6.005 Elements of Software Construction

Fall 2008

coding the photo organizer

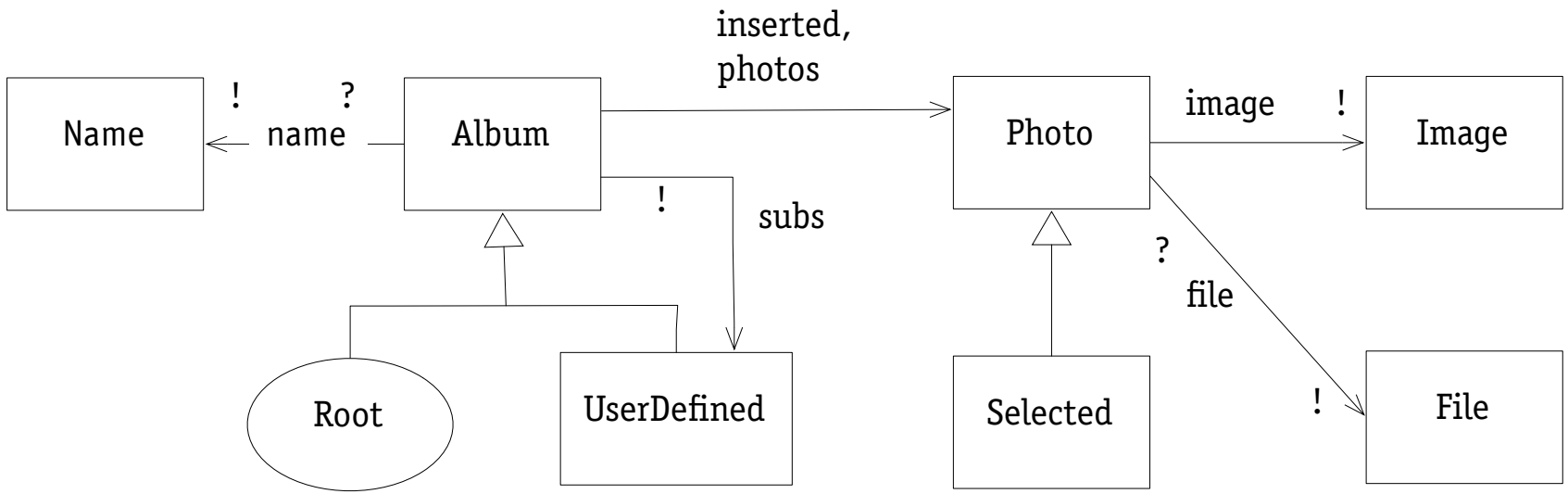
Daniel Jackson

topics for today

how to implement an object model

- key idea: transform to allocate state
- basic patterns
- navigation direction
- derived components
- maintaining invariants

