Prof. Dr.-Ing. Stefan Deßloch
AG Heterogene Informationssysteme
Geb. 36, Raum 329
Tel. 0631/205 3275
dessloch@informatik.uni-kl.de

Technische Universität Kaiserslautern

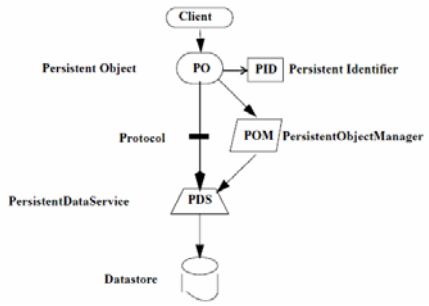
Chapter 6 – Object Persistence, Relationships and Queries

Object Persistence

- Persistent objects
  - Lifetime of a persistent object may exceed the execution of individual applications
  - Goals
    - simplification of programming model for persistent data access and management
      - no explicit interaction with data source using SQL, JDBC, ...
      - eliminate "impedance mismatch"
    - hide heterogeneity of existing data stores
      - data model, query language, API
  - Basic approach
    - application (component) interacts with objects
      - create, delete
      - access object state variables
      - method invocation
    - persistence infrastructure maps interactions with objects to operations on data sources
      - e.g., INSERT, UPDATE, SELECT, DELETE
- Variations
  - explicit vs. implicit (transparent) persistence
  - type-specific vs. orthogonal persistence

CORBA – Persistent Object Service

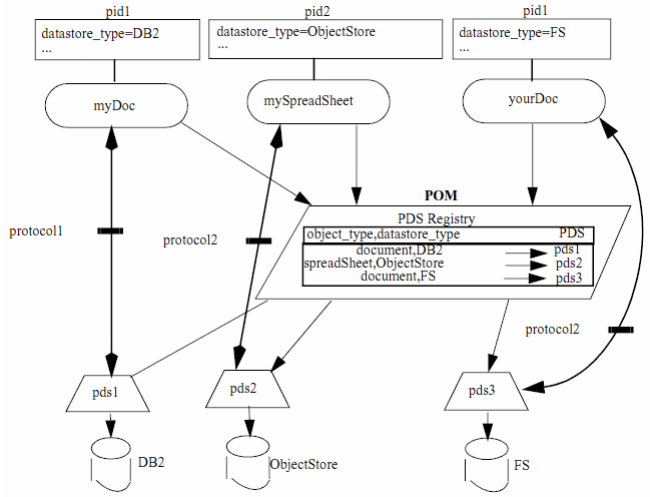
- Goal: uniform interfaces for realizing object persistence
- POS (Persistent Object Service) components
  - PO: Persistent Object
    - are associated with persistent state through a PID (persistent object identifier)
      - PID describes data location
  - POM: Persistent Object Manager
    - mediator between POs and PDS
    - realizes interface for persistence operations
    - interprets PIDs
    - implementation-independent
  - PDS: Persistent Data Service
    - mediator between POM/PO and persistent data store
    - data exchange between object and data store as defined by protocols
  - Datastore
    - stores persistent object data
    - may implement *Datastore_CLI* (encapsulates ODBC/CLI)



CORBA Persistence Model

- CORBA object is responsible for realizing its own persistence
  - can use PDS services and functions
  - implicit persistence control
    - client is potentially unaware of object persistence aspects
  - explicit persistence control
    - persistent object implements PO interface, which can then be used by the client
- Explicit persistence control by CORBA client:
  - client creates PID, PO using factory objects
  - PO Interface
    - connect/disconnect – automatic persistence for the duration of a "connection"
    - store/restore/delete – explicit transfer of data
    - delegated to POM, PDS
  - caution!: CORBA object reference and PID are different concepts
    - client can "load" the same CORBA object with data from different persistent object states

