Object-Oriented Software Construction

Bertrand Meyer

Lecture 16: Object Persistence

Stephanie Balzer
Object persistence

- During execution of application: objects are created and manipulated
- What happens to objects after termination?
- Various kinds of objects
  - Transient objects:
    - Disappear with current session
  - Persistent objects:
    - Stay around from session to session
    - May be shared with other applications (e.g., databases)
Approaches to manipulate persistent objects

- Persistence mechanisms from programming languages
- Relational databases
- Object-oriented databases
Agenda for today

- Persistence from programming languages
- Advanced topics:
  - Beyond persistence closure
  - Schema evolution
- From persistence to databases
- Commercials
Agenda for today

- Persistence from programming languages
- Advanced topics:
  - Beyond persistence closure
  - Schema evolution
- From persistence to databases
- Commercials
Persistence from programming languages

- Mechanisms for storing objects in files and retrieving them
  - Simple objects:
    - e.g. integers, characters
    - conventional methods usable
  - Composite objects:
    - contain references to other objects
    - **Persistence Closure** principle:
      - Any storage and retrieval mechanism must handle the object and all its dependents.
      - otherwise: dangling references
Dependents of an object

- **Direct dependents** of an object:
  - Objects attached to its reference fields, if any

- **Dependents** of an object:
  - Object itself and dependents of its direct dependents
Example: Persistence closure

**class TREE feature**
- item: PERSON
- left: TREE
- right: TREE

**class PERSON feature**
- name: STRING
- age: INTEGER
- best_friend: PERSON
- tree: TREE object
- OT*: TREE objects
- OP*: PERSON objects

Diagram:

- tree
- OT1
  - item
    - left
    - right
- OP1
  - name: "Regula"
  - age: 24
  - best_friend
- OT2
  - item
    - left
    - right
- OP2
  - name: "Urs"
  - age: 18
  - best_friend
- OT3
  - item
    - left
    - right
- OP3
  - name: "Reto"
  - age: 29
  - best_friend
Example from EiffelBase: the class STORABLE

class interface STORABLE feature

    retrieve_by_name (file_name: STRING): ANY
    retrieved (medium: IO_MEDIUM): ANY

feature -- Element change

    basic_store (medium: IO_MEDIUM)
    general_store (medium: IO_MEDIUM)
    independent_store (medium: IO_MEDIUM)
    store_by_name (file_name: STRING)

end
STORABLE: Format variants

- **basic store (medium: IO_MEDIUM)**
  - Retrievable within current system only
  - Most compact form possible

- **general store (medium: IO_MEDIUM)**
  - Retrievable from other systems for same platform

- **independent store (medium: IO_MEDIUM)**
  - Retrievable from other systems for the same or other platform
  - Portable data representation
  - Basic information about classes in the system

- Independent of store-variant — always same retrieved
What is storable?

- Interfaced
- Deferred
- Effected
- Inheritance link

Diagram:
- IO_MEDIUM
  - FILE
    - RAW_FILE
    - PLAIN_TEXT_FILE
      - CONSOLE
class PERSISTENT_TREE inherit STORABLE
end

class APPLICATION feature
  make is
    -- Create tree.
    do
      create tree.make
    end
  end

feature -- Access
  File_name: STRING is “out.dat”

feature {NONE} -- Implementation
  tree: PERSISTENT_TREE

feature -- Storage
  store_tree is
    -- Store tree to file.
    do
      tree.store_by_name (file_name)
    end

feature -- Retrieval
  restore_tree is
    -- Retrieve tree from file.
    do
      tree ?= tree.retrieved (file_name)
      if tree /= Void then
        -- Something
      end
    end

invariant
  tree_not_void: tree /= Void
end
Agenda for today

- Persistence from programming languages
- Advanced topics:
  - Beyond persistence closure
  - Schema evolution
- From persistence to databases
- Commercials
Beyond persistence closure

**class** MOVIEDESCRIPTION **feature**

- title: STRING
- director: STRING
- category: STRING
- release_date: DATE
- movie_file: MOVIE_FILE
- cast: LINKED_LIST [ACTOR]

**end**

**class** ACTOR **feature**

- name: STRING
- tel: STRING

-- ...

**end**

O1: MOVIEDESCRIPTION object
OL*: LIST_ITEM objects
OA*: ACTOR objects

---

Chair of Software Engineering

OOSC - Summer Semester 2005
Beyond persistence closure

class MOVIE_DESCRIPTION feature
  title: STRING
director: STRING
category: STRING
release_date: DATE
movie_file: MOVIE_FILE
cast: LINKED_LIST [ACTOR]
end

class ACTOR feature
  name: STRING
tel: STRING
  -- ...
end

O1: MOVIE_DESCRIPTION object
OL*: LIST_ITEM objects
OA*: ACTOR objects

---

movie file
("BLOB")

---

O1

| title     | “Office Space” |
| director  | “Mike Judge”   |
| category  | “Comedy”       |
| release_date | “19-02-1999” |
| movie_file |               |
| cast      |               |

OL1

| name     | “Ron Livingston” |
| tel      | “01/234912”      |
| ...      |                 |

OL2

| name     | “Jennifer Aniston” |
| tel      | “034/2314134”     |
| ...      |                    |
What to do?