starting point: 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

changes

- globally unique names; added File; renamed Collection to Album

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

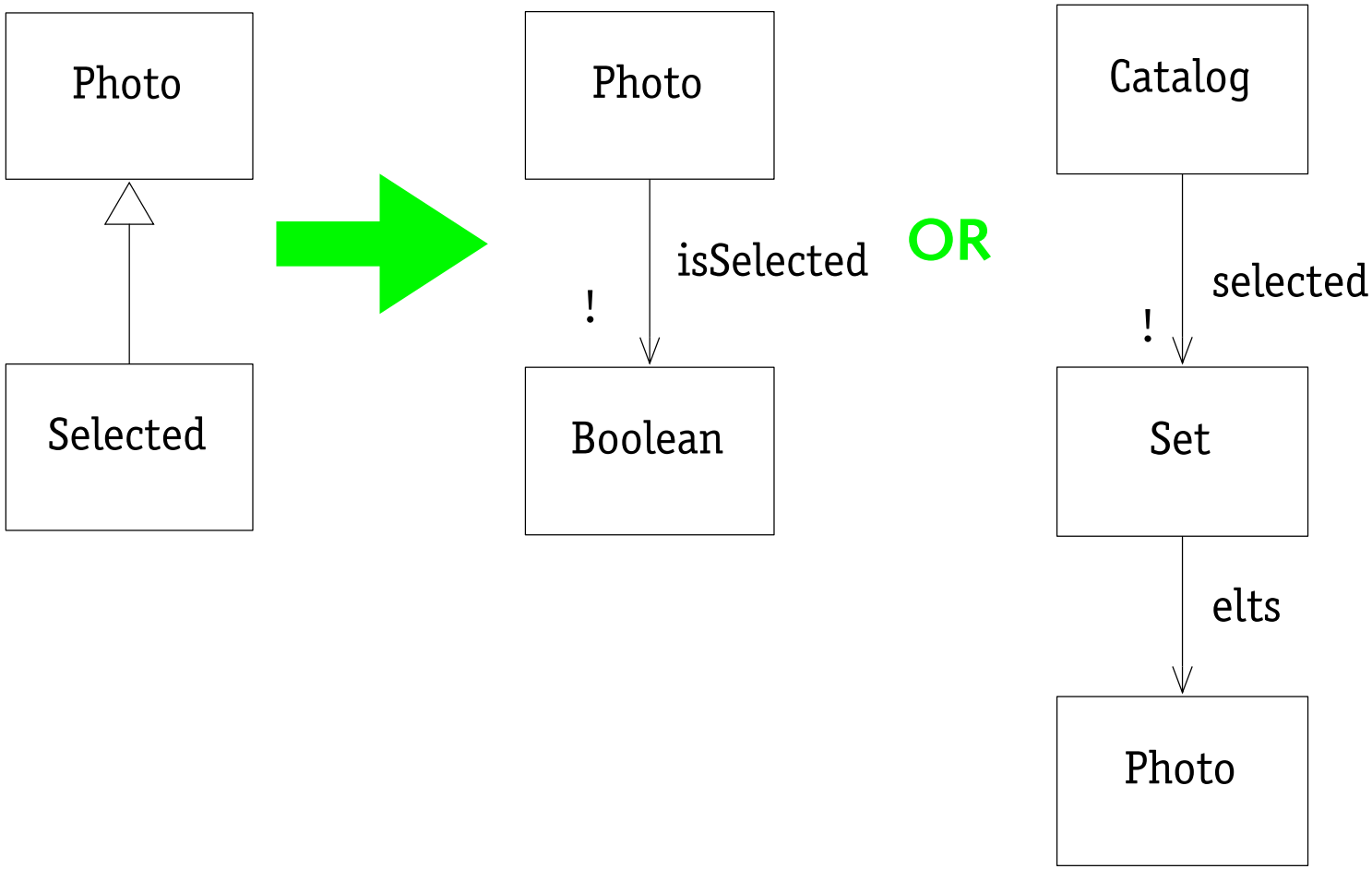
subset patterns

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

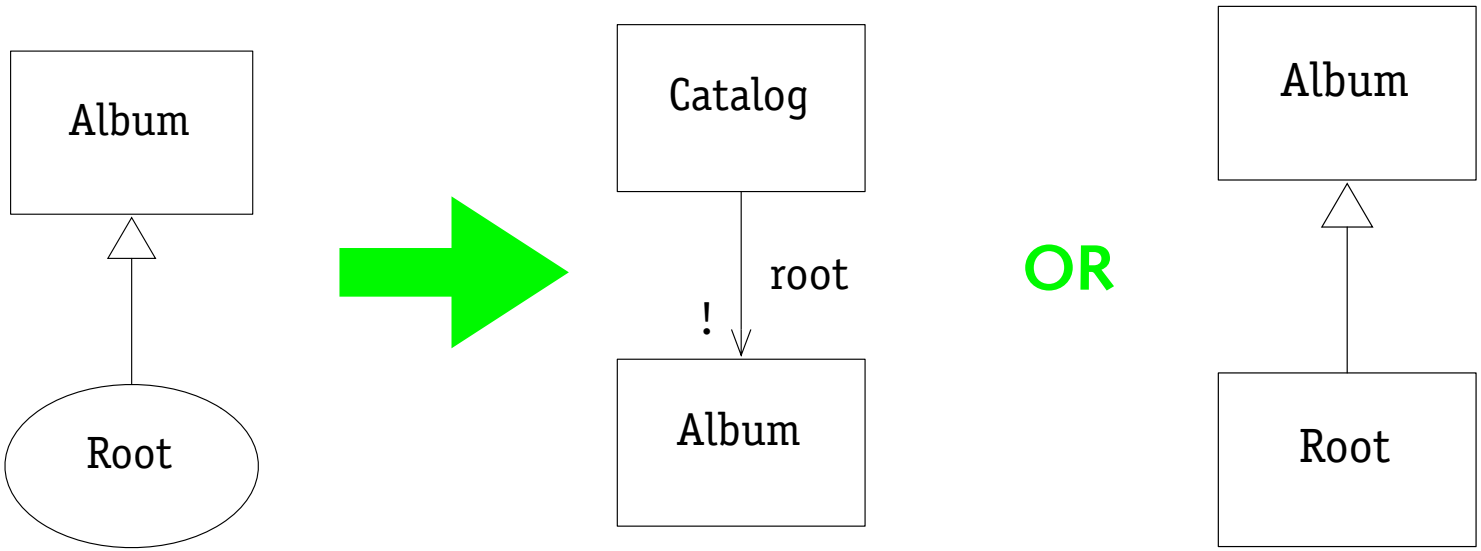
static subset patterns

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

example: Selected



example: Root



implementing relations

basic patterns (function)

