VI) OBJECT DATABASES

Value vs. Object-based models

- **Value-based models**: Relational, N1NF & complex values/objects
  - An object is represented by its value
  - An object is identified by a part of its value
  - More and more richer structuring
- **Object-based models**: ADTs, object ids
  - An object has a value (state)
  - An object is identified by an id which is independent from its value
  - More powerful structuring compared to complex value models
- **NOTE**: The introduction of object ids
  - Allow to express (objects) or not (values) data sharing
  - Leave to the system (objects) or the programmer (values) data identification
  - Facilitate (objects) or not (values) redundancy elimination and management of updates anomalies
A Model for Objects: Complex Values + Ids

- Set of atomic values: \( D = \cup D_i \), where \( D_i \) are atomic domains such as strings, integers, reals, etc.
- Set of Attribute names \( A \)
- Set of Object Identifiers \( O \)
- An object is pair \((o, v)\) where \( o \) in \( O \) and \( v \) a value defined as follows:
  - each element \( v \) of \( D \) is a value
  - each element \( o \) of \( O \) is a value
  - if \( v_1, v_2, ..., v_n \) are values and \( a_1, a_2, ..., a_n \) are attribute names then \([a_1:v_1, a_2:v_2, ..., a_n:v_n]\) is a tuple value
  - if \( v_1, v_2, ..., v_n \) are values then \( \{v_1, v_2, ..., v_n\} \) is a set value and \(<v_1, v_2, ..., v_n>\) is a list value
- The set of values given a subset of \( O \) is denoted \( \text{val}(O) \)
- NOTE: Values of objects are constructed from atomic values and other object identifiers using tuple, set and list constructors.

Complex Objects: Examples

\((o_1, \{\text{Name:”Claude Monet”, Style:”Impressionism”, Influenced_by:\{o_4\}, Artifacts:\{o_2,o_3\}\}})\)
\((o_4, \{\text{Name:”Edouare Manet”, Style:”Impressionism”, Influenced_by:\{\}, Artifacts:\{\}\}})\)
\((o_5, \{\text{Name:”Pierre Renoir”, Style:”Impressionism”, Influenced_by:\{o_4\}, Artifacts:\{o_6,o_7\}\}})\)
\((o_2, \{\text{Name:”Haystacks”, Museum:\{\text{Name:”San Diego Museum of Art”, Address:”US”}\}\}})\)
\((o_3, \{\text{Name:”Wheatstacks”, Museum:\{\text{Name:”Art Institute of Chigago”, Address:”US”}\}\}})\)
\((o_6, \{\text{Name:”The Parisian”, Museum:\{\text{Name:”Nat. Museum of Wales”, Address:”US”}\}\}})\)
\((o_7, \{\text{Name:”The Laundress”, Museum:\{\text{Name:”Art Institute of Chigago”, Address:”US”}\}\}})\)
Object Identity and Equality

- Two objects having the same identifier are **identical**:
  - \((o, [\text{Name}:\text{"Nat. Museum of Wales"}])\) and
  - \((o, [\text{Name}:\text{"National Museum of Wales"}])\) are identical
  - \((o, [\text{Name}:\text{"Nat. Museum of Wales"}])\) and
  - \((o, [\text{Name}:\text{"Nat. Museum of Wales"}, \text{Address}:\text{"US"}])\) are identical

- Two objects with different ids but the same value are **shallow equal**:
  - \((o_1, [\text{Name}:\text{"Nat. Museum of Wales"}])\) and
  - \((o_2, [\text{Name}:\text{"Nat. Museum of Wales"}])\) are equal

Object Deep Equality

- **Definition**:
  - Two equal objects are **deep equal**
  - Two objects with the same attributes where their values are either equal or deep equal objects, are **deep equal**

- **Example**:
  - \((o_1, [A:o_3, B:\text{"b"}])\)
  - \((o_2, [A:o_4, B:\text{"b"}])\)
  - \((o_3, 5)\)
  - \((o_4, 5)\)
Object References and Data Sharing

