ODMG Object Programming Language
Bindings

Objectives

- Implement the abstract model
  - mapping concepts
  - mapping types
  - mapping collections
- Adapt the model whenever necessary
  - some concepts are not supported by the language
    - interface: in C++ ==> class
    - association: in C++ and Java ==> attributes of type Ref<T>
    - keys: in C++ and Java ==> no keys!
- Integrate the OQL Query Language
Benefits

- Extend programming language with persistent capability
  - migrate programming applications in memory to databases
  - manipulate object databases with standard programming language
- Provide to programming language database functionality:
  - querying
  - transactions
  - security
  - indexing, ...

Architecture

- ODL Pre-processor
  - PL ODL Declarations
  - ODBMS Metadata
  - Database Objects
- Generated Declarations in PL
- Application Source Code in PL
- PL Compiler
  - Object Code
  - ODBMS Runtime
- Linker
  - Executable Application
  - ODBMS Runtime
I) ODMG C++ BINDING

Design Principles

- Provide a C++ library where there is a unified type system across the programming language and the database
- The C++ binding maps the ODMG Object Model into C++ through a set of persistence-capable classes
  - Uses C++ template classes for implementation
  - Based on the smart pointer ("ref-based") approach
- Main Features
  - Mapping between ODMG types and C++ types
  - Access to Meta-schema
  - C++/OQL Coupling
- The OML/C++ is C++ compliant (i.e. standard C++ compilers)
  - Compatible with Standard Template Library (STL) + persistence
Persistence-capable Classes

For each persistence-capable class $T$, a twin class $d\text{\_Ref}<T>$ is defined

- Instances of a persistence-capable class behave like C++ pointers (but OIDs $\neq$ C++ pointer)
- Instances may contain built-in types, user-defined classes, or pointers to transient data accessible through C++ references during a transaction

The C++ binding defines the class $d\text{\_Object}$ as the superclass of all persistence-capable classes

- Persistence propagation by inheritance
- Persistence declaration during creation time

The notion of interface is implicit in ODMG C++ binding

- interface: public part of C++ class definitions
- implementation: protected and private parts of C++ class definitions

Persistence-capable Classes: An O2 Example

class Person:
    public $d\text{\_Object}$ {
    private:
        char* name;
        $d\text{\_Ref<Person>}$ spouse;
        $d\text{\_Set<d\text{\_Ref<Person>}>}$ family;
    public:
        Person (char* name);
        $\sim$Person();
    };

Persistence-capable class

class Person: virtual public $o2\text{\_root}$ {
    private:
        char* name;
        $d\text{\_Ref<Person>}$ spouse
        $d\text{\_Set<d\text{\_Ref<Person>}>}$ family;
    public:
        Person (char* name);
        $\sim$Person();
    protected:
        virtual void $o2\text{\_new}()$
        virtual void $o2\text{\_read}()$
        virtual void $o2\text{\_write}()$
    };

Twin class in O2
Mapping the ODMG Object Model into C++

ODMG
structure
class C
set<T> bag<T>
array<T>
relationship T
relationship set<T>
extent of class T
keys
exception handling
collection
named operations
create, delete
basic type T

C++
struct/class
d_Ref<C>
d_Set<φ(T)> d_Bag<φ(T)>
φ(T)[] or d_Array<φ(T)>
d_Rel_Ref<φ(T)>
d_Rel_Set<φ(T)>
d_Extent <φ(T)>
not supported

C++ exception mechanism
abstract template class
C++ function members
constructor, destructor
d_T: d_String, d_Time, d_Date, d_Boolean,

ODMG C++ ODL

- The database schema is derived from the C++ class hierarchy
- All primitive C++ data types are explicitly supported except: unions, bit fields, and references
  - Objects may refer others only through a smart pointer called d_Ref
- Several predefined structured literal types are provided, including:
  - d_String, d_Interface, d_Date, d_Time, d_Interval, d_Timestamp
- A number of basic fixed-length types is also supported
- There is a list of parameterized collection classes:
  - d_Set, d_Bag, d_List, d_Varray, d_Dictionary
- C++ interpretation of the ODMG relationships
- The class d_Extent<T> provides an interface to the extent for a persistence-capable class T
### C++ ODL Basic Types

<table>
<thead>
<tr>
<th>Basic Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_Short</td>
<td>16 bits</td>
<td>signed integer</td>
</tr>
<tr>
<td>d_Long</td>
<td>32 bits</td>
<td>signed integer</td>
</tr>
<tr>
<td>d_UShort</td>
<td>16 bits</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>d_ULong</td>
<td>16 bits</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>d_Float</td>
<td>32 bits</td>
<td>single precision</td>
</tr>
<tr>
<td>d_Double</td>
<td>64 bits</td>
<td>double precision</td>
</tr>
<tr>
<td>d_Char</td>
<td>8 bits</td>
<td>ASCII</td>
</tr>
<tr>
<td>d_Octet</td>
<td>8 bits</td>
<td>no interpretation</td>
</tr>
<tr>
<td>d_Boolean</td>
<td>d_True,D_False</td>
<td></td>
</tr>
</tbody>
</table>

