Object-Oriented Databases
Design and Implementation: OMS Avon

- Architecture
- Storage, Model and Interface Layer
- Database Modules
OMS Avon

- Java implementation of the OM data model and OML
- Storage layer
  - manages low-level storage
  - package `ch.ethz.globis.avon.storage`
- Model layer
  - implements functionality associated with the OM data model
  - package `ch.ethz.globis.avon.om`
  - package `ch.ethz.globis.avon.omal`
- Interface layer
  - provides a high-level application programming interface
  - package `ch.ethz.globis.avon.omsjp`
OMS Avon Architecture

Client Application

Value Model  Database Management  Development Tools

Caching  Driver

Object Model  Event Model  OML

Parser  Constraint Management

Processor  Module Management

Storage Management  Query Engine  Storage Model

Index Management  Transaction Management

Storage Provider

OML Parser

Process Model Management

Query Engine

Storage Model

Index Management

Transaction Management

Storage Provider
OMS Avon Project Modules

- avon-ui
- avon-omsjp
- avon-oml
- avon-event
- avon-om
- avon-storage-api
- avon-storage-db4o
- avon-common
- avon-dist
- avon-test

December 5, 2008
Michael Grossniklaus – Department of Computer Science – grossniklaus@inf.ethz.ch
OMS Avon Storage Layer

- Storage interface based on type and information units
  - type units provide metadata
  - information units store data

- Application programming interfaces for
  - create, retrieve, update and delete operations
  - schema evolution
  - creating and managing indexes
  - low-level query operators
  - transactions, concurrency control and recovery

- Extent value handles manage bulk values
- Various storage providers
Storage Module

- Consists of three main parts
- Application Programming Interface
  - used by the model layer
  - encapsulates high-level functionality and concepts
- Internal
  - internal functionality
  - common and managed functionality
- Service Provider Interface
  - interface for storage providers that provide low-level functionality
  - db4o, Berkeley DB, in-memory
Storage Module Design

<<interface>>
Storage

<<interface>>
ManagedStorage

<<interface>>
StorageInternal

StorageInternal

<<interface>>
StorageProvider

AbstractStorageProvider

InMemoryStorageProvider
OMS Avon Model Layer

- One generic Java abstraction to represent OM objects
  - single point of extensibility
  - flexibility for database evolution
  - central control for transactions and recovery
- Classes for managing generic objects
  - create, retrieve, update and delete operations
- Utility classes to access and interpret generic objects
  - cache metadata and access data from the storage layer
- Entire OM metamodel is bootstrapped
  - metamodel is expressed using OM (metacircularity)
  - different flavours of the metamodel
  - metamodel extensibility
Metamodel
Application Programming Interface

OMDatabase database = OMDatabaseManager.openDatabase("contacts.oms");
OMTransaction tx = database.beginTransaction();
OMObject object = database.objects().create(tx);
OMObject tPerson = database.namedObjects().retrieve(tx, "person");
object.dress(tx, tPerson);
OMObject tContact = database.namedObjects().retrieve(tx, "contact");
object.setAttributeValue(tx, tContact, "name", "Moira C. Norrie");
tx.commit();
Database Modules

- OMS Avon support database modules that can extend or adapt the system for special application domains.

- A database module consists of:
  - Metamodel extension to define new concepts.
  - Functionality that manages the new concepts.
  - Query language extension to interact with new concepts.

- Existing database module:
  - Main system.
  - Event system.
  - Database proxies.
  - Peer-to-peer data sharing.
  - Personal information management.
OML Query Engine

User Interface/Shell

User enters the language sentences and gets the system response or an exception.

Query Engine

Parser

AST

Query Tree Converter

Query Evaluator

Avon OM_Level
OML Query Evaluation

- **Parser**
  - used JavaCC to generate parser and lexer
  - returns an abstract syntax tree (AST)

- **Query Tree Converter**
  - uses the visitor design pattern to processes AST in post-order
  - transforms the AST into a query tree (QT)

- **Query Tree Evaluator**
  - uses the visitor design pattern to process QT in post-order
  - returns only the last result from the OML script
  - stores intermediated results in the node structure
Query Tree

```
SituatedAt
  rr(
    all $l in Locations
    having
    ($l.city = "Zurich")
  )
```

- QT nodes are atomic construct
- Used to build different database operations
  - selection, domain, range, iteration, object access
OMS Avon Interface Layer