- Edouare Manet is an early impressionist artist which has influenced both Claude Monet and Pierre-Auguste Renoir
  
  \[ o_1, \text{Name:"Claude Monet", Style:"Impressionism", Influenced_by:{o_4}, Artifacts:{o_2, o_3}} \]
  
  \[ o_4, \text{Name:"Edouare Manet", Style:"Impressionism", Influenced_by:{}, Artifacts:{}} \]
  
  \[ o_5, \text{Name:"Pierre Renoir", Style:"Impressionism", Influenced_by:{o_4}, Artifacts:{o_6, o_7}} \]
  
- “Wheatstacks” of Monet and “The Laundress” of Renoir are both paintings kept at the Art Institute of Chicago
  
  \[ o_6, \text{Name:"Wheatstacks", Material:"Oil on canvas", Date:"1865", Museum:o_8} \]
  
  \[ o_7, \text{Name:"The Laundress", Material:"Oil on canvas", Date:"1880", Museum:o_8} \]
  
  \[ o_8, \text{Name:"Art Institute of Chicago", Address:"US"} \]

Objects Continued …

- An object is defined by:
  
  1. Its identifier
  2. Its value
  3. Its structure
  4. Its operations (methods)
- The state (memory) of an object is defined by 1 and 2
- The type/class of an object is defined by 3 and 4
A Classification of Concepts

CLASS

FACTORY

TYPE

EXTENSION

OBJECT

VALUE

IDENTIFIER

METHODS

- **NOTE:** The extension of a class (i.e., the set of its instances) is not always persistent (i.e., stored in the DBMS)

---

Object: Structure and ...

- Set of atomic Types: STRING, INT, REAL
- Set of Attribute names A
- Set of Class names C
- The Type of a Value is defined as follows:
  - each atomic type is a type
  - each element of C is a type
  - if t₁, t₂, ..., tₙ are types and a₁, a₂, ..., aₙ are attribute names then [a₁:t₁, a₂:t₂, ..., aₙ:tₙ] is a tuple type
  - if t is a type then {t} is a set type and <t> a list type
- The set of types given a subset of class names C is denoted \(\text{types}(C)\)
- **EXAMPLE:** The types of the values Artist, Artifact and Museum are:
  - ARTIST [Name:STRING, Style:STRING, Artifacts:{ARTIFACT}, Influenced_by:{ARTISTS}]
  - ARTIFACT [Name:STRING, Material:STRING, Date:STRING, Museum:MUSEUM]
  - MUSEUM [Name:STRING, Address:STRING]
Object: ... and Behavior

- The operations (methods) which can be applied in an object are defined in its class by denoting their signatures:
  - The name of the method
  - The type of the method arguments
  - The type of the method result

- To execute a method, a message is send to an object containing the method name and arguments: we don’t opere directly on objects
  - Object ➜ Method(v1, ..., vn)

- EXAMPLE:
  - In the class ARTIST we have the method AddArtifact of signature: ARTIFACT ➜ ARTIST
  - In the object Monet of the class ARTIST we send the message: Monet ➜ AddArtifact(Morning-Snow-Effect) where Morning-Snow-Effect is an object of the class ARTIFACT

Data Encapsulation

- The state of an object (may include a subset of its methods) is entirely encapsulated within the object
- Objects are manipulated only through their external visible interface (i.e., public methods)
- Motivations: (Support of ADTs, Modular programming)
  - Definition of user-oriented concepts
  - Protection from non authorized manipulation of objects
  - Separation of objects interface specification and implementation
- Two complementary encapsulation views: PL vs. DB
Encapsulation: PL vs DB View

- **PL View**
  - An object has a specification and an implementation
  - The specification is a set of operations applied to the object
  - The implementation is the structure of the physical data, representing object memory, and the procedures realizing the operations
  - The programmer can see both the data structure and the operations
  - The client (user) see only the interface (object specification)

- **DB View**
  - An object encapsulate the notion of data and behavior
  - The data is the memory of the object
  - The behavior is the operations associated with the object
  - In some cases data encapsulation can be violated (e.g., queries)

Class Inheritance

- **ARTIST** inherit, is a sub-class, specialize **PERSON**
- Class/Type Inheritance can be
  - Inferred during compilation: deduction of types, sub-types by the system
  - Explicitly declared: refined attributes/methods are automatically inherited

