object models to java

Daniel Jackson
April 14, 2010
topics for today

how to implement an object model

• key idea: transform to allocate state
• basic patterns
• navigation direction
• derived components
• maintaining invariants
example: colour palettes

modelling the state of an application

› how colours are organized

essential idea

› elements are coloured
› can assign colour from palette
› gives consistent appearance
palette object models

three subtly different approaches

‣ think what happens when palette is modified
‣ hard vs. soft links: as in Unix

“Every problem in computer science can be solved by introducing another level of indirection”
-- David Wheeler
completing the organizer
issues to resolve

can albums hold photos and subalbums?
  • decision: yes, so not Composite pattern

how are “all photos” in catalog represented?
  • decision: introduce non-visible root album

unique album names?
  • decision: globally unique (not like file system paths)

do parents hold children’s photos?
  • in logic: all a: Album | c.subs.photos in c.photos ?
  • decision: use two relations instead
    a.inserted: the photos explicitly inserted into album c
    a.photos: the photos in album a implicitly and explicitly
    invariant relates these: a.photos = a.inserted + a.subs.photos

renamed collections to albums

class
discussion
about usability,
hidden state: like style
inheritance
final object model

additional constraints

- all albums reachable from root (implies acyclic)

  Album in Root. *subs

- implicit photos are inserted photos plus photos in subalbums

  all a: Album | a.photos = a.inserted + a.subs.photos
implementing the OM
basic strategy

object model can be implemented in many ways

› key issue: where state resides

eg, where does relation from A to B go?

› inside A object, or inside B object
› or inside a new singleton C object, as Map<A,B>
› or nowhere: compute on-the-fly

considerations

› ease & efficiency of navigation
› multiplicity (might call for collections)
› minimizing memory usage
› exploiting immutability
› minimizing dependences
implementing sets

top-level sets become classes

- set as class: `class Album {...}, class Photo {...}
- set as built-in class: Name as String

subset patterns

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

static subset patterns

- classification of object does not change over time
- subset as subclass: `class Root extends Album {...}
example: Selected

```
Photo

Selected

Photo

isSelected

Boolean

Catalog

selected

Set

elts

Photo

OR
```
example: Root
implementing relations

basic patterns (function)

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

basic patterns (one-to-many)

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

how to choose?

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

OR
example: subs

UserDefined → Album

! subs

OR

Catalog → subMap → Map
? ! entries

Map → entries → Entry
?

Set → elts

Catalog → parentMap → Map
? ! entries

Entry → key !
val

Set → elts

Album → key !
val

© Daniel Jackson 2010
relation direction

navigation direction

\* direction of relation in object model is \textit{semantic}
\* navigation direction depends on \textit{operations}
\* for relation $R$: can implement $R$, transpose of $R$, or both

implementation must support navigation

\* consider inserted: \texttt{Album $\rightarrow$ Photo} and operation \texttt{add (a, p)}

\* relation as field: \texttt{class Album} $\{\text{Set<Photo> insertedPhotos;}\}$
  \texttt{or class Photo} $\{\text{Set<Album> insertedInto;}\}$

\* relation as map: \texttt{class Catalog} $\{\text{Map<Album, Set<Photo>> insertedPhotos;}\}$
  \texttt{or class Catalog} $\{\text{Map<Photo, Set<Album>> insertedInto;}\}$

\* for basic \texttt{add} operation, implementing as \texttt{Album $\rightarrow$ Photo} is fine

\* but if \texttt{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
• UserDefined = Album - Root
  so to determine if \( a \) in UserDefined, can just check \( a == \) Root
• all \( a: \) Album \( | \) a.photos = a.inserted + a.subs.photos
  so can compute photos set for given \( a \) by traversing subalbums
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 Album 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 getUserChildren method has to iterate-and-check to find children)
- to compute UserDefined and photos on the fly
architecture of GUI may influence decisions

- regard selection and images as part of view, not model
- and want to avoid back-dependences of model on view
public class Catalog {
    private static final String ROOTNAME = "all photos";
    // root album, cannot be deleted
    private final Album root;
    // map from child album to parent
    private final Map<Album, Album> parent;
    // map from albums to photos that were explicitly inserted into them
    private final Map<Album, Set<Photo>> inserted;
}

public final class Album {
    private String name;
}

public class Photo {
    private final File file;
}
public class PreviewPane extends JScrollPane {
    private JPanel content;
    private List<Thumbnail> thumbnails;
}

public class Thumbnail extends JComponent {
    public static final int THUMBNAIL_SIZE = 150;

    private Photo photo;

    // the loaded, displayable thumbnail image
    private BufferedImage bufferedImage;

    private int width;
    private int height;

    private boolean isSelected;
private void checkRep() {
    /*
     * 1) All fields are non-null
     * 2) The root has no parent; all other albums have one parent
     *    all a: albums | parent.get(a) == null iff a == root
     * 3) Each album has a unique name
     *    all a1, a2: albums | a1.equals(a2) or !a1.getName().equals(a2.getName())
     * 4) Map of inserted photos has all albums as keys
     *    inserted.keySet() = parent.keySet() + root
     *    where albums is the set of Album objects that are keys or values in the parent map
     */

    // checking rep (1)
    assert root != null: "root cannot be null!";
    assert parent != null: "parent cannot be null!";
    assert inserted != null: "inserted cannot be null!";

    // checking rep (2,4)
    assert parent.get(root) == null: "Root cannot have a parent!";
    Set<Album> a1 = new HashSet<Album>(inserted.keySet());
    Set<Album> a2 = new HashSet<Album>(parent.keySet());
    a2.add(root);
    assert a1.equals(a2) : "Inconsistent album sets!";

    // checking rep (3)
    Set<Album> x = new HashSet<Album>(inserted.keySet());
    for (Album a: x) {
        for (Album d: x) {
            assert (a == d || !a.getName().equals(d.getName())): "Albums exist with duplicate names";
        }
    }
}
summary: principles

keep abstract model abstract
• relations are conceptual; no containment notion

implementation is OM transformation
• from abstract to code object model
• key decision: where state should reside

consider all criteria
• use built-in collections when possible
• consider navigation, encapsulation, immutability