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

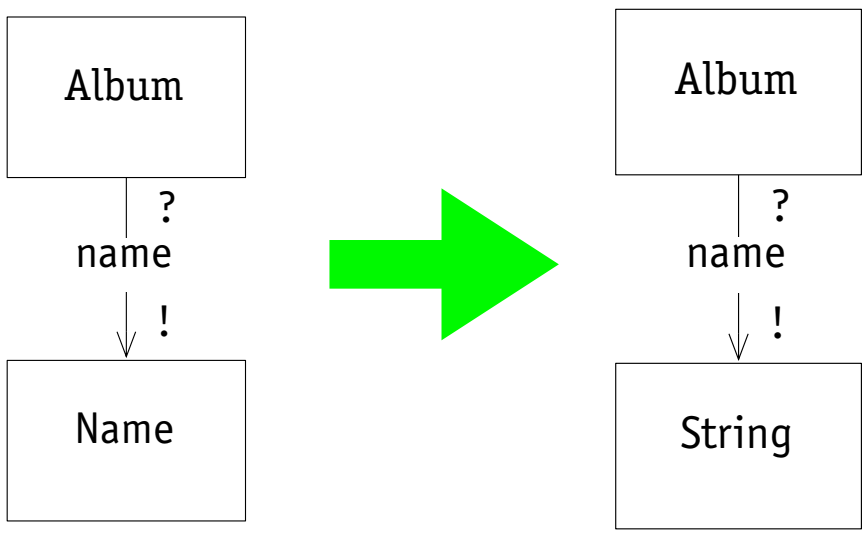
basic patterns (one-to-many)

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

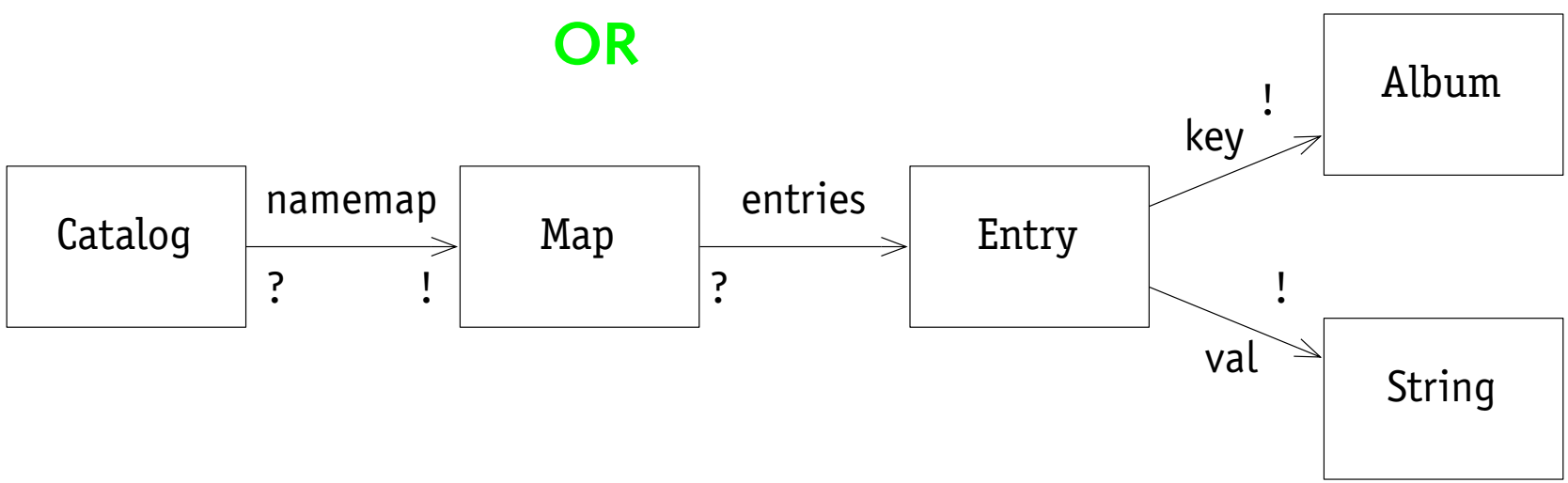
how to choose?

