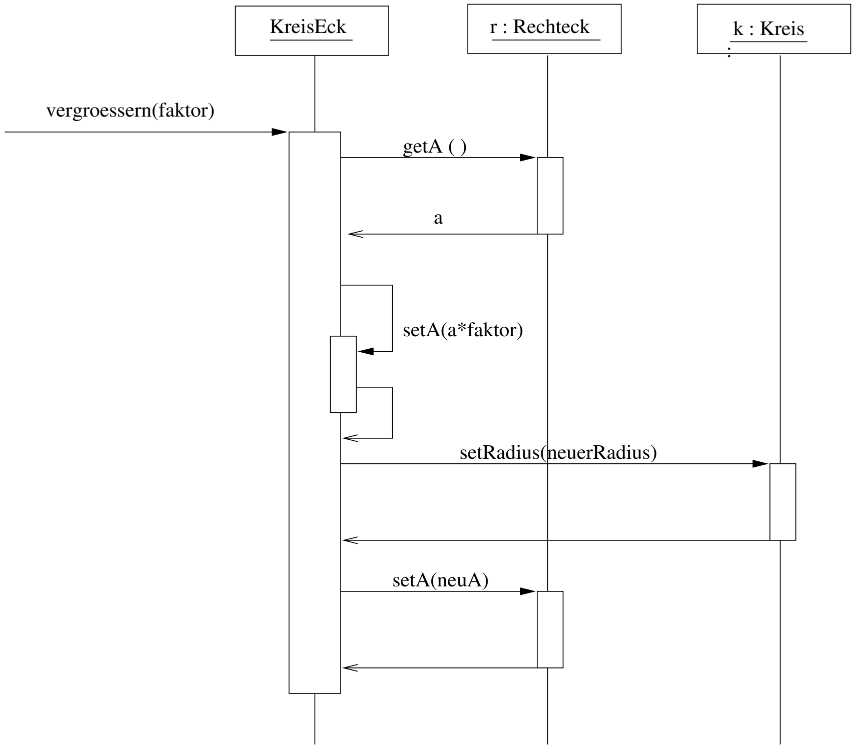
elements:

- vertical time axis
- horizontally: all objects that contribute to a workflow
- information which objects are active at a given timepoint,
- submitted messages, possibly with constraints
- Example: phone call in [Kappel, Ex 2-63], extend by creation of a Kermit-connection.



Sequence Diagrams

Example: enlargement of a *button*-object



Sequence Diagrams

The following things can also be described

Example graphics see Blackboard/[Kappel, p110ff]

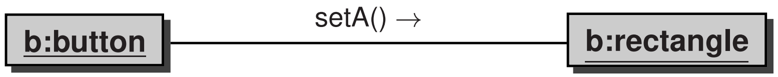
- objects, that have an own control flow (Process/Thread): thick lines, {active}, continuous activity bar,
- creation of objects,
- deletion of objects,
- synchronous (—▶), asynchronous (⇀), unspecified (→) communication,
- return: (- - - - ➤),
- time-consuming messages,
- constraints for messages: [*constraint*]
- branching (if-then-else-messages) (with annotated condition)
- alternative lifelines of objects (dependent of conditional messages)
- iterations
- comments at the left margin, or at the objects/messages

Collaboration Diagrams

- based on an excerpt of the class diagrams, that contains those model elements (classes, objects, relationships) that contribute to a certain outcome.

elements:

- all participating objects,
- passed messages



- possibly with answers:

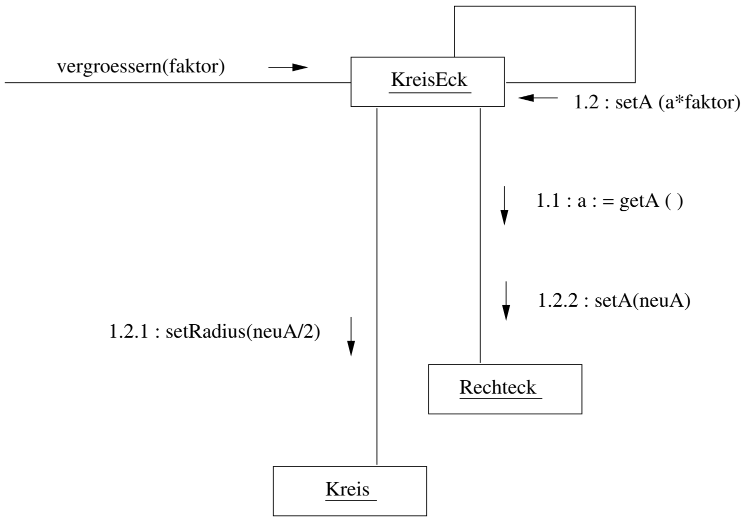
  a := getA() →

- temporal sequence denoted by *sequence numbers* at the messages

  1.1 a := getA() →   1.2 setA(a * factor) →

- hierarchical structuring (including. concurrency) of actions by sequence numbers: 1, 1.1, 1.2, 1.2a, 1.2b, 1.2a.1, 1.2a.2,
- additionally, messages to *self* are noted.

Collaboration Diagrams

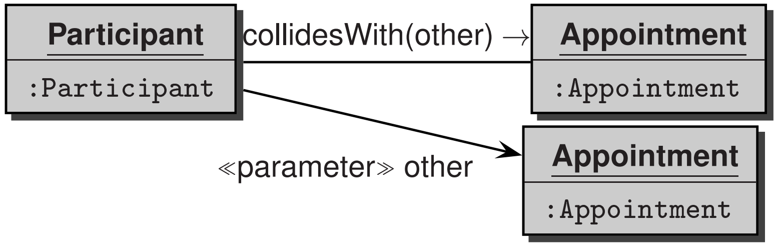


[Blackboard: overlay an excerpt of the class diagram (Slide 82)]

Collaboration Diagrams

- Multiobjects: if a message is sent in parallel to several objects (double box):

- allowed to use objects as parameters/arguments: arrow with annotation:



- constraints: annotation of the arrow: 2.2 $[x > 0]$: divideBy(x)
- iteration: 2.2 $*$ [i=1..n]
- concurrent execution: 2.2 $\|*$ [i=1..n]

Exercise: UML vs. ER-model

Exercise 9 (ER-modeling/UML-modeling)
Compare the ER-modeling with UML-modeling. What are the differences, what are the (different) goals of both approaches.