Persistent Object Manager



Persistence Protocols

- CORBA Persistence Service defines three protocols
  - Direct Access (DA) protocols
    - PO stores persistent state using so-called *direct access data objects* (DADOs)
      - CORBA objects whose interfaces only have attributes
      - defined using Data Definition Language (IDL subset)
    - DADOs may persistently reference other DADOs, CORBA objects
  - ODMG'93 protocols
    - similar to DA protocol (is a superset)
      - own DDL (ODL) for defining POs
    - ideal for OODBMS-based persistence
  - Dynamic Data Object (DDO) protocols
    - "generic", self-describing DO
      - methods for read/update/add of attributes and values
      - manipulation of meta data
    - used for accessing record-based data sources (e.g. RDBMS) using DataStore CLI interface
      - SQL CLI for CORBA
- Protocols are employed in the implementation of DOs

CORBA Queries and Relationships

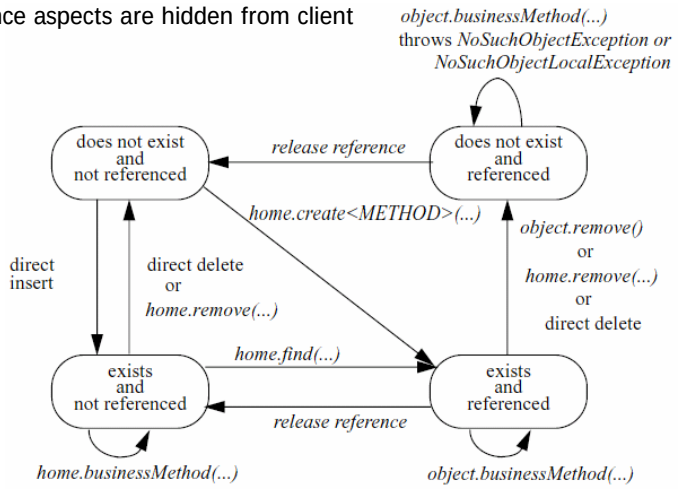
- Query Service
  - set-oriented queries for locating CORBA objects
  - SQL, OQL
  - query results are represented using Collection objects
    - iterators
- Relationship Service
  - management of object dependencies
  - relationship: type, role, cardinality

EJB – Entity Beans

- Follows *transparent persistence* approach
  - persistence-related operations (e.g., synchronizing object state with DB contents) are hidden from the client
- Persistence logic is implemented separately from business logic
  - entity bean "implements" call-back methods for persistence
    - ejbCreate – insert object state into DB
    - ejbLoad – retrieve persistent state from DB
    - ejbStore – update DB to reflect (modified) object state
    - ejbRemove – remove persistent object state

Entity Beans - Client-Perspective

- Persistence aspects are hidden from client



Container-Managed Persistence (CMP)

- Bean developer defines an *abstract persistence schema* in the deployment descriptor
  - persistent attributes (*CMP fields*)
  - relationships
- Mapping of CMP fields to DB-structures (e.g., columns) in deployment phase
  - depends on DB, data model
  - tool support
    - *top-down, bottom-up, meet-in-the-middle*
- Container saves object state, maintains relationships
  - bean does not worry about persistence mechanism
    - call-back methods don't contain DB access operations
- Manipulation of CMP fields through access methods (get*field*(), set*field*(...))
  - access within methods of the same EB
  - client access can be supported by including access methods in the remote interface
  - provides additional flexibility for container implementation
    - lazy loading of individual attributes
    - individual updates for modified attributes

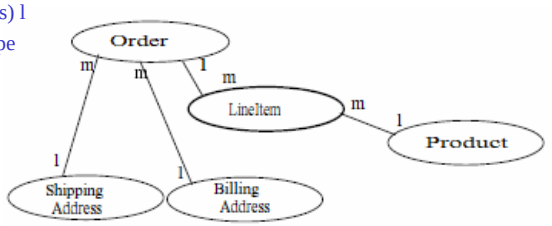
Container-managed Relationships

- Relationships can be defined in deployment descriptor
  - part of abstract persistence schema
- Relationships may be
  - uni-directional ("reference")
  - bi-directional
- Relationship types
  - 1:1, 1:n, n:m
- Access methods for accessing objects participating in a relationship
  - like CMP field methods
  - Java Collection interface for set-valued reference attributes
- Container generates code for
  - relationship maintenance
  - persistent storage
  - cascading delete (optional)