- 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

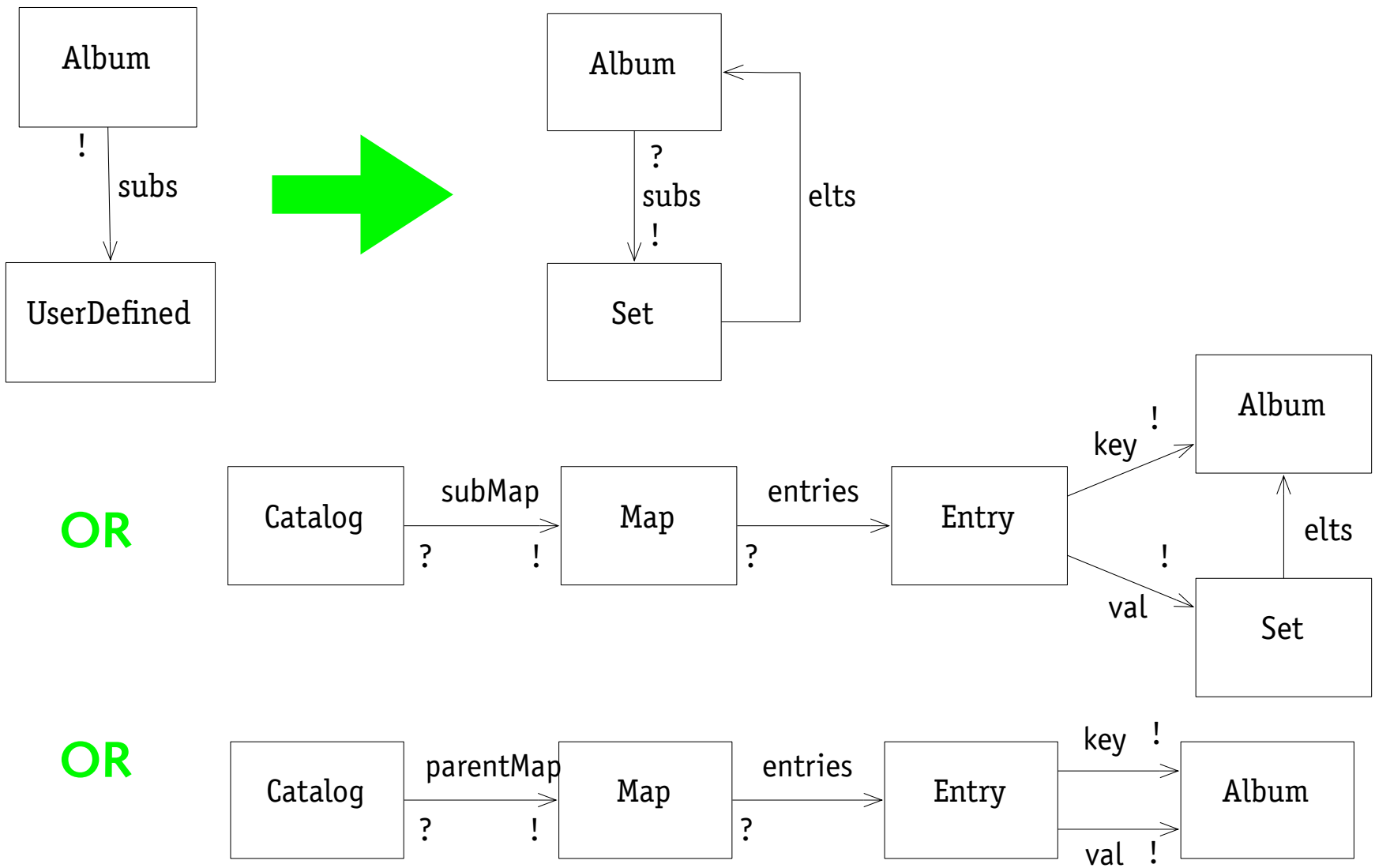
example: name



OR



example: subs



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: Album -> Photo` and operation `add (a, p)`
- relation as field: **class** Album {Set<Album> insertedPhotos;}
  or **class** Photo {Set<Collection> insertedInto;}
- relation as map: **class** Catalog {Map<Album, Set<Photo>> insertedPhotos;}
  or **class** Catalog {Map<Photo, Set<Album>> insertedInto;}
- for basic `add` operation, implementing as `Album -> Photo` 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 = ...