Exercise 10 (OOM – Soccer-Championship (Dynamics))

- Describe the actions, that must be executed when `WC.create(2002,Parms)` is called, i.e. when the system initializes a new championship object with given parameters.
- specify further actions that must be completed for initializing the system (registration of teams, players, groups).
- Give sequence diagrams and collaboration diagrams for characteristic events of a match (begin, substitution of a player, yellow card, end).

Lecture Information Systems

Exercise: UML vs. ER-model: Solution

ER-modeling describes a part of the static aspects of an application domain: the focus is on entity types and relationship types, each of them with attributes. More special concepts (aggregation, composition, has-relationships etc.) are translated into relationship types with suitable cardinalities and roles. There are mappings how to transform an ER-model into a relational schema.

UML-modeling describes the static *and* dynamic aspects (behavior, communication) of an application domain. Due to the object-orientation, special concepts for representing different relationships (subclasses, general associations, aggregation, composition) are provided.

The *behavior* of an application can also be described in UML (activity diagrams, sequence diagrams, state diagrams). Corresponding formalisms do not exist in the ER-model.

Additionally, high-level formalisms are provided for describing workflows and business processes (roles, use cases, CRC-cards), that can be used for conceptual design of workflows.

There are tools (e.g. ObjectiF, ROSE) that create a Java/C++/Visual Basic skeleton from an UML-specification.

2003: UML 2.0 compatible with the new OMG standard "Model-Driven Architectures (MDA)"

Solution: Soccer-Championship

Generation of required objects:

- createWC(WC2002,Parameters):
  creates an empty championship-object. Parameters describes the pattern of the schedule, i.e. how many groups there are, how many participants they have, how many final rounds take place, and how the participants of the finals are determined. The group objects are generated, also the match objects, but all of them are empty. The teams are abstractly denoted by A1,...,A4,B1,...,H4. The teams that participate in the finals are not yet known.
- WC2002.createTeam(Team,GrNr,Parameter), ...:
  creates team-objects and assigns them to groups.
- WC2002.createPlayer(Player,Team,Parameter), ...:
  creates player objects, and assigns them to teams.

State: The score of all matches is 0:0, tables are built only with 0:0 results, the opponents in the group matches are known, those of the finals are not yet known.

Modeling of Events

Assignment of events to objects?

In general, there are two possibilities:

- All events are assigned to the root object WC2002, e.g. WC2002.▷Goal(group match,D,USA,85,name) or WC2002.▷YellowCard(semifinal,BR,AUS,Ronaldo). This case corresponds to a completely encapsulated IS where the user is not involved with internal objects.
- events are identified with the object where they occur, e.g. in the object WC2002.Team("D").GroupMatch_vs("USA") an even ▷Goal(85,name) occurs, or in the object WC2002.Semifinal("BR","AUS") an even ▷YellowCard(85,name) occurs. This corresponds to a distributed IS, where events occur at different places at arbitrary timepoints.

Modeling of Events

Begin of a Match:

- Match.▷Begin(FormationTeamA,FormationTeamB) changes the match status to ongoing, CurrScore to 0:0 and aktualizes the current group ranking. Additionally, Team.▷Formation is invoked with the current formation of the teams. This in turn sends a message ▷Match to the objects that represent the nominated players of the teams, and send a message ▷pause for each player of these teams who does not participate. This event is used for maintaining bans due to yellow cards: in case that Player.BannedYellow = "next", its is changed to Player.BannedYellow = "this".

Yellow Card:

- Match.▷YellowCard(Player,Comment) is added to the match report. Additionally, the method Player.YellowCard of the corresponding player object is invoked. In case that it is the second one, Player.BannedYellow := "next" is set.

Modeling of Events

Goal in a GroupMatch:

- Match.▷Goal(Minute,Goalgetter,Comment) updates the CurrScore of the Match object and calls the method ▷Evaluate of the corresponding group, which then updates the group ranking. For this, all matches are asked abou their CurrScore. Then, the participants (Group leader and second) for the quarterfinals are re-determined. The change of the CurrScores is also forwarded to the participating team objects to update the statistics. Additionally, the goal message is sent to the player-object of the goalgetter.

End of a GroupMatch:

- Match.▷EndMatch sets the match status to finished. EndMatch is also sent to the team objects, that invoke Player.▷EndMatch for their players, here again bans due to yellow cards are processed: in case of Player.BannedYellow = "this", Player.BannedYellow = "no" is set.

Modeling of Events

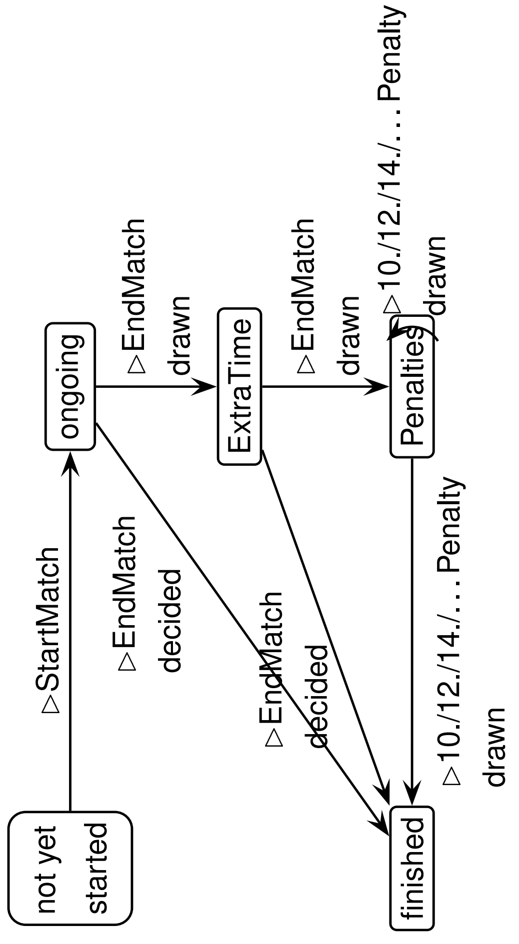
Goal in a final match:

- Match.▷Goal(Minute,Goalgetter,Comment) updates the CurrScore of the match object, and updates the participants of the subsequent final game. Other actions as described above.