- OMS Avon provides alternative interfaces for application development
- OMSjp
  - uniform access to heterogeneous OMS databases
  - programmatic interface based on Java
  - equivalent to JDBC in the OMS world
  - maps Java types to OM types
  - provides Java abstractions for OM system concepts
- Object Model Language (OML)
- Graphical User Interface
  - OMSjp Eclipse plug-in with graphical schema editor
  - OMSjp Browser
OMSDriver and OMSDatabase

- **OMSDriver**
  - provides database management functionality
  - abstraction of underlying implementation
  - one driver per supported platform
  - driver manager loads drivers based on configuration file
  - driver is configured via URL
  - `omsjp:platform://user/password@host:port/database`

- **OMSDatabase**
  - provides database functionality
  - create, update and delete object
  - query interface
  - schema management
  - import and export of databases
Impedance Mismatch

- Mapping the OM object-model to the Java object-model creates a new impedance mismatch
  - multiple instantiation
  - multiple inheritance

- A single object is represented by multiple classes
  - a metamodel class of type OMSObject
  - several model classes of type OMSInstance

- Application-specific instance classes can be used instead of generic instance classes
  - if data model does not multiple inheritance, Java inheritance can be used to implement application-specific model classes
  - if data model does use multiple inheritance, application-specific model classes cannot use Java inheritance
OMSObject and OMSInstance

**OMSObject**

- `getObjectID(): OMSObjectID`
- `getDressTypes(): Collection<OMSObjectType>`
- `browse(type: OMSObjectType): OMSInstance`
- `browse(class: Class<T>): T`
- `dress(type: OMSObjectType)`
- `dress(class: Class<T>)`
- `strip(type: OMSObjectType)`
- `strip(class: Class<T>)`

**OMSInstance**

- `#getBaseObject(): OMSObject`
- `setBaseObject(object: OMSObject)`
- `getInstanceType(): OMSObjectType`
- `setAttributeValue(name: String, value: Object)`
- `getAttributeValue(name: String): OMSValue`
- `executeMethod(name: String, in: Object[])`

**Contact**

- `getName(): String`
- `setName(name: String)`
- `getPhone(): String`
- `setPhone(phone: String)`
- `getFax(): String`
- `setFax(fax: String)`
- `getEmail(): URI`
- `setEmail(email: URI)`
- `getWww(): URI`
- `setWww(www: URI)`
Using Application-Specific Instances

```java
public class Contact extends AbstractInstance {
    public void setName(final String name) throws OMSException {
        this.setAttributeValue("name", name);
    }
    public String getName() throws OMSException {
        OMSString s = (OMSString) this.getAttributeValue("name")
        return s.getString();
    }
    ...
}
```

- Façade design pattern
- Access
  - method `browse()` of class `OMSObject`
  - automatically if context defines a type, e.g. in a query
- Mapping file defines instance type registrations
  - `contact = ch.ethz.globis.demo.contacts.Contact`
OMSjp in Action

// connect driver and open database
OMSDriver driver =
    OMSDriverManager.getDriver("omsjp:avon:local://localhost");
OMSDatabase db = driver.openDatabase("contacts.oms");

// retrieve and access an object
Contact contact = db.getObject(Contact.class, "name", "Moira C. Norrie");
URI www = contact.getWww();

// dress an object and set new attribute value
Person person = contact.dress(Person.class);
person.setTitle("Prof");

// retrieve a collection and perform a query
OMSBinaryCollection worksFor =
    (OMSBinaryCollection) db.getCollection("WorksFor");
Organisation organisation =
    (Organisation) worksFor.dr(contact).range().first();

// close database and disconnect driver
driver.closeDatabase();
driver.disconnect();
Using OML in OMSjp

// query for a single object
OMSValue result = db.evaluateQuery("first(all $p in Persons having ($p.name = 'Moira Norrie')");
Person moira = (Person) result;
moira.setName("Moira Norrie");

// query for a collection of object
OMSValue result = db.evaluateQuery("dom(SituatedAt rr(all $l in Locations having ($l.city = 'Zurich')")");
OMSCollection collection = (OMSCollection) result;
OMSBulkValue< ? > extent = collection.getExtent();
for (Object o: extent) {
    Contact contact = (Contact) o;
    System.out.println(contact.getName());
}
OMSjp Eclipse Plug-in
Next Week
Support for Context-Aware Data Management

• Notion of Content and Version Model
• Query Processing
• Implementation