AP Assignment **I** / Ex 1
EightPuzzle with Java Beans

Francesco Lorenzoni

July 2024

EightPuzzle with Java Beans

1.1 About folders content

The assignment development was tracked using git. The repository is available at github.com/frenzis01/AP-assignments. The folder of the first exercise is `assignment1/ex1`.

There are many subdirs which mirror the structure of the NetBeans project, but only the `.java` class files are actually present, for ease of reading. There is also the `.jar` file (renamed to `.raj`) of the app which may be tested as it is. There is also a `NetBeans.zip` archive resulting from the export of the NetBeans project, which contains the source code and the NetBeans project files; it may be imported in NetBeans to navigate and test project from the IDE.

1.2 Main Design Choices

Components are exactly the ones required by the assignment: `EightTile`, `EightBoard`, `EightController` and `Flip`, with the only addition of a simple wrapper class `IntWrapper` under the `utils` package.

1.2.1 `position` and `label` attributes

It is important to keep in mind that `EightTile` components are positioned as in Fig. 1.1. Every `EightTile` is instantiated with the expected corresponding `position`, i.e. `eightTile6` → `eightTile6.position = 6`.

`EightTiles` *never* "move" in the board and neither their `final position` attribute ever changes, only the `label` attribute is changed throughout the game, mimicking the sliding of the tiles.

`label` is private, there is no getter and no setter. It is changed only when invoking `labelRequest()` (which fires a vetoable change) and upon receival of `"swapOK"` and `"restart"` events.

Figure 1.1: `EightTile` components organized in the `EightBoard` grid

1.2.2 Observer pattern

The interaction between components is based on events and listeners, following the Observer pattern.
`EightTile.label` is a *Constrained Property* which fires a `PropertyChangeEvent` by invoking `fireVetoableChange(...)` on the `EightController` instance, which is responsible for ensuring that only legal changes are allowed. The three scenarios for a `label` change are:

1. *Tile got clicked*: clicked `EightTile` may move to the *hole* (i.e. `EightTile` whose `label == 9`), if adjacent to it.
2. *Flip button got clicked*: tiles in position 1 and 2 are swapped, if the *hole* is in the middle (`position == 5`)
3. *Restart button got clicked*: tiles receive a permutation and infer their new label (`eightTiles[i].label := permutation[i]`); in this case, such change is not vetoed.

1.2.3 Events

1.2.3.1 Clicking a Tile

When an `EightTile` gets clicked it attempts to swap its position to the hole. To do so, sets the property `requestedLabel = 9` (i.e. hole), and executes `EightTile.labelRequest(9)` which fires a `PropertyChangeEvent` with `propertyName = "label"`, `oldValue = position` and `newValue = 9`.

Bending the rules

Considering the signature Lst. 1.1 below, we might say that the parameters are not exactly used as their name suggests, and we'd probably be right ☺.
In fact `oldValue` is used to pass the *position* of the tile, while the `newValue` is used to pass the new *label* of the tile.
I chose to pass the position in the event to avoid making the Controller check the position of the requesting tile using methods (e.g. `((EightTile)pce.getSource()).getPosition()`) as the assignment requires.

It is needed to wrap position because in case `oldValue == newValue` the event is not fired, and it is not unlikely that `position == newLabel`.

```
// Signature of fireVetoableChange
fireVetoableChange(String propertyName, Object oldValue, Object newValue)
// How it is used in EightTile
this.mVcs.fireVetoableChange(propertyName, new IntWrapper(this.position), newLabel);
```

Listing 1.1: How `fireVetoableChange` is used in `EightTile`

`EightController` which a `vetoableChangeListener`, checks if the requesting tile is adjacent to the hole, and if so, it allows the change, otherwise throws an exception. In case an `EightTile` successfully moves, fires an `ActionEvent("swapOK")`[1], which is listened by all other tiles, so that the tile —typically the *hole*, unless it's a *Flip*— which has been swapped can update its label. The requested label is obtained by listener by invoking on the event source `getClientProperty("requestedLabel")`.

7 tiles out of 8 listen to the event uselessly, the clicked tile could send it only to the hole, but it would require for it to have a reference to the tile currently holding the hole; for simplicity and readability, I chose to make all tiles listen to the event.

1.2.3.2 Restarting a Game

The restart button generates an array representing a permutation of the labels —with the i^{th} element being the label for the `EightTile` in i^{th} position— and puts it in a property `"permutation"`, and emits an event `"restart"`, which is listened by both the tiles to update their label, and by the Flip button to update the position of the hole.

The board is initialized by mimicking a click on the restart button.

1.3 Flip Button

The `Flip` button is not aware of the labels of the tiles, it only knows the position of the hole, and the labels of the tiles in position 1 and 2; the button listens for `"restart"` and `"swapOK"` events, to update its state.

When clicked, it emits an event `"flip"`, which is listened by the `EightController`, which checks if the hole is in the middle, if so the button updates its *internal state* (`label1` and `label2`).
To implement the actual Flip on the tiles, the label update is delegated to the tiles theirselves by simply setting the `"requestedLabel"` property of the tile in position 1 to the label of the tile in position 2, and then mimicking a click on the tile in position 1.

```
// Code taken from the flip1 actionListener in EightBoard
if (flip1.flipTiles()){
   // label1 and label2 inside flip1 have been successfully swapped!
   eightTile1.putClientProperty("requestedLabel", flip1.getLabel1());
   eightTile1.doClick();
}
```

The change will be again vetoed by `EightController` which will notice that the requested label is not the hole, and will allow the change.

The controller checks that the hole is in the middle even if at this point, the hole should be there; this double check is not necessary

[1] Almost analogous behaviour may be obtained by using `PropertyChangeEvent`