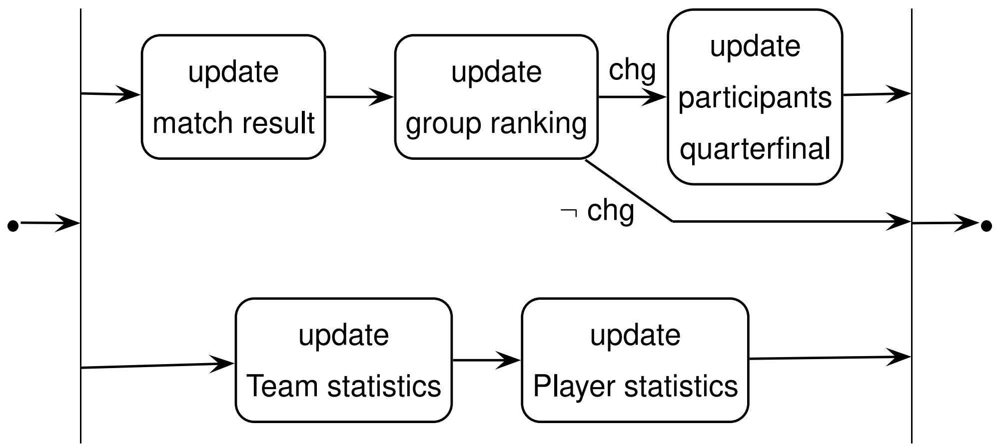
EndMatch of a final match:

- Match.▷EndMatch in a final match: sets the match status to finished if there is a winner. Otherwise, the status is changed to ExtraTime or penalty shootout. Other actions as described above.

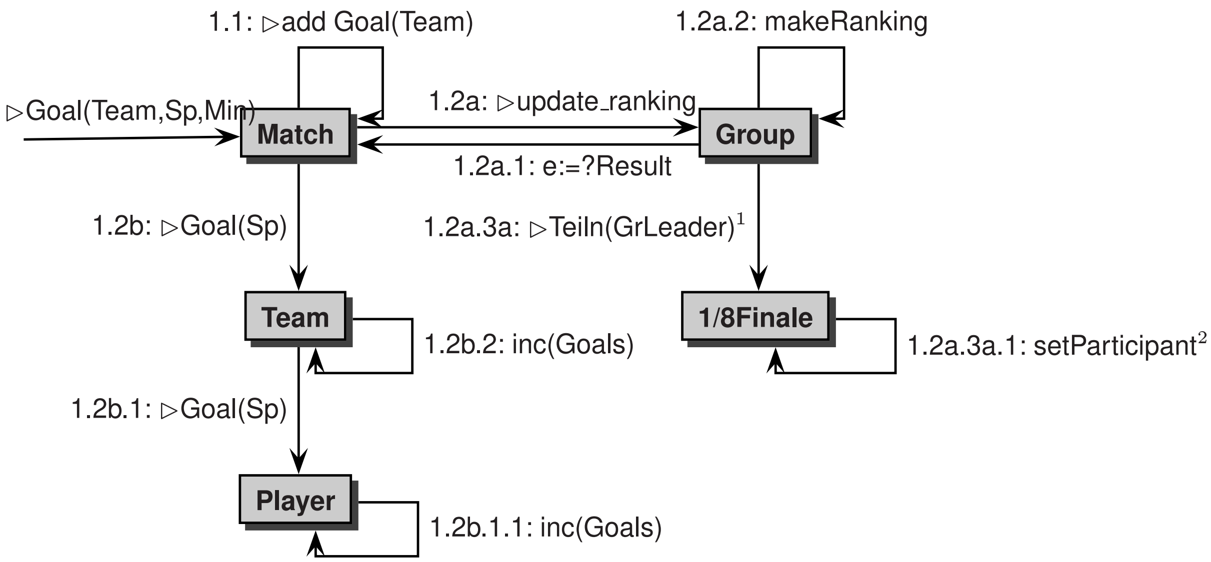
State Transition Diagram: Match in the Final Phase