EJB Query Language

- Query language for CMP EntityBeans
  - used in the definition of user-defined Finder methods of an EJB Home interface
    - no arbitrary (embedded or dynamic) object query capabilities!
  - uses abstract persistence schema as its schema basis
  - SQL-like
- Example:

```
SELECT DISTINCT OBJECT(o)
FROM Order o, IN(o.lineItems) l
WHERE l.product.product_type
    = 'office_supplies'
```



Bean-Managed Persistence (BMP)

- Callback-methods contain explicit DB access operations
  - useful for interfacing with legacy systems or for realizing complex DB-mappings (not supported directly by container or CMP tooling)
- No support for container-managed relationships
- Finder-methods
  - have to be implemented in Java
  - no support for EJB-QL

Entity Beans

- Problems
  - distributed component vs. persistent object
    - granularity
    - potential overhead (and possible performance problems)
      - solution in EJB 2.0: local interfaces
      - but: semantic differences (*call-by-value* vs. *call-by-reference*)
    - complexity of development process
  - missing support for class hierarchies with inheritance
- Alternatives?
  - use JDBC, stored procedures
    - complex development
  - use O/R Mapping product
    - proprietary
  - implement own persistence framework
    - complex
  - JDO

JDO – Java Data Objects

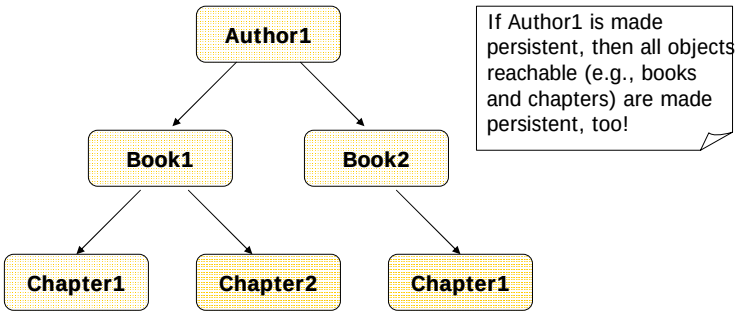
- JDO developed as new standard for persistence in Java-based applications
  - first JDO specification 1.0 released in March 2002 (after ~ 3 years) through Suns JCP (Java Community Process)
  - > 10 vendor implementations plus open-source projects
  - *mandatory features* and *optional features* in the specification (i.e., some optional features are „standardised“ → promises better portability).
- Features, elements
  - orthogonal persistence
  - native Java objects (inheritance)
  - byte code enhancement
  - mapping to persistence layer using XML-metadata
  - transaction support
  - JDO Query Language
  - JDO API
  - JDO identity
  - JDO life cycle
  - integration in application server standard (J2EE)

Orthogonal Persistence in JDO

- Object-based persistence, independent of type/class
  - Java class supports (optional) persistence (implements PersistenceCapable)
  - not all instances of the class need to be persistent
    - application can explicitly turn a transient object into a persistent object (and vice versa)
- Persistence logic is transparent for application
  - interacting with transient and persistent objects is the same
- "persistence by reachability"

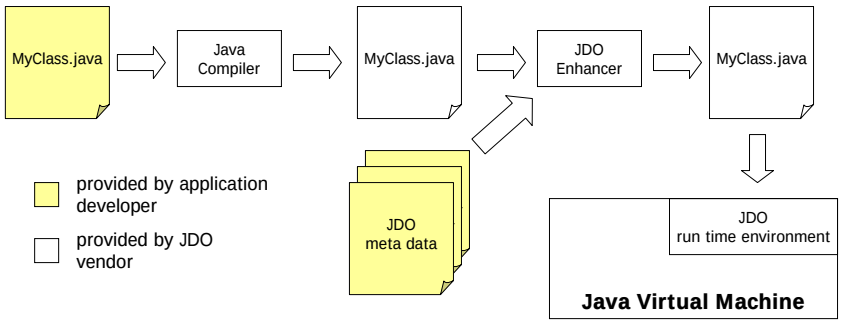
Persistence by Reachability

- all **`PersistenceCapable`** objects reachable from persistent object within an object graph are made persistent, too
- cascading delete? optional in JDO