### C++ ODL d_Ref

- A `d_Ref` is parameterized by the type of the referenced object
  ```
  d_Ref<Professor> profP;
  d_Ref<Department> deptRef;
  profP->grant_tenure();
  deptRef = profP->dept;
  ```
- `d_Ref` is defined as a class template:
  ```
  template <class T> class d_Ref {.....}
  ```
- `d_Ref <T>` can be converted into `d_Ref<Any>`, in order to support a reference to any type, similar to that of void *
- Operators `==` and `!=` are defined to compare the objects referenced, rather than memory addresses (shallow copy semantics)
- The dereferencing operator (`→`) is used to access members of the persistent object “addressed” by the specific reference
C++ ODL Collections

- \texttt{d\_Set}, \texttt{d\_Bag}, \texttt{d\_List}, \texttt{d\_Varray}, \texttt{d\_Dictionary} are subclass of \texttt{d\_Collection}

- The collections classes are delivered as templates:
  
  \begin{verbatim}
  template <\texttt{class T}> \texttt{d\_Collection} : public \texttt{d\_Object}
  template <\texttt{class T}> \texttt{d\_Set} : public \texttt{d\_Collection}
  \end{verbatim}

- Elements therein are accessed through iterators

- Elements to be inserted in a collection must define:
  
  - default \texttt{ctor}, \texttt{copy ctor}, \texttt{dtor}, \texttt{assignment op}, and \texttt{equality op};
  - types requiring ordering must also provide the \texttt{less-than op}

---

C++ ODL Collections: Examples

\begin{align*}
\texttt{d\_Ref}<\texttt{T}> & \quad \texttt{d\_Ref}<\texttt{d\_Set}<\texttt{d\_Ref}<\texttt{T}> > > \\
\texttt{d\_Set}<\texttt{d\_Ref}<\texttt{T}> > & \quad \texttt{d\_Set}<\texttt{T}>
\end{align*}
C++ ODL Relationships

- Declaring various types of relationships between classes
  - 1-1: `d_Rel_Ref`
  - 1-n: `d_Rel_Set` or `d_Rel_List` and `d_Rel_Ref`
  - m-n: `d_Rel_Set` or `d_Rel_List` and `d_Rel_Set` or `d_Rel_List`

- Relationships are also implemented with class templates
  - `template < class T, const char* Member> class d_Rel_Ref :
    public d_Ref <T>`
  - `template < class T, const char* Member> class d_Rel_Set :
    public d_Set < d_Ref <T>>`
  - `template < class T, const char* Member> class d_Rel_List :
    public d_list < d_Ref <T>>`

- Creating, traversing and updating relationships using C++ OML

C++ ODL Relationships: Example

```cpp
extern const char _dpt[], _emps[];

class Department {
    d_Rel_Set <Employee, _dpt> emps;
};

class Employee {
    d_Rel_Ref <Department, _emps> dpt;
};

const char _dpt[] = "dpt";
const char _emps[] = "emps";
d.emps.insert_element(&e);
```

initial state

final state

```
d.emps.insert_element(&e);
```

is equivalent to `e.dpt = &d`
C++ ODL d_Extent<T>

- The class d_Extent<T> provides an interface to the extent of a persistence capable class T in the C++ binding
- The database schema definition contains a parameter for each persistent class specifying whether the ODBMS should maintain the extent for the class
- The content of this class is automatically maintained by the ODBMS
- The class includes optional support for polymorphism
- Comparison and set operations are not defined for this class

ODMG C++ OML

- Object creation
- Object deletion
- Object modification
- Object naming
- Manipulating collections
- C++/OQL
Database and Memory Cache

- The `d_Object` class allows the type definer to specify when a class is capable of having persistent, as well as transient instances.
- Pointers to transient objects in a newly retrieved persistent object must be initialized by the application.
- When a persistent object is committed, the ODBMS sets its embedded `d_Ref`s to transient objects to null.
- Binding-specific member functions `d_activate` and `d_deactivate` called when a persistent object enters, or exits respectively the application cache.