Activity Diagram: Goal in a GroupMatch

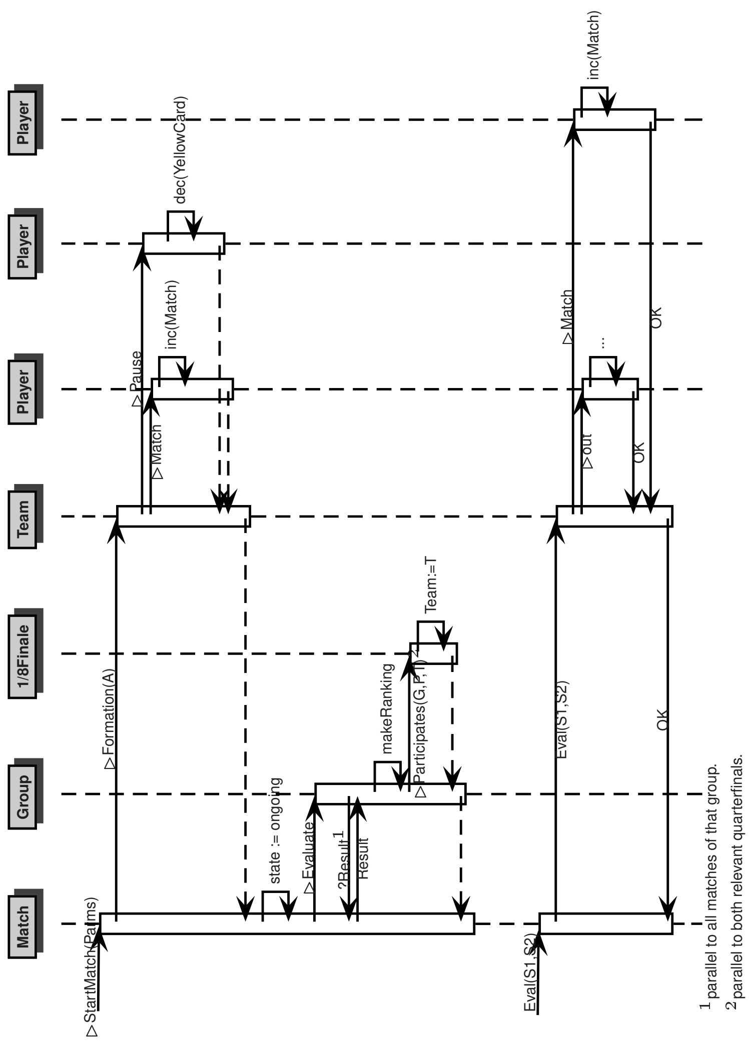


Collaboration Diagram: Goal in a GroupMatch



[1] parallel: 1.2a.3b: ▷Participates(T) to the other relevant quarterfinal with the second-ranked team of the group.
[2] parallel: 1.2a.3b.1: setParticipant to the other relevant quarterfinal.

Sequence Diagram: GroupMatch



1 parallel to all matches of that group.
2 parallel to both relevant quarterfinals.

Languages: Requirements

(not only for OODB languages!)

Closedness: the result of a query must be representable with the concepts of the data model.

Completeness, Adequacy: the language must provide generic operations corresponding to the concepts of the data models (constructors, component selection, dereferencing, handling of sets/lists ...).
All concepts must be usable in the result.

Descriptive Language

Formal Semantics

Optimization (e.g. algebraic semantics)

Orthogonality of operations (i.e. arbitrary nesting/combinations allowed)

Exercise: check these requirements for the relational model/SQL.

Queries: Result types and -Classes

- answer to a query consists of an object or a set of objects (often implicit).
- insert results of a query into the database: results must be of a type that is defined in the DB, and they must be added to a class.
- result type:
  - result type is a supertype (less attributes) of the original type (projection)
  - result type = original type (selection, set op's)
  - result is a subtype of the original type (join)
- result class:
  - projection: class does not change
  - selection, difference: result is a subclass of the original class (person, employee)
  - intersection: subclass of both original classes (student $\cap$ employee)
  - union: superclass of both contributing classes
  - join: extends the set of attributes of a class by attributes of another class. Class membership does not change.
    Joins are here non-symmetric!

Derived Classes and Views

- define super- and subclasses as *derived classes* (Teenager = σ[14<age<20](persons))
- e.g. restrict attributes by projection: (teenager = σ[14<age<20](π[name,age](persons)))
- Views: $\Rightarrow$ sets of virtual instances with less attributes in a virtual (intensionally defined) class, that are derived on-demand.

Generic Update Operations

Updates on classes:

- create: `NEW` or `CREATE`; type constructors
- delete: `DELETE`, `FORGET`; type destructors
- insert: `INSERT`, `ADD` adds an existing object to another class (if multiple class membership is allowed).
- remove: `REMOVE` removes an object from a class (but does not delete it)

Updates on generically structured things:

- tupels: componentwise access
- sets, lists: the usual operators ... insert, delete, union, intersection, difference.

OQL

- Part of the ODMG-Standard
- Underlying data format: OIF (cf. Slide 121)
- Data Definition Language: ODL
- Query language: OQL (Object Query Language)
- similar to SQL:
  SELECT - FROM - WHERE - clause, extended by complex objects, object identity, path expressions, invocation of operations, polymorphism and late binding.

- but: functional language (holds also for SQL), fully orthogonal (does not hold for SQL)
- no explicit UPDATE-statement: use object methods.
- not computationally complete (cf. SQL/transitive closure)
- OQL can be embedded into suitable object-oriented programming languages (C++, Java, Smalltalk).

Extents

SQL: SELECT ... FROM <relation> ...

What corresponds to a *relation* in the ODBMS ?

⇒ *Extension*: set of all instances of a class.

Extensions are defined in ODL with the class declaration:

```
class <name>
(extent <extent_name>
 keys <key list>)
{  <attribute-def>;
   <relationship-def>;
   <operations-def>; }
```

```
class Country
(extent Countries
 keys Code)
{  attribute string name;
   relationship City capital;
   set<string>languages;
   ...}
```


Queries

Queries against the database are expressed by the SELECT command. It has a simple basic structure:

```
SELECT   <expression>
FROM     <extents>
WHERE    ...
```

```
SELECT c.Population
FROM Countries c
WHERE c.Name = 'Germany'
```

Queries

Similar to SQL:

- DISTINCT
- aggregation functions: COUNT, SUM, ...
- set operations: UNION, INTERSECT, EXCEPT (MINUS)

Extended: EXISTS
Additionally: FOR ALL

{FOR ALL | EXISTS} x IN (subquery) : (condition)

Conditional evaluation: X ANDTHEN Y, X ORELSE Y

Aside: similar concepts will be found in the XQuery language for XML.

Example: Relational Division

Compute all political organizations that have at least one member country on each continent.

