CS 151 - Prof Kim

Assignment #3 Documents

Design patterns

MVC Pattern

Controller , “Add Photo” action listener

```
// File: src/photoalbum/controller/PhotoAlbumController.java

// Method: doAddPhoto()  (invoked by the Add button's action listener)


view.addBtn.addActionListener(e -> doAddPhoto());


private void doAddPhoto() {

    JFileChooser chooser = new JFileChooser();

    chooser.setDialogTitle("Choose an image file");

    int result = chooser.showOpenDialog(view);

    if (result != JFileChooser.APPROVE_OPTION) return;


    File f = chooser.getSelectedFile();

    // (validation with ImageIO omitted for brevity)


    String defaultName = f.getName();

    String name = (String) JOptionPane.showInputDialog(
        view, "Enter a display name:", "Photo Name",
        JOptionPane.PLAIN_MESSAGE, null, null, defaultName);
    if (name == null || name.isBlank()) name = defaultName;

    // ↓↓↓ Controller calls the MODEL mutator
    Photo p = new Photo(name.trim(), f.getAbsolutePath(), new Date());
    model.addPhoto(p);

    // Put selection on the newly added photo (uses model accessors)
    List<Photo> ordered = model.getSortedPhotos();
    int idx = ordered.indexOf(p);
    if (idx < 0) idx = 0;
    model.setCurrentIndex(idx);
}
```

Model — data structure + mutator that updates and notifies the view

```
// File: src/photoalbum/model/PhotoAlbumModel.java

// --- Data structure that stores photos ---
private final List<Photo> photos = new ArrayList<>();

// --- Change listeners (View/Controller subscribe here) ---
```

```
private final List<ChangeListener> listeners = new ArrayList<>();
public void addChangeListener(ChangeListener l) { listeners.add(l); }
private void fireChange() {
    ChangeEvent evt = new ChangeEvent(this);
    for (ChangeListener l : List.copyOf(listeners)) l.stateChanged(evt);
}

// --- Mutator: add a photo, update state, then notify listeners (View) ---
public void addPhoto(Photo p) {
    if (p == null) return;
    photos.add(p);
    if (currentIndex < 0) currentIndex = 0; // select first item if album was empty
    fireChange();                            // ← notifies View to refresh
}
```

View — shows current photo and triggers repaint

> Note: In this MVC, the **controller** reads the model’s accessors and passes a `Photo` to the view. The view renders it and explicitly calls `revalidate()/repaint()` so changes appear immediately.

```
// File: src/photoalbum/view/PhotoAlbumView.java


public void showCurrentPhoto(Photo p) {
    if (p == null) {
        photoLabel.setText("No photo");
        photoLabel.setIcon(null);
    } else {
        ImageIcon icon = p.getDisplayIcon(1000, 520); // model-provided data already
resolved by controller
        if (icon == null) {
            photoLabel.setText("No photo (unsupported or unreadable file)");
            photoLabel.setIcon(null);
        } else {
            photoLabel.setText(p.getName());
            photoLabel.setIcon(icon);
            photoLabel.setHorizontalTextPosition(SwingConstants.CENTER);
            photoLabel.setVerticalTextPosition(SwingConstants.BOTTOM);
        }
    }
    // ↓↓↓ Forces the component to redraw with the new image/text
    photoLabel.revalidate();
    photoLabel.repaint();
}
```

For completeness, here is the controller method that **uses the model’s accessors** and then calls the view (typical MVC flow):

```
// File: src/photoalbum/controller/PhotoAlbumController.java


private void refreshView() {
    List<Photo> ordered = model.getSortedPhotos(); // model accessor
    List<String> names  = new ArrayList<>(ordered.size());
    for (Photo p : ordered) names.add(p.getName());

    syncingView = true;
    try {
        view.setPhotoNames(names);
        int idx = model.getCurrentIndex();         // model accessor
        if (!ordered.isEmpty()) {
            if (idx < 0 || idx >= ordered.size()) idx = 0;
            view.setSelectedIndex(idx);
            view.showCurrentPhoto(ordered.get(idx)); // triggers repaint in the view
        } else {
            view.setSelectedIndex(-1);
            view.showCurrentPhoto(null);
        }
        view.setStatus("Sort: " + model.getStrategy().name()
                  + " | Photos: " + ordered.size());
    } finally {
        syncingView = false;
    }
}
```

Strategy Pattern

Concrete strategies

```
// File: src/photoalbum/strategy/SortByName.java
public class SortByName implements SortingStrategy {
    @Override public List<Photo> sort(List<Photo> photos) {
        List<Photo> copy = new ArrayList<>(photos);
        copy.sort(Comparator.comparing(Photo::getName,
String.CASE_INSENSITIVE_ORDER));
        return copy;
    }
    @Override public String name() { return "Name"; }
}

// File: src/photoalbum/strategy/SortByDate.java
public class SortByDate implements SortingStrategy {
    @Override public List<Photo> sort(List<Photo> photos) {
        List<Photo> copy = new ArrayList<>(photos);
        copy.sort(Comparator.comparing(Photo::getDateAdded));
        return copy;
    }
    @Override public String name() { return "Date"; }
}


// File: src/photoalbum/strategy/SortBySize.java
public class SortBySize implements SortingStrategy {
    @Override public List<Photo> sort(List<Photo> photos) {
        List<Photo> copy = new ArrayList<>(photos);
        copy.sort((a, b) -> Long.compare(a.getFileSize(), b.getFileSize()));
        return copy;
    }
    @Override public String name() { return "Size"; }
}
```

Context: code that accepts/uses a strategy

```
// File: src/photoalbum/model/PhotoAlbumModel.java


// current strategy (context state)
private SortingStrategy strategy = new SortByDate(); // default


// plug in a new strategy dynamically
public void setStrategy(SortingStrategy s) {
    if (s == null) return;

    this.strategy = s;

    fireChange(); // view will refresh in the new order

}


// use the active strategy to produce the ordered view of data

public List<Photo> getSortedPhotos() {

    return strategy.sort(photos);

}
```

(Buttons in the controller call `model.setStrategy(new SortByName())`, etc.)

Iterator Pattern

Iterator class

```
// File: src/photoalbum/model/AlbumIteratorImpl.java

public class AlbumIteratorImpl implements AlbumIterator {

    private final List<Photo> ordered;

    private int index;


    public AlbumIteratorImpl(List<Photo> ordered, int startIndex) {

        this.ordered = ordered;
        this.index = Math.max(0, Math.min(startIndex, Math.max(0, ordered.size() - 1)));
    }

    @Override public boolean hasNext()     { return !ordered.isEmpty() && index <
ordered.size() - 1; }
    @Override public boolean hasPrevious() { return !ordered.isEmpty() && index > 0; }

    @Override public Photo current() {
        if (ordered.isEmpty()) return null;
        return ordered.get(index);
    }

    @Override public Photo next() {
        if (hasNext()) index++;
        return current();
    }

    @Override public Photo previous() {
        if (hasPrevious()) index--;
        return current();
    }

    /** Exposes the iterator’s current index so the controller can sync it back. */
    public int getIndex() { return index; }
}
```

And the model exposes an iterator over the **current sorted** album:

```
// File: src/photoalbum/model/PhotoAlbumModel.java
public AlbumIterator iterator() {
    return new AlbumIteratorImpl(getSortedPhotos(), Math.max(0, currentIndex));
}
```