Object Creation

- Whether an object is transient or persistent is decided at creation time.
- Objects are created with the `new` operator, overloaded to accept arguments specifying object lifetime:
  - transient object
    ```
    void * operator new(size_t)
    ```
  - persisted object to be placed “near” the clustering object
    ```
    void *operator new(const d_Ref.Any &clustering, const char *typename)
    ```
  - persistent object to be placed in the database (unspecied clustering)
    ```
    void * operator new (d_Database *database, const char *typename)
    ```
- Examples:
  - `d_Ref <Person> adam = new (&base) Person;`
  - `d_Ref <Person> eve = new (adam) Person;`
Object Deletion

- Deleting a persistent object is done with the operation `delete_object()`, which is a member function of the `d_Ref` class.
- The effect of the delete operation is to cut the link between the referenced object in memory and the corresponding object into the database.
- The definitive deletion is subject to transaction commit.

Object Modification

- When a persistent object is brought into memory, its initial state is identical to the database state.
- In memory, the state is modified by updating properties or by invoking operations on it, as regular C++ ops.
- To inform the runtime ODBMS that the state has been modified, we need a special function `mark_modified` of class `d_Object`.
  - In some implementations, it is possible to detect that the state of an object has been modified.
- The ODBMS updates the database at transaction commit time.
To retrieve objects in a database we need persistent names. These names are the root of the persistency. A name can identify a unique object or a collection of objects. Object names can be assigned and modified at run-time. The facility for naming is related with the class d_Database. Objects of this class are transient and their semantics do not include actual database creation. Other supported operations include opening and closing an existing database.

Examples of Object Naming

```java
.....
session.set_default_env;
session.begin(argc,argv);
base.open("origin");
trans.begin();
adam = base.lookup_object("adam");
d_Ref<Person>
eve = new (&base) Person("eve");
adam -> spouse = eve;
d_Ref <Person>
cain = new (&base) Person("cain");
adam->family.insert_element(cain);
trans.commit;
base.close();
session.end();
```
Iterating on Collections

- To iterate on collections a template class `d_Iterator<T>` is defined
- Iterators can be obtained through the `create_iterator` method defined in the class `d_Collection`
- The `get_element` function returns the value of the currently “pointed-to” element in the collection (if any)
- The `next` function checks the end of iteration, advancing the iterator and returning the current element
- Iterators mean great flexibility:
  - collection type independence
  - element type independence
  - generality and reusability

```cpp
d_Iterator<d_Ref<Person>> iter = adam->family.create_iterator();
d_Ref<Person> s;
while (iter.next(s)) {
    .......
}
```

OQL Embedded in C++

- Queries are supported in two forms: collections vs database
- Directed queries on collections
  - made through the `query` method of collection classes
  - they require as parameter the query predicate in a string; the syntax is that of the OQL `WHERE` clause
  - the result is returned in a second collection class
- Queries on the database
  - first an object of type `d_OQL_Query` must be constructed
  - then the query must be executed with the `d_oql_execute` function
Query Examples

\[
d_{\text{Set}} < d_{\text{Ref}} < \text{Student} > > \text{Students, mathematicians, old;} \]

\[
\text{Students.query} (\text{mathematicians,}
\quad "\text{exists c in this.takes: c.subject = 'Math'}");\]

\[
d_{\text{OQL_Query}} \ q1( "\text{select s}
\quad \text{from s in $1}
\quad \text{where exists c in s.taken_by:}
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad s.\text{age} > c.\text{taught_by}.\text{age}");\]

\[
\text{query} = q1 << \text{Students};
\text{d_oql_execute} (\text{query, old});
\text{query} = q1 << \text{others};
\text{d_oql_execute} (\text{query, old});\]

Queries on the Database

- \text{d_oql_execute} returns either a collection, or an iterator on a collection (both are type-checked)
- queries can be \text{constructed incrementally} (using \text{<<})
- queries can be parameterized
  - parameters in the query string are signified with $i$
  - the shift-left operator \text{(<<<)} is used to provide values as right-hand operands
  - the clear method re-initializes the query, so that it can be reused with different values
  - upon successful execution of a query the values are automatically cleared
The ODMG database schema can be accessed through appropriate interfaces defined in the C++ binding.

- The schema-access API is in effect an object-oriented framework.
- C++ specific ODL extensions are included in the schema-access API (the C++ ODL is a superset of the ODMG ODL).
- Currently only the `read` interface is defined; the `write` interface is vendor-dependent.
- Schema access is necessary not only for inspecting a database schema, but also for manipulating it at run-time (e.g. extents).