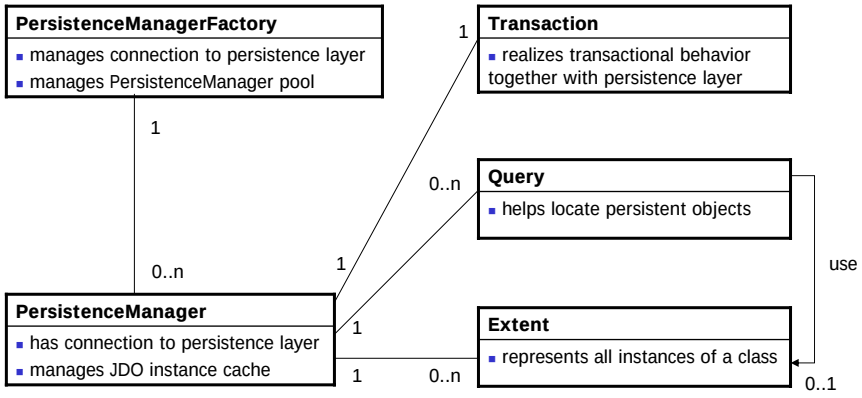


Byte-Code-Enhancement

- Java bytecode (*.class) and metadata (*.jdo)
- Same object class (now implements PersistenceCapable)
- O/R-mapping specification is vendor-specific



JDO API



PersistenceManager API - Example

```
1  Author author1 = new Author("John", "Doe");
2  PersistenceManager pm1 = pmf.getPersistenceManager();
3  pm1.currentTransaction.begin();
4  pm1.makePersistent(author1);
5  Object jdoID = pm1.getObjectId(author1);
6  pm1.currentTransaction.commit();
7  pm1.close();

8  // Application decides that author1
9  // must be deleted
10 PersistenceManager pm2 = pmf.getPersistenceManager();
11 pm2.currentTransaction.begin();
12 Author author2 = (Author)pm2.getObjectById(jdoID);
13 pm2.deletePersistent(author2);
14 pm2.currentTransaction.commit();
15 pm2.close();
```

Transactions

- JDO transactions supported at the object level
- Datastore Transaction Management (standard):
  - JDO synchronises transaction with the persistence layer
  - transaction strategy of persistence layer is used
- Optimistic Transaction Management (optional):
  - JDO progresses object transaction at object level
  - at commit time, transaction is synchronized with persistence layer
- Transactions and object persistence are orthogonal

object characteristics	transactional	non-transactional
persistent	standard	optional
transient	optional	standard (JVM)

JDO Query Language

- A JDOQL query has 3 parts
  - *candidate class*: class(es) of expected result objects
    → restriction at the class level
  - *candidate collection*: collection/extent to search over
    → (optional) restriction at the object extent level
  - *filter*: boolean expression with JDOQL (optional: other query language)
- JDOQL characteristics
  - read-only (no INSERT, DELETE, UPDATE)
  - returns JDO objects (no projection, join)
  - query submitted as string parameter → dynamic processing at run-time
  - logical operators, comparison operators: e.g. !,==,>=
  - JDOQL-specific operators: type cast using "( )", navigation using "."
  - no method calls supported in JDOQL query
  - sort order (ascending/descending)
  - variable declarations

Query

- JDO-Query with JDOQL for locating JDO instances:

```
1  String searchname = "Doe";
2  Query q = pm.newQuery();
3  q.setClass(Author.class);
4  q.setFilter("name == \"" + searchname +"\"");
5  Collection results =(Collection)q.execute();
6  Iterator it = results.iterator();
7  while (it.hasNext()){
8      // iterate over result objects
9  }
10 q.close(it);
```


JDOQL Examples

- Sorting:

```
1 Query query = pm.newQuery(Author.class);
2 query.setOrdering("name ascending, firstname ascending");
3 Collection results = (Collection) query.execute();
```

- Variable declaration

```
1  String filter = "books.contains(myBook) &&  " +
2                  "(myBook.name == \"Core JDO\")";
3  Query query = pm.newQuery(Author.class, filter);
4  query.declareVariables("Book myBook");
5  Collection results = (Collection) query.execute();
```