**PERSON**

- Type [Name:STRING, Live-Time:DATE]
- Methods
  - Age: DATE ➔ INTEGER

**ARTIST**

- Type [Name:STRING, Live-Time:DATE, Style:STRING, Artifacts:{ARTIFACT}]
- Methods
  - Age: DATE ➔ INTEGER
  - Add_Artifact: ARTIFACT ➔ ARTIST

**ARTIST** inherits **PERSON**

- Style:STRING, Artifacts:{ARTIFACT}
Modeling Inheritance [R. BRACHMAN 83]

- **Substitution Semantics** (PL view)
  - An Artist is a Person since each time we can substitute a person with an Artist
- **Descriptive Semantics** (Logic view)
  - An Artist is a Person since the fact that a Person satisfying certain logic formulas allows us to infer that is an Artist
- **Inclusion Semantics** (Set-theory view)
  - An Artist is a Person since the set of Artists is a subset of Persons
- **Specialization Semantics** (Conceptual view)
  - An Artist is a Person since it has all the characteristics of a Person plus some additional ones
- **Syntactic Characterization in ODBMS: Covariant Model**
  - Structural part (types)
  - Behavioral part (methods signatures)

Structure: Class Inheritance & Sub-Typing

- A class hierarchy is a triple \((C, \sigma, \prec)\), where \(C\) is a subset of class names, \(\sigma\) a function returning for each class its type in \(\text{types}(C)\) and \(\prec\) a partial order in \(C\)
- Given a class hierarchy \((C, \sigma, \prec)\), the subtyping relation \(\leq\) defined in \(\text{types}(C)\) is the smaller partial order defined as follows:
  - if \(c, c'\) in \(C\): \(c \prec c'\) then \(c \leq c'\)
  - if \(t \leq t'\) then \(\{t\} \leq \{t'\}\) and \(\langle t \rangle \leq \langle t' \rangle\)
  - if for all \(j\) in \([1..m]\) exists \(i\) in \([1..n]\): \(b_i = a_i\) and \(t_i \leq t'_j\) then \([a_1: t_1, ..., a_n: t_n] \leq [b_1: t'_1, ..., b_m: t'_m]\)
- A class hierarchy \((C, \sigma, \prec)\) is well-formed if for all \(c, c'\) in \(C\): \(c \prec c'\) then \(\sigma(c) \leq \sigma(c')\)
Behavior: Covariant Order

- Assuming a well-formed class hierarchy \((C, \sigma, \prec)\) where each class \(c\) in \(C\) has a set of associated methods
  - The signature of a method is an expression of the form \(m: c \times t_1 \times t_2 \times \ldots \times t_n \rightarrow t\) where \(m\) is the method name, \(c\) is the class on which the method is defined and \(t_1, \ldots, t_n\) the types of its arguments and \(t\) the type of the result in \(\text{types}(C)\).
- The set of method signatures associated with the class of a hierarchy is denoted \(M\).
- Given a well-formed class hierarchy \((C, \sigma, \prec)\) and a set of associated method signatures \(M\), the signatures are well-formed if:
  - For all \(m\) in \(M\) where \(m\) is defined in two classes \(c, c'\) in \(C\), \(c \prec c'\), \(m: c \times t_1 \times t_2 \times \ldots \times t_n \rightarrow t\) and \(m: c' \times t'_1 \times t'_2 \times \ldots \times t'_m \rightarrow t'\) we have \(\forall i \in [1..m] \exists j \in [1..n] \ t_i \leq t'_j\) and \(t \leq t'\).

**NOTE:** in the contravariant order \(t \leq t'\) but \(t'_j \leq t_i\)

The Use of Inheritance

- **Extensibility**: incremental specification of the schema using class refinement
- **Reusability**: sub-classes inherit the methods of their super-classes
  - Inherited methods have the same name and code (i.e., no code duplication)
- **Modularity**: modifications of the methods of a class affect only the behavior of its sub-classes
- **Overloading** and **Late Binding**: multiple definitions of a method in different classes using compatible signatures
  - Overloaded methods have the same name but different code
- **NOTE**: Overloading and Late Binding introduce a certain degree of polymorphism
Overloading and Late Binding

- We need to display three types of objects:
  - Artists
  - Artifacts
  - Museums