SQL:

```
SELECT abbreviation from organization o
WHERE NOT exists (
(SELECT continent FROM encompasses)
MINUS
(SELECT continent FROM is_member, encompasses
 WHERE organization = abbreviation
 AND is_member.country = encompasses.country));
```

OQL:

```
SELECT org.abbreviation from organization org
WHERE FORALL cont IN (SELECT c
                      FROM continent AS c):
   (EXISTS country IN (SELECT m
                       FROM org.members AS m):
        (cont IN (SELECT cont2
                  FROM country.encompassed AS cont2)))
```

Queries

- All SQL queries

  SELECT a,b,c FROM ...

  return virtual "relations" (i.e. sets of tupels) as a result,
- in OQL, the result is a virtual set of objects,
- mostly of the datatype

  collection <...>

  or, more detailed (see later)

  bag <... >
- the bag constructor is not explicitly mentioned in the query.

```
SELECT c.Capital
FROM Countries c
```

Result is of the type

collection <city>

⇒ queries can be arbitrarily nested

Lecture Information Systems

Queries

in case that the result has multiple "columns" (e.g., for SELECT *),

automatically, a

bag <struct{...}>

is generated:

```
SELECT c.Name, c.Population
FROM Countries c
WHERE c.Name = 'Germany'
```

Result is of the type

collection <struct {string Name; number Population}>

method invocation can be used instead of attributes; again, late binding is applied:

```
SELECT c.max_city()
FROM Countries c
```

Complex Results

- bags (here: set-valued relationship) can be handled as a whole,
- by explicitly generating a struct, properties of the results can be (re)named:

```
SELECT struct(name: c.Name,
              cities: c.has_cities)
FROM Countries c
```

Result is of the type

```
collection <struct {string Name;
                    collection<city> cities}>
```

How to select something from *inside the collection*, or to generate a collection?

... very simple: apply a SELECT-Statement:

```
SELECT struct(Name: c.Name,
              cities: (SELECT cty
                       FROM c.has_cities as cty
                       WHERE cty.population>1000000))
FROM Countries c
```

- Traversing of the relationship *has_cities* by a *path expression* in the query
- nested SELECT in the SELECT-Statement: the inner SELECT iterates over the (virtual) set *c.has_cities* of instances of type *set<Cities>*.
- the inner SELECT is separately evaluated for every iteration (i.e. for every instance c) of the outer SELECT.

Path Expressions

for navigation along *scalar* relationships:

```
SELECT Name: c.Name,
       cpp: c.capital.province.population
FROM Countries c
```

Undefined attributes

Definedness can be tested by
WHERE IS_DEFINED(<attr>) or
WHERE IS_UNDEFINED(<attr>).

Each other use of an *undefined*-value results in an error message.

```
SELECT Name: c.Name,
       cpp: c.capital.province.population
FROM Countries c
WHERE IS_DEFINED (c.capital.province)
  ANDTHEN IS_DEFINED (c.capital.province.population)
```


Lecture Information Systems

Path Expressions

Navigation along set-valued methods

`not allowed:`

```
SELECT name: c.has_cities.Name,
       pop: c.has_cities.population
FROM Countries c
WHERE c.Name = “Germany”
```

has_cities is a *set of references*, thus, the method *population* cannot be applied to it.

SELECT in the FROM-Line

can e.g. be expressed by SELECT-Statements in the FROM-line:

```
SELECT name: cty.Name,
       pop: cty.population
FROM (SELECT c.has_cities
      FROM Countries c
      WHERE c.Name = “Germany”) as cty
```

Correlated Joins

... even more elegant for the above example:

```
SELECT name: cty.Name,
       pop: cty.population
FROM Countries c, c.has_cities cty
WHERE c.Name = "Germany"
```


Lecture Information Systems

Named Objects

- the ODMG-model supports objects that have an externally known object identifier
- OIF:

  ```
  Berlin City
     {Name "Berlin", in_country Germany,
      Coordinates {Longitude 13.3, Latitude 52.45} }
  ```

  defines the object *Berlin*.
- Object identifiers can be used in OQL:

  ```
  Berlin
  Berlin.in_country
  Berlin.Coordinates.Longitude
  ```
- names of extents allow for accessing the set of instances of a class:

  ```
  Persons
  ```

OQL: Functional Language Design

SQL:

- declarative
- ad-hoc language (around SELECT – FROM – WHERE)
- no orthogonal composition rules

OQL:

- orthogonal composition rules: operators can be arbitrarily nested as long as the type system is not violated.
- functional concept that subsumes the SQL syntax for *simple* queries.
- Result of a query is always a collection
  - of structs (by default), or
  - of objects;
- thus, it can be used in the same way as an extension (*intensional* part of the database).

OQL: Formal Language Definition

Recall the *"Introduction to Databases and SQL":*

- semantics of SQL equivalent to relational algebra
- shown in DB I: equivalence of Algebra and basic SFW
- not for grouping (GROUP BY, HAVING), subqueries in WHERE, aggregate functions etc.

Now: OQL as an example of a "clean" definition.

- OQL-*expressions* are recursively defined by *operators* based on typed *operands*.

Type Compatibility

Two types t and t' are *compatible*, if instances of these types can be inserted together in a collection (of a supertype):

1. t is compatible with t.
2. if t and t' are compatible, then also
   - set(t) and set(t'),
   - bag(t) and bag(t'),
   - list(t) and list(t'),
   - array(t) and array(t'),

   are compatible.
3. struct(p_0:t_0,...,p_n:t_n) and struct(p_0':t_0',...,p_m':t_m') are compatible, if $m = n$ and t_i and t_i' are compatible for all i.
4. object types t_1 and t_2 are compatible, if there is a t, such that t is a supertype of t_1 and t_2.

Lecture Information Systems

Type Compatibility

(4) especially implies

- literal types are incompatible with object types,
- atomic literals are only compatible, if they belong to the same (atomic literal)type,
- Structured literal types are compatible, if they are derived from the same interface (Date, Interval, Time, Timestamp are interfaces),
- Atomic object types are compatible, if they have a common supertype,
- collection types (both collections of literals as well as collections of objects) are compatible, if they are of the same collection interface, and their types of collection elements are compatible.

Minimal Supertype:

If t_1, t_2, ..., t_n are compatible are, then there is a unique t, such that

- t $>$ t_i, and
- for all t' such that t' $>$ t_i and t'$\neq$ t, t' $>$ t
- this t is called *lub(* t_1, t_2, ..., t_n*)*.