Java Persistence API

- Result of a major 'overhaul' of EJB specification for persistence, relationships, and query support
  - simplified programming model
  - standardized object-to-relational mapping
  - inheritance, polymorphism, "polymorphic queries"
  - enhanced query capabilities for static and dynamic queries
- API usage
  - from within an EJB environment/container
  - outside EJB, e.g., within a standard Java SE application
- Support for pluggable, third-party persistence providers

Entities

- *"An entity is a lightweight persistent domain object"*
  - entities are not remotely accessible (i.e., they are local objects)
  - no relationship with the EntityBeans concept, but co-existence
- Simplified programming model for EJB entities
  - entity is a POJO (plain old Java object)
    - no additional local or home interfaces required
    - no implementation of generic EntityBean methods needed
  - entity state (instance variables) is **encapsulated**, client access only through accessor or other methods
  - use of annotations for persistence and relationship aspects
    - no XML deployment descriptor required
- Requirements on Entity Class
  - public, parameterless constructor
  - top-level class, not final, methods and persistent instance variables must not be final
  - entity state is made accessible to the persistence provider runtime
    - either via instance variables (protected or package visible)
    - or via (bean) properties (get*Property*/set*Property* methods)
    - consistently throughout the entity class hierarchy
  - collection-valued state variables have to be based on (generics of) specific classes in java.util

Mapping to RDBMS

- Entity mapping
  - default table/column names for entity classes and persistent fields
    - can be customized using annotations, deployment descriptor
  - mapping may defines a primary table and one or more secondary tables for an entity
    - state of an entity/object may be distributed across multiple tables
- Relationship mapping
  - represented using primary key/foreign key relationships
  - table for the "owning" side of the relationship contains the foreign key
  - N:M-relationships represented using a relationship table
- Addition capabilities for constraints, column properties

Entity Inheritance

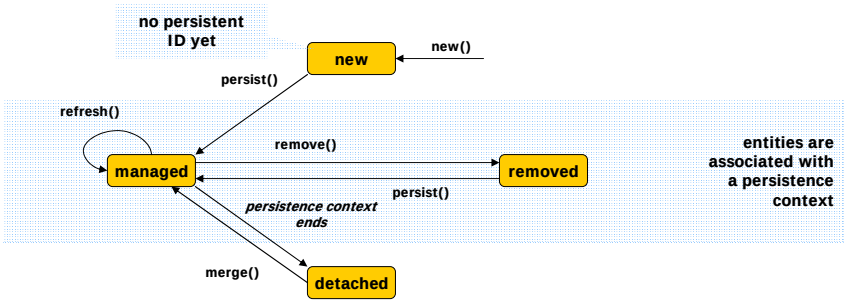
- Entities and inheritance
  - abstract and concrete classes can be entities
  - entities may extend both non-entity and entity classes, and vice versa
- Polymorphism and query support
  - references can refer to instances of subclasses
  - querying a class will return instances of subclasses
- Inheritance mapping strategies supported for the mapping
  - single table with discriminator column (default)
    - table has columns for all attributes of any class in the hierarchy
    - tables stores all instances of the class hierarchy
  - horizontal partitioning
    - one table per entity class, with columns for all attributes (incl. inherited)
    - table stores only the **direct** instances of the class
  - vertical partitioning
    - one table per entity class, with columns for newly defined attributes (i.e., attributes specific to the class)
    - table stores information about **all** (i.e., **transitive**) instances of the class

Entity Identity and Embeddable Classes

- Entities must have primary keys
  - defined at the root, exactly once per class hierarchy
  - may be simple or composite
  - must not be modified by the application
    - more strict than primary key in the RM
- Embeddable classes
  - "fine-grained" classes used by an entity to represent state
  - instances are seen as embedded objects, do not have a persistent identity
    - mapped with the containing entities
    - not sharable across persistent entities
  - current version of the specification requires/defines only basic support for embedding
    - only one level
    - no support for collections of embedded objects
    - inheritance and polymorphism of embedded classes is not required

Entity Life Cycle and Persistence

- Orthogonal persistence
  - instances of entity classes may be transient or persistent
  - persistence property controlled by application/client (e.g., a SessionBean)
  - selective persistence-by-reachability
    - defined using CASCADE options on relationships



Entity Manager

- Manages entity state and lifecycle within persistence context
  - persist(obj) -> *INSERT*
  - merge(obj) -> *UPDATE*
  - remove(obj) -> *DELETE*
  - find(class, pKey) -> *SELECT*
  - getReference(class, pKey) -> *(lazy) SELECT*
- Entity state is reflected in the database at TA commit
  - includes
    - effects of persist, merge, remove operations
    - modifications of object state
  - may also happen before commit time
    - explicit invocation of flush()
    - implicitly if automatic flush mode is in effect (default)
      - e.g., to guarantee correct query results
    - immediately when state modification occurs (proprietary!)

