



Object-Oriented Software Construction

Bertrand Meyer

Lecture 16: Object Persistence

Stephanie Balzer



Approaches to manipulate persistent objects 3

- Persistence mechanisms from programming languages
- Relational databases
- Object-oriented databases



Object persistence 2

- 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)



Agenda for today 4

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

Agenda for today

5

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

Dependents of an object

7

- 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

Persistence from programming languages

6

- 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

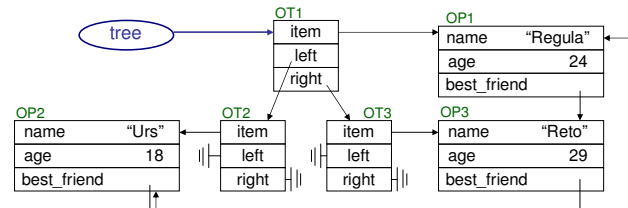
Example: Persistence closure

8

```
class TREE feature
  item: PERSON
  left: TREE
  right: TREE
end

class PERSON feature
  name: STRING
  age: INTEGER
  best_friend: PERSON
end
```

tree: TREE object
OT*: TREE objects
OP*: PERSON objects



Example from EiffelBase: the class STORABLE

class interface STORABLE feature

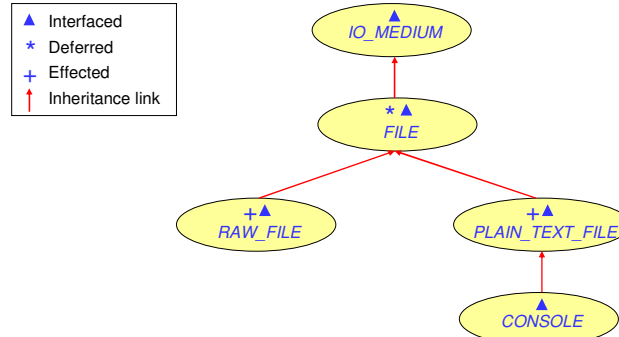
```
retrieve_by_name (file_name: STRING): ANY
retrieved (medium: IO_MEDIUM): ANY
```

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

What is storable?



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*

STORABLE: Example

```

class PERSISTENT_TREE inherit
  STORABLE
end

class APPLICATION feature

  make is
    -- Create tree.
  do
    create tree.make
  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

13

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

Beyond persistence closure

15

class MOVIE_DESCRIPTION feature

```

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

end

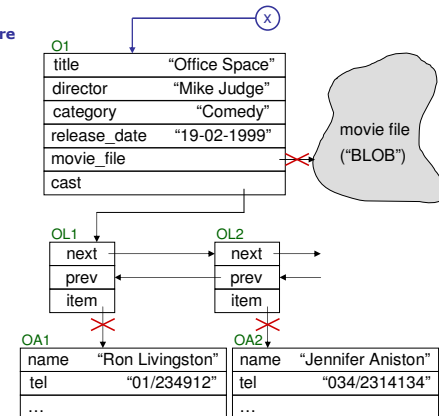
class ACTOR feature

```

name: STRING
tel: STRING
-- ...
    
```

end

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



Beyond persistence closure

14

class MOVIE_DESCRIPTION feature

```

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

end

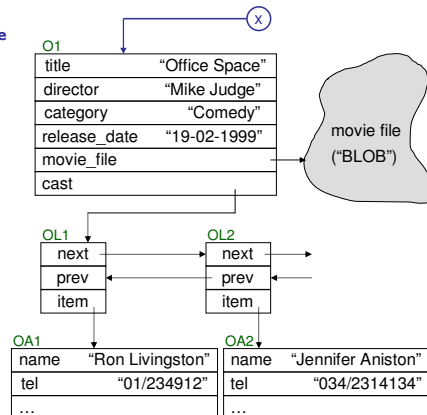
class ACTOR feature

```

name: STRING
tel: STRING
-- ...
    
```

end

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



What to do?

16

- "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



A customized class STORABLE

17

```

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

```



Schema evolution

19

- 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.
 - consequence for one particular object
- No fully satisfactory solution



A customized class STORABLE (cont'd)

18

- *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



Different approaches

20

- 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

21

- 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: Structural Approach

23

- 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



Detection

22

- 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



Notification

24

- What happens when the detection mechanism has caught an object mismatch?
- Class *ANY* could include a procedure:

```

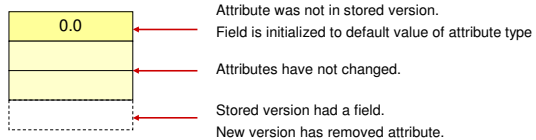
correct_mismatch is
  -- Handle object retrieval mismatch.
  local
    exception: EXCEPTIONS
  do
    create exception
    exception.raise ("[
      Routine failure: Object
      mismatch during retrieval
    ]")
  end

```

Correction

25

- 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



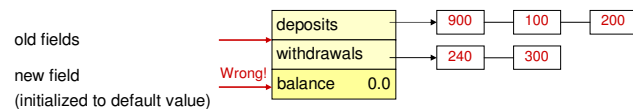
STORABLE: Limitations

27

- 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

Correction

26



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

Agenda for today

28

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



From persistence to databases

29

- 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



Operations

31

- Selection: *date > 1833*

Title	date	pages	author
"The Carterhouse of Parma"	1839	307	"Stendahl"
"Madame Bovary"	1856	425	"Flaubert"

- Projection
- Join



Object-relational interoperability

30

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

Relation books:

Title	date	pages	author
"The Red and the Black"	1830	341	"Stendahl"
"The Carterhouse of Parma"	1839	307	"Stendahl"
"Madame Bovary"	1856	425	"Flaubert"
"Eugénie Grandet"	1833	346	"Balzac"

Annotations: "field name (= attribute)" points to the header row; "tuple (= row)" points to the first data row; "field" points to the 'date' column; "column" points to the 'author' column.



Operations

32

- Selection
- Projection: *on author*

author
"Stendahl"
"Flaubert"
"Balzac"

- Join

Operations

33

- Selection
- Projection
- **Join:** *books* ⋈ *authors*

Relation authors:

name	real name	birth	death
"Stendahl"	"Henri Beyle"	1783	1842
"Stendahl"	"Henri Beyle"	1783	1842
"Flaubert"	"Gustave Flaubert"	1821	1880
"Balzac"	"Honoré de Balzac"	1799	1850

Title	date	pages	author	real name	birth	death
"The Red and the Black"	1830	341	"Stendahl"	"Henri Beyle"	1783	1842
"The Carterhouse of Parma"	1839	307	"Stendahl"	"Henri Beyle"	1783	1842
"Madame Bovary"	1856	425	"Flaubert"	"Gustave Flaubert"	1821	1880
"Eugénie Grandet"	1833	346	"Balzac"	"Honoré de Balzac"	1799	1850

Object-oriented databases

35

- 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

Using relational databases with O-O software

34

- Comparison of terms:

Relational	O-O
relation	class
tuple	object
field name	attribute

- Class library to provide operations
- Usage:
 - Usage of existing data in relational databases
 - Simple object structure
- **Impedance mismatch!**

Requirements for OODBs

36

- **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

37

- Gemstone
- Itasca
- Matisse
- Objectivity
- ObjectStore
- Ontos
- O2
- Poet
- Matisse
- Versant
- at ETHZ: OMS Pro



End of lecture 16