OQL: Definition of Semantics

By structural induction:

Terms:

- constants:
  - literals
  - named objects: individual objects
  - named extents: collections of objects
- composed terms:
  - path expressions: attributes, relationships & functions
  - operators (on built-in domains)

Expressions:

- Constructors (in the SELECT line)
  - object constructors
  - struct constructors
  - collection constructors
- auxiliary:
  - operations on collections
  - aggregate operations [over collections]
  - quantification over collections
- SELECT-FROM-WHERE

Expressions in OQL

Atomic Literals

For an atomic literal l, l is an expression that evaluates to l itself:

- nil is an object literal,
- true, false are boolean literals,
- 42, 3.14, 'z', and 'bla' are Integer-, Float-, Character-, and String-literals, respectively.

Named Objects

For an object name <name> <name> is an ein expression, whose value is the corresponding object, e.g. *John*. The type of <name> is the type of that object.

Extents, Sets of Objects

For a name <name> of an extent of a type t, the value of <name> is the set of corresponding instances, e.g. *Cities*, the type of <name> is collection(t).

Expressions in OQL: Path Expressions

For a type t_0 that provides a method (property or functional operation) f with arguments of types $t_1, \ldots, t_n$ and result type t, and for expressions $<$expr$_0>$, $<$expr$_1>$, ..., $<$expr$_n>$ each of type t_i,

$<$expr$_0>$.f($<$expr$_1>$, ..., $<$expr$_n>$)

is an expression of type t.

The value of the expression is the value that results from applying the method to the host object; or *nil* if the method returns nothing. In case that $<$expr$_0>$ is not an existing object, the result is *undefined*.

Germany.president(2002)

results in the object *Johannes_Rau*, whose age can be selected by

Germany.president(2002).age(2002)

Lecture Information Systems

Expressions in OQL: Operators

For an expression $<$expr$>$ of type t, and a unary operator $<$op$>$ (e.g. abs, sin, not, len) over t,

$<$op$>$ $<$expr$>$

is an expression of the corresponding result type:

not true, abs(-5), len('bla')

Analogously for binary operators.

Germany.population / Germany.area

Operators:

+, -, *, /, =, !=, $<$, $<=$, $>$, $>=$,
string || string, string + string (both denotes concatenation),
string[from:to], string like pattern

(Un)Equality:

- allowed between compatible types, evaluated over lub(t_1,t_2),
- also defined over sets, bags, lists and arrays.

Expressions in OQL: Constructors

For a type t and properties $p_1, p_2, \ldots, p_n$ of t, each of them of type t_i, and expressions $<\text{expr}_i>$ of type t_i,

$t(p_1{:}<\text{expr}_1>, \ldots, p_n{:}<\text{expr}_n>)$

is an expression of type t.

The value of the expression is an object of type t, whose properties are initialized with the given values of $<\text{expr}_i>$ (use of t as *constructor*):

City(name: “Berlin”, in_country: Germany, . . .)

Expressions in OQL: Structs

For names of properties $p_1, p_2, \ldots, p_n$, and expressions $<\text{expr}_1>, <\text{expr}_2>, \ldots, <\text{expr}_n>$ of types t_i,

$\text{struct}(p_1{:}<\text{expr}_1>, \ldots, p_n{:}<\text{expr}_n>)$

is an expression of type

$\text{struct}(p_1{:}\ t_1, \ldots, p_n{:}\ t_n)$.

The value of the expression is the corresponding *struct*.

struct(name: “Berlin”, in_country: Germany, . . .)

Note.: this *struct* is completely different from the *object* on the preceding slide that had been defined in a similar way using an object constructor.

```
SELECT struct(name: cty.name,
              country: cty.country.name)
FROM Cities cty
```

Expressions in OQL: Collections

For expressions <expr$_1$>, <expr$_2$>, $\ldots$, <expr$_n$> of compatible types t$_1$, t$_2$, $\ldots$, t$_n$,

set(<expr$_1$>, <expr$_2$>, $\ldots$, <expr$_n$>)

is an expression of type set(t), where t = lub(t$_1$, t$_2$, $\ldots$, t$_n$).

The value is the set, whose elements are the values of <expr$_i$>.

Analogously for *list*, *bag* and *array*.

In most cases, collections are created implicitly:

```
SELECT struct(name: c.name,
              cities: SELECT cty
                      FROM c.has_cities AS cty)
FROM Countries AS c
```

creates a collection of structs.

Lecture Information Systems

Expressions in OQL: Membership

For an expression <expr$_1$> of type t and an expression <expr$_2$> of type set(t') for any supertype t' of t,

<expr$_1$> IN <expr$_2$>

is an expression of type *boolean*.

The value of the expression is *true* if the literal or object denoted by <expr$_1$> is contained in the collection denoted by <expr$_2$>.

```
Berlin IN (SELECT c.capital FROM Countries c)
```

Expressions in OQL: Set Functions

For expressions <expr$_1$> and <expr$_2$> of type *set* or *bag* over compatible types t_1 and t_2,

<expr$_1$> <op> <expr$_2$>

for <op> $\in$ {UNION, INTERSECT, EXCEPT} is an expression of type

- set(t), if both are of type set(t),
- bag(t), if at least one of the expressions io of type bag(t).

The value of the expression is the collection that results from the given set operation.

```
Cities EXCEPT (SELECT c.capital
               FROM Countries c)
```

Furthermore, compatible collections can be compared by $<$, $<=$, $>$, $>=$.

Lecture Information Systems

Expressions in OQL: Aggregation Functions

For an expression <expr> of type *collection(t)*,

<op>(<expr>)

for <op> $\in$ {MIN, MAX, SUM, AVG, COUNT} is an expression of type

- t, if <op> $\neq$ COUNT,
- number, if <op> = COUNT.

```
COUNT(Persons) - COUNT(Students)
```

Note that aggregate functions are used in OQL with the following syntax:

```
MAX(SELECT c.population FROM Countries c)
```

that is equivalent with the "classical" SQL syntax (which is also supported in OQL)

```
SELECT MAX(c.population) FROM Countries c
```

Expressions in OQL: Quantification

For a variable x and an expression <expr_1> of type *collection*, and an expression <expr_2> of type *boolean*,

for all x in <expr_1> : <expr_2(x)>

is an expression of type *boolean*.