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 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

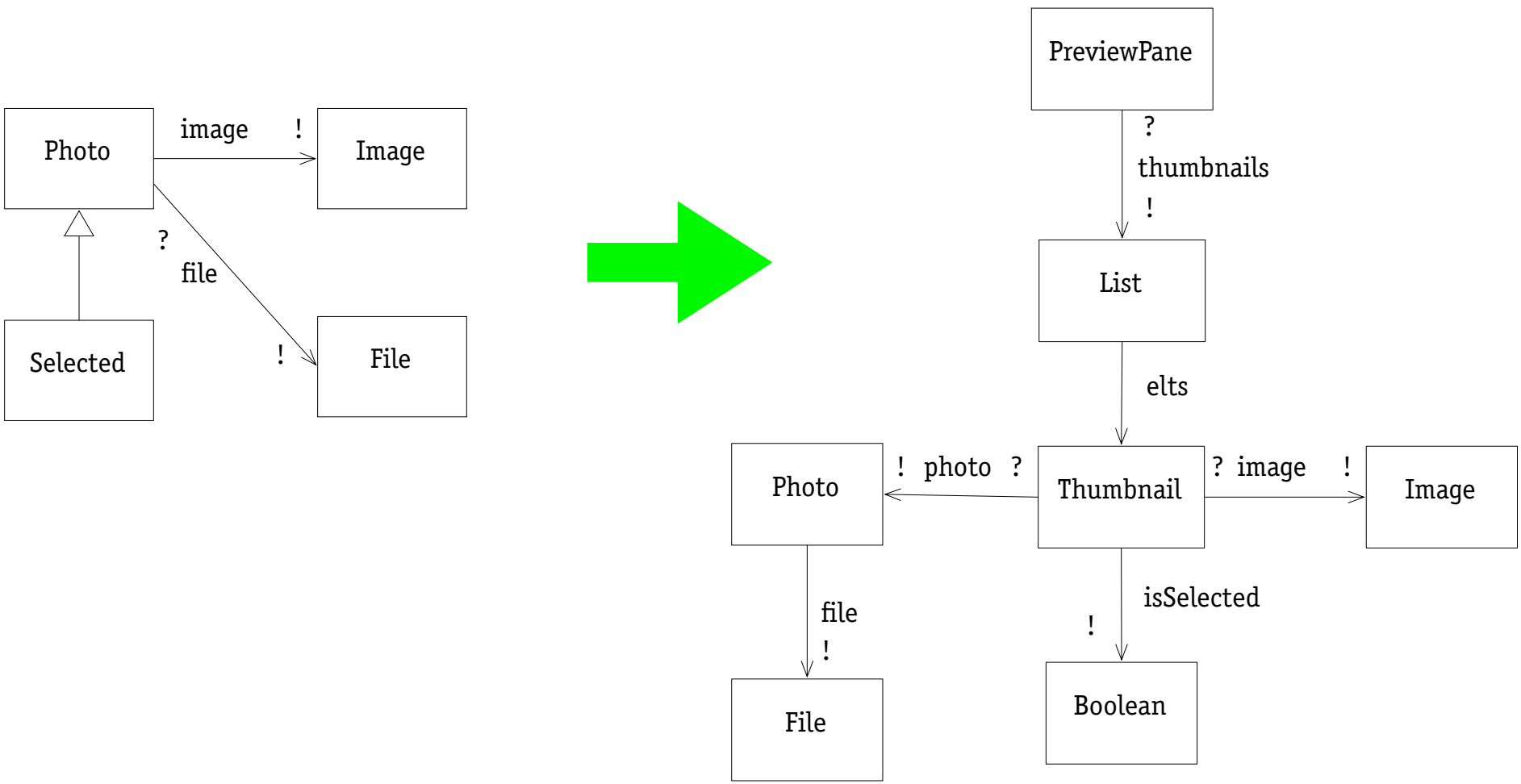
in implementing the photo organizer, we chose

- subset as boolean field for Selected (in Thumbnail class)
- relation as field for name (in 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 getChildren method has to iterate-and-check to find children)
- to compute UserDefined and photos on the fly

thumbnails

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



final code: catalog, album, etc

```
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;
}
```

final code: selected, etc

```
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;
```

catalog rep invariant

```
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