The ODMG Schema Access Class Hierarchy:

```
- d_Scope
  - d_Meta_Object
    |  - d_Module [d_Scope]
    |  - d_Type
    |    |  - d_Class [d_Scope]
    |    |  - d_Ref_Type
    |    |  - d_Collection_Type
    |    |    |  - d_Keyed_Collection_Type
    |    |    |  - d_Primitive_Type
    |    |    |  - d_Enumeration_Type [d_Scope]
    |    |    |  - d_Structure_Type [d_Scope]
    |    |  - d_Alias_Type
    |    |  - d_Property
    |    |  - d_Relationship
    |    |  - d_Attribute
    |  - d_Operation [d_Scope]
    |  - d_Exception
    |  - d_Parameter
    |  - d_Constant
   - d_Inheritance
```
II) ODMG Java BINDING

Main Features

- Unified type system between the Java language and the database
  - No modifications to Java syntax
- Java classes can be made persistence-capable
  - **importation**: Java classes $\Rightarrow$ ODMG ODL
  - **exportation**: ODMG ODL $\Rightarrow$ Java classes
- Automatic storage management semantics of Java
  - transient, dynamic persistency
- Transparent access and update to the database
- Collections
- Query facilities
Persistency

- Persistency is by reachability from persistent roots
  - All the persistent objects are directly or transitively attached to the persistent root
- At commit time if an object is connected to a persistent root then the object is written automatically into the database
- The class Database allows to define persistent roots and to retrieve them:
  - `bind` (Object, String) // give a name
  - `lookup` (String) // retrieve an object by its name

Retrieving and Creating Objects: Example

```java
Database base = new Database;
base.open("origin");
Transaction trans = new Transaction;
trans.begin();
Person adam = base.lookup("adam");
base.bind(adam, "adam");
trans.commit();
base.close();
```

```java
Database base = new Database;
base.open("origin");
Transaction trans = new Transaction;
trans.begin();
Person adam = base.lookup("adam");
Person eve = new Person("eve");
Person cain = new Person("cain");
adam.spouse = eve;
adam.family.insertElement(cain);
trans.commit();
base.close();
```
public class Database {
    public static Database open ( String name) 
            throws ODMGException; 
    public void close ( ) 
            throws ODMGException; 
    public void bind (Object obj, String name); 
    public Object lookup (String name) 
            throws ObjectNaneNotFoundException; 
}
Mapping from ODMG to Java

ODMG

Date
Struct or tuple
Set (t)
Bag(t)
List (t)
Array (t)
Iterator

Java

java.sql.Date
Java Class
interface Set
interface Bag
interface List
array type [ ] or Vector
Enumeration

ODMG Java OML

- **Same Java syntax** for manipulating persistent or transient objects
- **Object creation** is done with the `new` operator
  - If an object type intermixes persistent data and transient data only persistent data is loaded in memory, transient data is set to default
- **There is no object deletion** since persistency is by reachability
- **Modification** of objects is done with standard Java operations
  - At commit time modifications are written back automatically
- **Collection** interfaces
  - Collection, Set, Bag, List
- The interface *Transaction* provides standard operations as: `begin()`, `commit()`, `abort()`
  - Operations are also defined for using the possibility of multithreading in Java, such operations are: `join`, `current`, etc.
- **Java/OQL Coupling**
  - similar to the C++ binding: `bind` is used instead of `<<"}
public interface Collection {
    public int size();
    public boolean isEmpty();
    public removeElement();
    ........
    public Collection query(String predicate);
}

public interface Set extends Collection {
    public Set union(Set otherSet);
    public Set intersection(Set otherSet);
    .......
    public boolean properSubsetOf(Set otherSet);
    .......
}
Interface List

```java
public interface List extends Collection {
    public void add (int index, Object obj)
        throws ArrayIndexOutOfBoundsException;
    public void put (int index, Object obj)
        throws ArrayIndexOutOfBoundsException;
    ........
    public Object get (int index)
        throws ArrayIndexOutOfBoundsException;
    public List concat (List other);
}
```

OQL Embedded in Java

- Directed queries on collections
  - Filtering a collection with the operation query defined in the interface Collection

- Example:
  SetOfObject mathematicians;
  mathematicians =
      Students.query("exists c in this.takes: c.subject='Math'");

- Queries on the database
  - Using the class OQLQuery (String question) and the operations bind(Object parameter) and execute()

- Example:
  Set Students, young;
  OQLQuery q1;
  q1 = new OQLQuery ("select s from s in $1 where s.age = $2");
  q1.bind(Students); q1.bind(25);
  young = (Set) q1.execute();
**General Architecture**

- **Java Application**
  - read, write
- **Java Object Cache**
- **ODBMS Engine**
- **Object Database**

**Conclusion**

- Java is well adapted to become a database programming language.

- Some features of the ODMG model are not supported:
  - Relationships & collections are not parameterized by a type or a class
  - Extents & Keys

- The concepts of Java interface and implementation allow the possibility to define very powerful implementations for Set and List with HashSet, ArraySet, ArrayList, LinkedList, etc.
V) REFERENCES

- C. Delobel: “The ODMG PL Bindings”, Course Slides, University of ORSAY
- B. Amann: “Object-Oriented Database Systems”, Course Slides, CNAM, Paris