Object-Module Design

- Classic object-orientation
  - Group data and applicable operations together.
  - Encapsulate identity, data, and behavior details under an interface of exported services.
  - Establish inheritance hierarchies among encapsulated units.
- Classic database-system services
  - Decide how to manage concurrency.
  - Add transaction processing.
  - Specify views and their update and authorization characteristics.

Object Module ~ Generalized ADT

An object module is a named pair $N = (D, B)$.

- $N$ is the name
- $D$ is a data description
  - intent (set of possible elements)
  - extent (set of current instances)
  - constraints
- $B$ is a behavior description
  - events and conditions to which the behavior responds
  - states or circumstances under which the response happens
  - a description of the responses
  - services = the responses associated with events and conditions
    - uniform service availability
    - nonuniform service availability
Object Modules – Examples

- A single object set (lexical or nonlexical)
- A “built-in” object set (e.g., Integer)
- A passive high-level object set with included lexical object sets (a typical database record, flat or nested)
- An object set with a given state net
- An object set that includes both passive object and relationship sets and a state net.

Special Types of Object Modules

- Class (typical)
  - an implicit intent and (possibly) an explicit extent
  - uniformly available two-way interactions
- Type
  - intent (extent implicit and same as intent)
  - uniformly available two-way interactions (operations)
- Variable
  - extent with at most one value
  - query, add, remove, and modify
  - usually declared as a specialization of a type
- Value
  - extent with one value
  - query
Encapsulation

- **Object Integrity**
  - immutable identity (be careful with lexicalized objects)
  - obeys its behavior specification (e.g., no arbitrary state change)

- **Interface Specification**
  - implicit or explicit declaration of services
  - tradeoffs for query processing

- **Implementation Independence**
  - separates specification from implementation
  - view one way, but implement in another (more efficient) way
  - overriding and late binding for polymorphic variations

We should observe these principles even if not language supported.

ORM Computations

- Table Implementation
- Virtual Implementation
- Interaction Implementation
- No Implementation
Table Implementation

- Advantage: faithful to the ORM diagram.
- Disadvantage: must store computable information.
- Note: for some applications, the computation might be expensive, needed often including inverse computations, and the data may be relatively static.

Virtual Implementation

- Advantage: faithful without storing computable data
- Disadvantage: must also provide all inverse computations.
- Scheme generation: data not stored cannot be redundant.
  - Room(RoomNr, Cost, Last Rate Change Date, Length At Current Rate)
  - Currency Exchange(Currency, Cost, Exchange Rate, Exchange Amount)
Interaction Implementation

- Advantage: need not store computable data.
- Disadvantage: not faithful to the ORM diagram.
- Note:
  - These interactions become methods.
  - Often, the inverse transformations are never needed.

No Implementation

- Advantage: no service to provide.
- Disadvantage: the client process must do any needed computation.
  - The client process can retrieve needed information and execute the computation.
  - If no client needs the service, everyone wins.
Simple Enumeration

Partitioned Category Specialization
Unconstrained Category Specialization

Chapter 11 - 13

Generated Value-Set Types

Chapter 11 - 14

Note: We allow duplicates in facilities to illustrate the correspondence to regular expressions.
Data Structures

Exchange-Rate Table

Currency Exchange Rate

Array [Currency] of Exchange Rate

Concurrency in OSM

- Ontological point of view
  - Objects naturally behave independently and thus concurrently (inter-object concurrency).
  - Objects may perform multiple tasks simultaneously (intra-object concurrency).

- Managing concurrency (highly complex)
  - Fortunately, databases provide transaction processing.
  - Distributed database systems manage transaction processing in a multiprocessor environment.

- State-net patterns can lead to efficient implementations
ADT State Net

- Set of objects (represented by an ORM diagram)
- Set of uniformly available services
  - single transitions, with event triggers
  - no states (zero-state state net)
- At most one thread of control.

ADT State-Net Example

```
@ make_reservation(n: Name, a: Address, d: ArrivalDate, y: NDays, r: RoomNr)
record Guest information and the reservation

@ cancel_reservation(g: GuestNr, r: RoomNr, a: ArrivalDate)
remove reservation and Guest
```
Patterns Easily Transformed to an ADT State Net (Zero-State State Net)

- An interaction initiates the pattern.
- There is a single thread of control.
- A transition terminates the pattern (and the thread of control).