The value of the expression is *true*, if all elements of the collection <expr_1> satisfy the condition <expr_2>, otherwise *false*.

for all x in Cities : ABS(x.latitude) $<$ 90

Analogously for

exists x in <expr_1> : <expr_2(x)>

Note that the SQL syntax

SELECT ... FROM ...
WHERE EXISTS (SELECT – FROM – WHERE)

is also allowed.

... and now to the core: SELECT-FROM-WHERE

Expressions in OQL: Iterator Variables

For a variable x that is defined in the FROM-clause of a SELECT – FROM – WHERE-Statements, for a collection of type t, i.e.,

- ... <expr> AS x
- ... <expr> x
- ... x in <expr>

where <expr> is type collection(t),

- x is of type t,
- the value of x is the current value of iterating over this collection.

```
SELECT cty
FROM Cities as cty
WHERE cty.country.name = 'Germany'
```

Expressions in OQL: SFW-Expressions

The general form of a SELECT-Statement in OQL is

SELECT [DISTINCT] $<\text{expr}_0(x_1, \ldots, x_p)>$
FROM x_1 in $<\text{expr}_1(x_{n+1}, \ldots, x_{n+p})>$
    x_2 in $<\text{expr}_2(x_1, x_{n+1}, \ldots, x_{n+p})>$
    $\vdots$
    x_n in $<\text{expr}_n(x_1, \ldots, x_{n-1}, x_{n+1}, \ldots, x_{n+p})>$
[WHERE $<\text{pred}(x_1, \ldots, x_p)>$]
[ORDER BY ...]

(equivalent: $<\text{expr}_i(...)>$ AS x_i)

where $<\text{expr}_i>$ are of type *collection* (e.g. extents or SFW-statements), and and $<\text{pred}>$ is of type *Boolean*.

The result is then of type collection(t), where t is the result type of $<\text{expr}_0(...)>$.

Evaluation of a SFW-Statement

The result of the FROM clause is of type

bag $<\text{struct}(x_1\colon t_1, x_2\colon t_2, \ldots, x_n\colon t_n)>$

where the t_i are the types of $<\text{expr}_i(...)>$ and holds the cartesian product of the collections

$<\text{expr}_1(\ldots)> \times \ldots \times <\text{expr}_n(\ldots)>$.

From this *bag*, only those Tupels

struct($y_1, y_2, \ldots, y_n$) are selected that satisfy

the predicate $<\text{pred}(x_1, \ldots, x_p)>$.

Then, for each of these tuples, the expression $<\text{expr}_0(...)>$ is evaluated and the resulting *collection* is returned: the result is of type

- *list*, if ORDER BY is specified (semantics as known from SQL),
- *set*, if DISTINCT (but not ORDER BY) is specified,
- *bag* otherwise.

Expressions in OQL: GROUP BY, HAVING

... different from in SQL (but includes the SQL case):

Review: SQL

`GROUP BY` computes for every group *one row* that contains data that is computed by applying aggregate functions on all rows of that group.

SELECT <expr-list>
FROM <table>
WHERE <predicate>
GROUP BY <attr-list>;

where `<expr-list>` contains only

- constants,
- attributes from `<attr-list>`,
- attributes, that have the same value for all tuples in each group (e.g. `Code`, if `<attr-list>` is *Country*),
- *aggregation functions* which are then computed from all tupels of each group.

Example (SQL):

SELECT Country, count(*)
FROM City
GROUP BY Country

Expressions in OQL: GROUP BY

OQL provides a more flexible functionality:

SELECT ...
FROM ...
WHERE ...
GROUP BY <partition attributes>
HAVING ...

1. the grouping attributes can be computed arbitrarily from the input
2. each row of the result contains a collection of objects that belong to that group.

Example:

The “classical” GROUP BY from SQL where the grouping is done by attribute values is obtained by choosing <partition attributes> as the identity mapping of one or several attributes:

SELECT *
FROM Cities c
GROUP BY country: c.country

result type:

set <struct(country: Country, partition: bag<struct(c: City)>)>

Expressions in OQL: GROUP BY

SELECT *
FROM Cities c
GROUP BY small: population $<$ 100000,
medium: 100000 $<=$ population
AND population $<$ 1000000
big: 1000000 $<=$ population

result type:

set <struct(small: boolean, medium: boolean, big: boolean,
partition: bag<struct(c: City)>)>

small	medium	big	partition
true	false	false	$\{o_{hintertupfing}$ $o_{kleinkleckersdf}$ $\vdots$ $\}$
false	true	false	$\{o_{freiburg}$ o_{basel} $\vdots$ $\}$
false	false	true	$\{o_{berlin}$ o_{paris} $\vdots$ $\}$

Expressions in OQL: GROUP BY

What happens for

GROUP BY country:country ?

- if defines a new, virtual attribute “country”, that exactly holds the value of the country attribute of the input object.

Exact proceeding in OQL:

- partition attributes is of the Form

  attr_name$_1$: <expr$_1$>, ... attr_name$_n$: <expr$_n$>,
- for each element of the input collection, the new, virtual attributes are computed (view-like),
- the resulting extended objects are grouped by these attributes,
- each tuple of values of the partition attributes attr_name$_1$, ... , attr_name$_n$) defines a row of the result, where attr$_{n+1}$ is a collection that contains all elements of the group.

Expressions in OQL: GROUP BY, HAVING

SELECT *
FROM ...
WHERE ...
GROUP BY attr$_1$: <expr$_1$>,
attr$_2$: <expr$_2$>,
⋮
attr$_n$: <expr$_n$>

Assume t$_i$ to be the type of <expr$_i$>, the the result is of the type

set <struct(attr$_1$: t$_1$, attr$_2$: t$_2$, ..., attr$_n$: t$_n$,
partition: bag<t>)>

where t is a *struct* that contains one attribute for each collection mentioned in the FROM-clause:

struct(v$_1$: type_of_from$_1$,..., v$_k$: type_of_from$_k$)

- With

  HAVING <pred (attr$_1$, ..., attr$_n$, partition)> ,

  further selections can follow the partitioning.