- How we can implement it?
  - We define a hierarchy of classes/types
    - Object_for_Display
      - Artist
      - Artifact
      - Museum
  - We write a dummy method `display` of class `Object_for_Display`
  - We overload (redefine) for each of the `Object_for_Display` subclasses a method with the same name

- How the system will find the good method code?
  - During execution the type of objects is detected (dynamic typing) and the name of the method is bind to the corresponding code

Overloading vs Overriding

- **Overloading** of a method:
  - method has different signature
    - `m(a: integer)` in class `c`
    - `m(a: integer, b: string)` in class `c' < c`

- **Overriding** of a method
  - method has different behavior
    - `m(a: integer)` in class `c` { return a; }
    - `m(a: integer)` in class `c'` { return a + 1; }
Aspects of Polymorphism

- Traditional programming languages are monomorphic
  - For each variable or function a unique type is statically determined during compilation (e.g., Pascal)
- In polymorphic languages functions may have several types
  - For a family of different types we can use the same function (name, code)
- Parametric Polymorphism: family of types with similar “structure”
  - The type is given as parameter (e.g., C++ templates)
- Inclusion Polymorphism: family defined by a sub-typing order
  - For each \( c \prec \text{PERSON}, \) the method Age: \( c \rightarrow \text{INTEGER} \)
- Ad-hoc Polymorphism: the case of overloading (e.g., ADA and C++)
- Unlike ad-hoc polymorphism, parametric and inclusion concern an infinite set of types (universal polymorphism)

Type and Class Semantics

- Assuming a well-formed class hierarchy \((C, \sigma, \prec)\) a population function of disjoint oid’s is a function \( \pi_d: C \rightarrow 2^O \) associating to each class name a finite set of oid’s in \( O \) such that
  - if \( c, c' \in C \): \( c \neq c' \) then \( \pi_d(c) \cap \pi_d(c') = \emptyset \)
- An oid assignment is a function \( \pi \) such that for each class \( c \):
  - \( \pi(c) = \bigcup \{ \pi(c') | c' \prec c \} \)
- Given an oid assignment function \( \pi \), the interpretation of a type \( t \) in \( \text{types}(C) \), denoted \( \text{dom}(t) \), is defined as follows:
  - for each atomic type \( b \): \( \text{dom}(b) = b \) (concrete types)
  - for each class \( c \in C \), \( \text{dom}(c) = \pi(c) \)
  - \( \text{dom}(\{t\}) = \{v_1, ..., v_j \} \) if \( j \geq 0 \), and for all \( i \in [1..j] \) \( v_i \in \text{dom}(t) \)
  - \( \text{dom}(<\tau>) = \{<v_1, ..., v_j> | j \geq 0 \}, \) and for all \( i \in [1..j] \) \( v_i \in \text{dom}(t) \)
  - \( \text{dom}([a_1:t_1, a_2:t_2, ..., a_k:t_k]) = \{[a_1:v_1, a_2:v_2, ..., a_k:v_k] | l \geq 0 \} \) and for all \( i \in [1..k] \) \( v_i \in \text{dom}(t) \)
A Formal Definition of Object Databases

- A schema $S$ is a quintuple $(C, \sigma, \epsilon, M, G)$, where $(C, \sigma, \epsilon)$ is a well-formed class hierarchy, $M$ is a set of well-formed method’s signatures and $G$ a set of names (e.g., relations) with an associated type $\sigma(g)$ in $\text{types}(C)$ for all $g$ in $G$.

- An instance $I$ of the schema $S$, also called database, is a quadruplet $(\pi, \upsilon, \mu, \gamma)$ where:
  - $\pi$ is the population function for each class: $O = \cup \{\pi(c) \mid c \in C\}$
  - $\upsilon$ is a value assignment function for each object $o$ of a class $c$: if $o$ in $\pi(c)$ then $\upsilon(o)$ in $\text{dom}(\sigma(c))$
  - $\mu$ is a value assignment function for each method $m: c \times w \rightarrow t$ of a class $c$: if $m$ in $M$ then $\mu(m: c \times w \rightarrow t)$ in $(\text{dom}(c) \times \text{dom}(w))^{\text{dom}(t)}$
  - $\gamma$ is a value assignment function for each name $g$ in $G$: if $g$ in $G$ then $\gamma(g)$ in $\text{dom}(\sigma(g))$.