Single-State State Net (Transformable to ADT State Net)

Assume: We need to process only one request at a time.
Patterns Easily Transformed to a Single-State State Net

- An interaction initiates the pattern based on an object being in an initial state (e.g., Exists).
- There is a single thread of control.
- The thread of control starts and ends with the same state (e.g., Exists).
Nonuniform Service Availability
(Transformable to Single-State State Net)

As before, if we need to process only one request at a time, we can translate this to an ADT state net (zero-state state net).
Queued Concurrent Requests

- Suppose we have several concurrent requests for services.
- Suppose also that we can buffer these requests in a queue and service them sequentially.
- Then, we can simulate some concurrent state-net patterns.
  - Single-state state nets with multiple objects (inter-object concurrency)
  - Single-state state nets with multiple threads (intra-object concurrency)

Using Transaction Processing to Simulate Concurrency in State Nets

- Transactions preserve atomicity and integrity.
- Services are often exactly the units we should protect in transactions.
- OSM-L provides `start`, `commit`, and `abort`.

```plaintext
@ cancel reservation (nr: GuestNr)
start;
GuestNr(nr).Guest := { };
commit;
```
Intra-Object Concurrency with Nonuniform Service Availability
(Transformable to Single-State State Net)

Chapter 11 - 27

Intra-Object Concurrency with Nonuniform Service Availability
Transformed to Single-State State Net

Chapter 11 - 28
Multiprocessing
(Actual Concurrency)

- Multiple processors in a distributed processing system
  - highly complex
  - fine-tuning such a system (beyond scope)
- Maximizing concurrency
  - allows multiprocessing system to assign threads of control to different processors
  - two ways to increase concurrency
    - increase the number of objects (e.g., remove 0:1 participation constraints)
    - increase the number of threads (e.g., spawn more threads of control or redesign state net with forks and joins)
  - do not arbitrarily maximize concurrency; look for large benefits

Visibility

- Public
  - services (interactions)
  - implicit database operations (exposed ORM components)
- Read Only
  - implicit database operations
  - only query operations
- Hidden
  - ORM, OBM, OIM components
  - explicit and implicit operations not available
Visibility Example

object module Reservation Clerk includes

@ new reservation;
@ form filled;
@ cancel reservation;

read only

Guest [1:*] has reservation for Room [0:*]
on ArrivalDate [1:*] for NrDays [1:*];

hidden

Guest [1] has Name [1:*];
@ new reservation then ...
... end;

Object Module Views

- Different clients (of an object module) may have different visibility requirements.
  - A proprietor may wish to view all available data.
  - But guests should only be able to see their own information and general information about which rooms are reserved.
- Views can accommodate multiple visibility requirements.
- Views can provide:
  - authorization
  - derived data
- Views have one or more base object modules.
Authorization

• An authorization clause in an object module lets us specify who may use the view.
• An authorization clause designates an object set that includes the identity of clients authorized to use the view.
  – An object set in the application.
    • Example: “Proprietor” limits the visible services and and data of the object module to proprietors.
    • We can introduce an object set for this purpose.
  – Variable specialization of an object set in the application.
    • Example: “x:Guest” limits the use of the object module to the subset of guests in x—likely only one guest.
    • In this case, the view itself may depend on x (e.g., we may have the constraint “GuestNr(y).Guest = x” where y is the guest number of the client currently using the view).

Derived Data

• We derive the data for a view by a query (e.g., an OSM-QL query).
• Querying derived data presents no problem.
• Updating derived data may present an inherently unsolvable problem.
The View-Update Problem

- Q is the query that defines view V based on database D.
- D' is the updated database.
- V' is the updated view.
- U is the update specification.
- T is the translator for U, i.e., the actual update applied to D.

The problem is: there may be more than one translator T for a given update specification U.

View-Update Problem – Example

Base Schemes:

- \( r = \text{Guest} \times \text{Room} \)
- \( s = \text{Room} \times \text{View} \)

View:
- \( q = r \mid s \)

View Update: \( \text{Guest}(G1).\text{View} := \text{City} \)

Two Translations (ambiguous):
- \( \text{Guest}(G1).\text{Room} := \text{R2} \)
- \( \text{Room}(R1).\text{View} := \text{City} \)
View-Update Resolution – Example
(Designer Resolves Ambiguities)

\[\text{object module view Guest View based on Guest Room, Room View includes}\]

\[\text{@ change view(Guest, View);}\]

\textbf{read only}

\[\text{Guest has View;}\]

\textbf{hidden}

\[\text{Guest(x) has View(y) :- Guest(x) has Room(z), Room(z) has View(y);}\]

\[\text{@ change view(g: Guest, v: View) then}\]
\[\text{<< find an unoccupied room with view v >>}\]
\[\text{<< if such a room r exists, then g.Room := r >>}\]

\textbf{end:}

Chapter 11 - 37

Inheritance

- Incremental Specification
- Possible increments:
  - add data or behavior
  - subtract data or behavior
  - redefine behavior
- Conceptual-modeling (ontological) view of inheritance:
  - inherit only in ISA hierarchies
  - ISA should mean “is a”
- Consequence: this restricts possible increments to:
  - only add data or behavior (no subtraction)
  - no redefinition that changes the meaning (can only optimize specifications, e.g., areas for polygons vs. for rectangles)
Inheritance – Example

object module Guest includes

    Guest [1] has Name [1:*];
    Guest [1] has Address [1:*];

end;

object module Current Guest inherits from Guest includes

    Current Guest [0:*] incurs additional Expense [1:*] for Service [1:*];

    @ get total additional expense(Current Guest) -> (Amount);

end;