- Additionally,

  SELECT <expr(attr$_1$, ..., attr$_n$, partition)>

  can be used for restructuring the result collection.

Lecture Information Systems

Method Invocations

- As shown on Slide 191, method invocations can occur everywhere, where attributes are allowed (SELECT, WHERE, GROUP BY, HAVING ...).
- In case that an invocation parameter is UNDEFINED, the method returns UNDEFINED.
- Definedness of an expression can be tested by IS_DEFINED(<expr>) or IS_UNDEFINED(<expr>).

Syntax of the SELECT-Clause

In the “clean” type system-based syntax of OQL, also the result must be typed suitably:

- objects:
  - one-column results:
    SELECT cty FROM cities cty
  - explicit creation of objects:
    SELECT city(name: . . . , country: . . .)
- Structs:
  explicit creation of structs:
  (corresponds to the notion of tuples)
  SELECT struct(name: . . . , country: . . .)

... but since people are not used to this from SQL and ignore this ...

Syntax of the SELECT-Clause

- Structs:
  Implicit creation of structs:

a comma-list of expressions of the form

1. <expression>
2. <expression> as <identifier>
3. <identifier>: <expression>

SELECT <expr$_1$>, . . . , <expr$_n$>
FROM . . .

ist allowed as an abbreviated syntax for the corresponding struct

SELECT struct(id$_1$: <expr$_1$>, . . . , id$_n$: <expr$_n$>)
FROM . . .

if in case (1) it is possible to determine id$_1$ . . . id$_n$ from <expr$_1$> . . . <expr$_n$> uniquely (in case that an <expr$_i$> is a path expression that ends in some property, the name of this property is used for id$_i$).

Expressions in OQL: Other Issues

- constructs for manipulating lists (= indexed collections):
  <list>[from:to]
- access to dictionaries:
  <dict>[<expr>]
  yields the object that is stored with the key <expr>.
- access to the (only) element of a unary set:
  ELEMENT(<unary_collection>)

  ```
  ELEMENT(SELECT x
          FROM Countries
          WHERE Code = 'D')
  ```
- transform a list into a set:
  LISTTOSET(<list>)
- remove duplicates:
  DISTINCT(<list_or_bag>)
- flattening of nested collections:
  FLATTEN(<coll_1>, ..., <coll_n>)

View-Concept in OQL

Definition of *parameterized* views:

DEFINE <name(x_1,...,x_n)> AS
<expr(x_1,...,x_n)>

```
DEFINE metropoles() AS
  SELECT *
  FROM cities c
  WHERE c.population > 1000000
```

```
DEFINE cities(x) AS
  SELECT *
  FROM cities
  WHERE c.country = x
```

SELECT ... FROM metropoles()
SELECT ... FROM cities(Germany)

- definitions are re-evaluated for each occurrence in a query (cities('D'), cities('CH')).
- definitions are persistent, until they are changed, or deleted by

  DELETE DEFINITION <name> .

Expressions in OQL

For the expression

DEFINE <name($x_1,\ldots,x_n$)> AS
<expr($x_1,\ldots,x_n$)>

where <expr($x_1,\ldots,x_n$)> is of type t, then also <name($x_1,\ldots,x_n$)> if of type t. The value of <name($x_1,\ldots,x_n$)> is the value of the evaluation of <expr($x_1,\ldots,x_n$)> in the current database state.

ODMG Language Bindings

For the described languages

- OIF (Data Format),
- ODL (Object Definition Language),
- Object Creation (OML/ODL),
- OQL (Object Querying Language)

Language Bindings to C++, Java, and Smalltalk are defined.

- specification, how the ODMG-constructs are mapped to the individual languages,
- the *implementation of (non-generic) operations of the types* is then given in the target language (Object Manipulation Language).
- from the point of view of the programmer, there are no differences between the programming language and the database-programming language:
  - a common type system with persistent and transient instances,
  - language binding for ODL/OQL respects the syntax and semantics of the target language.
  - minimal extensions to the class libraries: descriptive queries (OQL) and a transaction concept.

C++ Language-Binding

- ODMG-object types are mapped to C++-classes
- the class d_Object allows for the definition of classes, that have both transient and persistent instances (*persistence-capable classes*),
  Name convention: d_<class>.
- generic d_Collection <T> class templates.
- persistent and transient objects are handled by the same syntax.

Lecture Information Systems

C++ OQL: Selection from a Collection

- d_Collection provides a method

  ```
  query(d_Collection<type> &result,
        const char* predicate)
  ```

  that filters the collection wrt. a predicate (given as a string in the syntax of the WHERE-clause) and assigns the result to the first (Variable)parameter.

  Inside of the predicates, the variable *this* can be used for accessing the current object of the collection.

```
d_Set<d_Ref<City>> metropoles;
Cities.query(metropoles,
             “this.population > 1000000”)
```

C++ OQL

The complete functionality of OQL can be used from C++-programs (similar to the JDBC approach).

- queries are created by using objects of the class *d_OQL_Query*:

```
d_OQL_Query qry("select c from Cities
                 where c.population > 1000000");
```

- template function (not assigned to any class)

  template<class T> void
  d_oql_execute(d_OQL_Query &query, T &result)

  - executes &query and,
  - assigns the result to &result (runtime typecheck).

```
d_Set<d_Ref<City>> metropoles;
d_oql_execute(qry, metropoles);
```

- Parameterized queries in the same style as for JDBC.

Java Language Binding

Specialties:

- Compatibility with the automatical garbage collector in Java:
- *persistence by reachability/transitive persistency*:
  - an object becomes persistent, if it is referenced by another persistent object,
  - an object is removed from the database if it is not any more referenced by a persistent object.
- creation of *persistence-capable classes*:
  - existing Java-classes can be made *persistence-capable*,
  - Java-class declarations can be generated by a preprocessor from ODMG-ODL.
- special classes DCollection etc.