Entity Manager (cont.)

- Entity state is read from the database using the following model
  - persistent properties may be marked as
    - eager (default for properties) – read when object is accessed
    - lazy (default for relationships) – read when object property is accessed
  - objects access occurs in the following cases
    - invocation of find methods
    - object returned as a query result
    - object is reference through an eager relationship property, and the referencing object has been accessed
  - explicit refresh(obj) invocation
    - will refresh the object state from the database
    - updates to the object that are not (yet) reflected in the database are lost
- What happens at transaction roll-back?
  - state of entities in the application is not guaranteed to be rolled back, only the persistent state

Optimistic Locking and Concurrency

- Note: most DBMSs don't support optimistic concurrency control
- Optimistic locking is assumed, with the following requirements for application portability
  - isolation level "read committed" or equivalent for data access
  - declaration of a *version* attribute for all entities to be enabled for optimistic locking
    - persistence provider uses the attribute to detect and prevent lost updates
  - inconsistencies may arise if entities are not protected by a version attribute
- Alternative: enforce pessimistic locking semantics by choosing the appropriate isolation level

Persistence Context Lifetime

- Entity manager provides a persistence context for managed objects
  - transaction-scoped persistence context (default in EJB containers)
    - scope implicitly begins and ends with transaction
    - after TA (and persistence context) ends, persistent objects become detached
      - eager state can still be accessed and modified
      - entity needs to be explicitly merged into a new persistence context again to make changes persistent or to refresh the object state
    - appropriate for stateless session beans
  - extended persistence context
    - scope begins when entity manager is created, ends when entity manager is closed
      - e.g., when a stateful session bean instance using an EM is created/removed
    - may span multiple TAs and non-transactional invocations
      - context is automatically associated with a TA, if the session bean is
    - persist, remove, merge operations and object state modifications may occur outside the scope of a transaction
      - effects are made persistent when the next transaction commits
    - objects are not automatically refreshed when a new transaction begins!

Java Persistence Query Language

- Extension of EJB-QL
  - named (static) and dynamic queries
  - range across the class extensions including subclasses
    - a *persistence unit* is a logical grouping of entity classes, all to be mapped to the same DB
    - queries can not span across persistence units
  - includes support for
    - bulk updates and delete
    - outer join
    - projection
    - subqueries
    - group-by/having
- Prefetching based on outer joins
  - Example:

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```


Summary

- Object persistence supported at various levels of abstraction
  - CORBA
    - standardised "low-level" APIs
    - powerful, flexible, but no uniform model for component developer
      - various persistence protocols
    - explicit vs. implicit (transparent) persistence
  - EJB/J2EE Entity Beans
    - persistent components
      - CMP: container responsible for persistence, maintenance of relationships
    - uniform programming model
    - transparent persistence
  - JDO
    - persistent Java objects
    - orthogonal persistence
  - Java Persistence API
    - successor of EJB entity beans
    - standardized mapping of objects to relational data stores
    - influenced partly by JDO
    - can be used outside the EJB context as well

Summary (2)

- Query Support
  - CORBA: queries over object collections
    - no uniform query language
      - uses SQL, OQL
    - persistent object schema?
  - EJB-QL: queries over abstract persistence schema
    - limited functionality, only for definition of Finder methods
    - more or less a small SQL subset
  - JDO: queries over collections, extents
    - limited functionality
    - proprietary query language
  - Java Persistence Query Language
    - based on EJB-QL (and therefore on SQL)
    - numerous language extensions for query, bulk update
    - static and dynamic queries
  - Queries over multiple, distributed data sources are not mandated by the above approaches!