6.005
elements of software construction
coding the photo organizer (lecture 17)
Daniel Jackson
April 16, 2007
topics for today

reviewing the photo organizer object model
  • another multiplicity example
  • resolving a subtle issue

implementing the object model
  • basic patterns
  • navigation direction
  • derived components
  • maintaining invariants
photo organizer OM
the photo organizer app

can select images to add/remove

define collections
an object model

we've added naming

› always an important and subtle issue
› is the multiplicity constraint desirable? necessary?
hold photos of subcollections?

initial state

after delete (P2,C2):
P2 not shown for C1 or C2

after delete (P1,C2):
P1 still shows in C1?
resolving issues

do parents hold children’s photos?

• can express the question in logic:
  \[ \text{all } c: \text{Collection } | \ c.\text{subs}.\text{photos} \text{ in } c.\text{photos} \quad ? \]

• consider removing a photo: does it disappear from parent collection too?

• decision: use two relations instead
  \[ c.\text{inserted}: \text{the photos explicitly inserted into collection } c \]
  \[ c.\text{photos}: \text{the photos in collection } c \text{ implicitly and explicitly} \]

• invariant relates these: \[ c.\text{photos} = c.\text{inserted} + c.\text{subs}.\text{photos} \]

another issue: “all photos”

• how’s the set of all photos in catalog represented?

• decision: introduce root collection distinct from user-defined collections
additional constraints

- all collections reachable from root (implies acyclic)
  \[ \text{Collection in Root.} *\text{subs} \]

- implicit photos are inserted photos plus photos in subcollections
  \[ \text{all } c: \text{Collection} \mid c.\text{photos} = c.\text{inserted} + c.\text{subs}.\text{photos} \]

- names unique within parent
  \[ \text{all } c: \text{Collection} \mid \text{no } c1, c2: c.\text{subs} \mid c1 \neq c2 \text{ and } c1.\text{name} = c2.\text{name} \]
implementing the OM
implementing sets

**top-level sets become classes**

- set as class: `class Collection {...}`

**subset patterns**

- subset as boolean field: `class Photo {boolean selected;}`
- subset as singleton set: `class Catalog {Set<Photo> selected;}`
  `class Catalog {Collection root;}`

**static subset patterns**

- classification of object does **not** change over time
- subset as subclass: `class Root extends Collection {...}`
implementing relations

basic patterns (function)

- relation as field: `class` Collection `{Name name;}
- relation as map: `class` Catalog `{Map<Collection, Name> name;}

basic patterns (one-to-many)

- relation as field: `class` Collection `{Set<Collection> subs;}
- relation as map: `class` Catalog `{Map<Collection, Set<Collection>> subs;}

how to choose?

- efficiency: relation as field uses marginally less time and space
- immutability: relation as map is preferable if Collection otherwise immutable
- encapsulation: choose so that OM invariant can be a rep invariant
relation direction

navigation direction

• direction of relation in object model is semantic
• navigation direction depends on operations
• for relation $R$: can implement $R$, transpose of $R$, or both

implementation must support navigation

• consider inserted: Collection $\rightarrow$ Photo and operation add $(p, c)$
• relation as field: **class** Collection {Set<Photo> insertedPhotos;}
  or **class** Photo {Set<Collection> insertedInto;}
• relation as map: **class** Catalog {Map<Collection, Set<Photo>> insertedPhotos;}
  or **class** Catalog {Map<Photo, Set<Collection>> insertedInto;}
• for basic add operation, implementing as $C \rightarrow P$ is fine
• but if add operation removes photo from other collections, will want both directions
derived components

**derived component**

- a set or relation that can be derived from others
- OM invariant has the form \( x = \ldots \)

**in this case**

- can choose not to implement at all!
- instead, construct value when needed

**examples**

- \( \text{UserDefined} = \text{Collection} - \text{Root} \)
  so to determine if \( c \) in \( \text{UserDefined} \), can just check \( c == \text{Root} \)
- all \( c : \text{Collection} | c\.photos = c\.inserted + c\.subs\.photos \)
  so can compute \text{photos} set for given \( c \) by traversing subcollections
maintaining OM invariants

OM invariants
• called “integrity constraints” for databases
• become rep invariants or invariants across classes

to maintain
• reject inputs that might break invariant (eg, duplicate name for collection)
• or compensate for bad input (eg, modify name to make it unique)

to check
• insert repCheck methods and assertions for cross-class invariants
decisions made

in implementing the photo organizer, we chose

• subset as boolean field for Selected (in Thumbnail class)
• relation as field for name (in Collection class), since the relation is immutable
• relation as map for subs and inserted (in Catalog class)
• to implement subs in the direction of child to parent
  (so getChildren method has to iterate-and-check to find children)
• to compute UserDefined and photos on the fly

see code in handout
summary

mapping OM to code

- is generally straightforward
- but some subtle choices

choices of implementation

- basic patterns combine fields, sets, hash maps
- navigation and encapsulation influence choices
- need to consider operations
lecture exercises

photo organizer puzzle
\· does clicking on a thumbnail and selecting delete make it disappear?
\· not always: find a scenario in which it doesn't
\· explain (using an OM) why this reflects a visibility problem in the GUI
\· how should this be fixed?

a redundant class?
\· since collections are uniquely named, is Collection needed in the code?

other organizer invariants
\· are there other invariants you’d impose on the organizer?
\· eg, are all names legal?