- “Cut out” the references of the shared structure
- At retrieval time, objects need to be consistent!
- Do not want to modify the original structure
- The references should be cut out only in the stored structure
class CUSTOMIZED_STORABLE inherit STORABLE

feature -- Storage

custom_store (medium: IO_MEDIUM) is
  -- Produce on medium an external
  -- representation of the entire object
  -- structure reachable from current
  -- object.
  do
    pre_store
    independent_store (medium)
    post_store
  end

pre_store is
  -- Execute just before object is stored.
deferred
end

post_store is
  -- Execute just after object is stored.
deferred
end

feature -- Retrieval

custom_retrieved (medium: IO_MEDIUM): ANY is
  -- Retrieved object structure, from external
  -- representation previously stored in medium
  do
    Result := retrieved (medium)
    post_retrieve
  end

post_retrieve is
  -- Execute just before retrieved objects rejoin
  -- the community of approved objects.
deferred
end

feature -- Storage (other solution)

store_ignore (field_names: LINKED_LIST [STRING]) is
  -- Store skipping the fields given by field_names.
  do
    -- Not yet implemented.
  end
end


- **pre_store** stores the reference to the object somewhere safe; sets the reference to Void

- **post_store** retrieves the object again

- **pre_store** must not perform any change of the data structure unless **post_store** corrects it immediately after

- **post_retrieve** will perform the necessary actions to correct any inconsistencies introduced by **pre_store** (often the same as **post_store**)

- **store_ignore** may simply skip the field
  - avoids the two-copy of **pre_store/post_store**
  - more efficient
Schema evolution

- Fact: Classes change
- Problem: Objects are stored of which class descriptions have changed

*Schema evolution*:
- At least one class used by the retrieving system differs from its counterpart stored by the storing system.

*Object retrieval mismatch*:
- The retrieving system retrieves a particular object whose own generating class was different in the storing system.
  $\rightarrow$ consequence for one particular object

- No fully satisfactory solution
Different approaches

- Naive, extreme approaches:
  - Forsake previously stored objects
  - Over a migration path from old format to new
    - one-time conversion of old objects
    - not applicable to a large persistent store or to one that must be available continuously

- Most general solution: **On-the-fly conversion**

- *Note*: We cover only the retrieval part. Whether to write back the converted object is a separate issue.
On-the-fly object conversion

- Three separate issues:
  - Detection:
    - Catch object mismatch
  - Notification:
    - Make retrieving system aware of object mismatch
  - Correction:
    - Bring mismatched object to a consistent state
    - Make it a correct instance of the new class version
Detection

- Detect a mismatch between two versions of an object’s generating class

- Two categories of detection policy:
  - **Nominal approach:**
    - Each class version has a version name
    - Central registration mechanism necessary
  - **Structural approach:**
    - Deduce class descriptor from actual class structure
    - Store class descriptor
    - Simple detection: compare class descriptors of retrieved object with new class descriptor
Detection: Structural Approach

- What does the class descriptor need to contain?
- Trade-off between efficiency and reliability
- Two extreme approaches:
  - C1: class name
  - C2: entire class text (e.g. abstract syntax tree)
- Reasonable approaches:
  - C3: class name, list of attributes (name and type)
  - C4: in addition to C3: class invariant
What happens when the detection mechanism has caught an object mismatch?

Class *ANY* could include a procedure:

```java
correct_mismatch is
   -- Handle object retrieval mismatch.
   local
      exception: EXCEPTIONS
   do
      create exception
      exception.raise ("[Routine failure: Object mismatch during retrieval]")
   end
```

*Correction for the previous statement:* A class named *ANY* could include a procedure named `correct_mismatch` that handles an exception for an object retrieval mismatch. The procedure would create an exception and raise it with a message indicating the routine failure due to object mismatch during retrieval.
Correction

- How do we correct an object that caused a mismatch?

Current situation:
- Retrieval mechanism has created a new object (deduced from a stored object with same generating class)
- A mismatch has been detected → new object is in temporary (maybe inconsistent) state

| Attribute was not in stored version. |
| Field is initialized to default value of attribute type |
| Attributes have not changed. |
| Stored version had a field. |
| New version has removed attribute. |
Correction

old fields

new field

(initialized to default value)

correct_mismatch is
-- Handle object retrieval mismatch
-- by correctly setting up balance.

do
  balance := deposits.total - withdrawals.total
ensure
  consistent: balance = deposits.total - withdrawals.total
end
STORABLE: Limitations

- Only the head object is known individually
  - Desirable to retain identity of other objects as well
  - Objects not selectively retrievable through contents-based or keyboard-based queries as in DBMS

- Call retrieves the entire object structure
  - Cannot use two or more such calls to retrieve various parts of a structure, unless they are disjoint

- No schema evolution

- No simultaneous access for different client applications
Agenda for today

- Persistence from programming languages
- Advanced topics:
  - Beyond persistence closure
  - Schema evolution
- From persistence to databases
- Commercials
A set of mechanisms for storing and retrieving data items is a DBMS if it supports the following items:

- Persistence
- Programmable structure
- Arbitrary size
- Access control
- Property-based querying
- Integrity constraints
- Administration
- Sharing
- Locking
- Transactions
Object-relational interoperability

- Operations: relational algebra: selection, projection, join
- Queries: standardized language (SQL)
- Usually “normalized”: every field is a simple value; it cannot be a reference

<table>
<thead>
<tr>
<th>Relation books:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>date</td>
<td>pages</td>
<td>author</td>
</tr>
<tr>
<td>“The Red and the Black”</td>
<td>1830</td>
<td>341</td>
<td>“Stendahl”</td>
</tr>
<tr>
<td>“The Carterhouse of Parma”</td>
<td>1839</td>
<td>307</td>
<td>“Stendahl”</td>
</tr>
<tr>
<td>“Madame Bovary”</td>
<td>1856</td>
<td>425</td>
<td>“Flaubert”</td>
</tr>
<tr>
<td>“Eugénie Grandet”</td>
<td>1833</td>
<td>346</td>
<td>“Balzac”</td>
</tr>
</tbody>
</table>

field name (= attribute)

tuple (= row)

field

column
Operations

- **Selection**: \( date > 1833 \)

<table>
<thead>
<tr>
<th>Title</th>
<th>date</th>
<th>pages</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>“The Carterhouse of Parma”</td>
<td>1839</td>
<td>307</td>
<td>“Stendahl”</td>
</tr>
<tr>
<td>“Madame Bovary”</td>
<td>1856</td>
<td>425</td>
<td>“Flaubert”</td>
</tr>
</tbody>
</table>

- Projection
- Join
Operations

- Selection
- **Projection**: on author

<table>
<thead>
<tr>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Stendahl”</td>
</tr>
<tr>
<td>“Flaubert”</td>
</tr>
<tr>
<td>“Balzac”</td>
</tr>
</tbody>
</table>

- Join
Operations

- Selection
- Projection
- Join: books \( \bowtie \) authors

Relation authors:

<table>
<thead>
<tr>
<th>name</th>
<th>real name</th>
<th>birth</th>
<th>death</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Stendahl”</td>
<td>“Henri Beyle”</td>
<td>1783</td>
<td>1842</td>
</tr>
<tr>
<td>“Stendahl”</td>
<td>“Henri Beyle”</td>
<td>1783</td>
<td>1842</td>
</tr>
<tr>
<td>“Flaubert”</td>
<td>“Gustave Flaubert”</td>
<td>1821</td>
<td>1880</td>
</tr>
<tr>
<td>“Balzac”</td>
<td>“Honoré de Balzac”</td>
<td>1799</td>
<td>1850</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title</th>
<th>date</th>
<th>pages</th>
<th>author</th>
<th>real name</th>
<th>birth</th>
<th>death</th>
</tr>
</thead>
<tbody>
<tr>
<td>“The Red and the Black”</td>
<td>1830</td>
<td>341</td>
<td>“Stendahl”</td>
<td>“Henri Beyle”</td>
<td>1783</td>
<td>1842</td>
</tr>
<tr>
<td>“The Carterhouse of Parma”</td>
<td>1839</td>
<td>307</td>
<td>“Stendahl”</td>
<td>“Henri Beyle”</td>
<td>1783</td>
<td>1842</td>
</tr>
<tr>
<td>“Madame Bovary”</td>
<td>1856</td>
<td>425</td>
<td>“Flaubert”</td>
<td>“Gustave Flaubert”</td>
<td>1821</td>
<td>1880</td>
</tr>
<tr>
<td>“Eugénie Grandet”</td>
<td>1833</td>
<td>346</td>
<td>“Balzac”</td>
<td>“Honoré de Balzac”</td>
<td>1799</td>
<td>1850</td>
</tr>
</tbody>
</table>
Using relational databases with O-O software

- Comparison of terms:

<table>
<thead>
<tr>
<th>Relational</th>
<th>O-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>relation</td>
<td>class</td>
</tr>
<tr>
<td>tuple</td>
<td>object</td>
</tr>
<tr>
<td>field name</td>
<td>attribute</td>
</tr>
</tbody>
</table>

- Class library to provide operations

- Usage:
  - Usage of existing data in relational databases
  - Simple object structure

- Impedance mismatch!
Object-oriented databases

- Remove impedance mismatch

- Overcome conceptual limitations of relational databases:
  - Data structure must be regular and simple
  - Small group of predefined types
  - Normal forms: no references to other “objects”

- Attempt to offer more advanced database facilities
Requirements for OODBs

- **Minimal requirements:**
  - Database functionality (listed on slide 27)
  - Encapsulation
  - Object identity
  - References

- **Additional requirements:**
  - Inheritance
  - Typing
  - Dynamic binding
  - Object versioning
  - Schema evolution
  - Long transactions
  - Locking
  - Object-oriented queries
OODBs examples

- Gemstone
- Itasca
- Matisse
- Objectivity
- ObjectStore
- Ontos
- O2
- Poet
- Matisse
- Versant
- at ETHZ: OMS Pro
End of lecture 16