Implementing the UNDO functionality for the DeViSoRGrid2 application

Dominik Göddeke

May 10th, 2002

This document describes the basic idea, the class design and some implementation details of the UNDO and REDO functionalities for the DeViSoRGrid2 application. We also work on a full-scale example to show to future contributors how new functions can be added to the UNDO management system.

Contents

1 Basic Ideas

When we discussed how to implement UNDO functionality for the DeViSoRGrid2 application, we had two different approaches in mind: Using a kind of *journal* or *log* to store information about everything that has been performed seemed to be the easier approach: Every operation would be logged, and by calling the UNDO operation, the whole journal would have been traversed, performing every operation except the last one again. Unfortunately, this required vast amounts of memory while still showing very poor performance results.
So, we came up with the idea of implementing *inverse operations* and using a *stack* to store UNDO information. After each operation we declared *undoable*, all we needed to do was to push a pair containing the affected items and an operation code onto the stack. Calling UNDO then results in removing the top element (i.e. the last operation that has been performed) from the stack, determining which operation has been performed, choosing the correct inverse operation and applying it on the selected items.
This concept turned out to be reasonably fast, surprisingly easy to implement, and by using a stack of limited capacity we managed to keep memory requirements effectively low.
After having implemented the new UNDO feature, we realized that our concept did support REDO as well with only very few adjustments to the code. So, we added the REDO functionality as well.

2 Class Design

All core undo functionality is put into three classes, which we put into a package of their own, located as subpackage of the backend package (`devisor2.grid.backend.undo`). We wanted to encapsulate as much functionality of the UNDO operation as possible in separate classes and not merge the UNDO operation with other functions of the application. So, the core UNDO classes provide a very narrow interface to the rest of the application, and whenever an undoable action is performed, all the application has to do is gather all neccessary information needed to take back that operation and notify the `UndoManager` class about it. The actual undoing and redoing can be performed by just one single method call.
But let's take a look at the class design first:

2.1 UndoContainer.java

This class is usually not needed from outside the UNDO package. It serves as a container or storage object for all information needed to uniquely classify one undoable operation. To be more precise, an instance of this class stores three different items:

- A unique integer value to define the type of operation. The constants used for this are all declared in the `UndoManager` class.
- References to all items affected by the operation encoded by the ID integer, conveniently gathered into a `Vector` of `GridItem`s.
- In some cases, this is not enough to take back the operation completely. for example in affine transformations. In these cases, the `UndoContainer` provides storage for additional information.

Additionally, access methods for all these items are provided in this class.

2.2 UndoStack.java

This class extends the `Stack` class provided by Java. It basically adds the capacity limit: As long as the capacity limit is not reached, it works just like the regular stack. If too many items are pushed onto the stack, the oldest item (i.e. the one at the bottom of the stack) is removed and cannot be recovered.
Additionally, easy access methods are provided, which automatically perform the casting to instances of the `UndoContainer` class.

2.3 UndoManager.java

This is the most important class in the UNDO package, because it contains all functionalities. Despite this fact, its interface to the rest of the application is very narrow: Just three public methods are provided:

- The `notifyUndo (opCode, items, info)` method has to be called by the application whenever an undoable operation is performed. Note that the application is responsible for passing ALL neccessary information to this method, otherwise, correctness and stability can not be guaranteed. This method then just creates a `UndoContainer` instance and pushes it onto the stack.
- The actual undoing is performed when the `performUndo ()` method is called: The top element is removed from the stack, and according to the encoded ID, the operation is undone. Before that, all neccessary information to redo the undo step are gathered and pushed on a second stack storing the `UndoContainer` instances responsible for REDO.
- The last method provided by the manager class is called `performRedo ()`. It works just like the corresponding method for UNDO described above.

3 Connecting The UNDO Package With The Application

As the UNDO functionality is needed globally throughout the application, it seems just natural to add a global reference to the `UndoManager` class to the `ControlCenter` instance of the application. Via this reference, all three interface methods are accessible. Every time an undoable action is performed, the `notifyUndo` method has to be called, and all neccessary information to perform the UNDO have to be passed as parameters to this method call.
In the event handlers for the UNDO and REDO buttons respectively menu items, the two `performUndo` and `performRedo` method are called.

4 A Full Scale Example Explained

In this last section, we will guide the reader through a full scale example, by explaining every step we performed when implementing the undo functionality for the `ADD NODE` operation.

- First of all, in the core classes, two new methods were added: `addNode (Node)` can be called to add a newly created node to the domain, and `delNode (Node)` removes the node again.
- After that, we implemented the code to add nodes to the domain, this took place in the `GridListener` class.
- Next up, a new UNDO ID was created in the `UndoManager` class, called `UNDO_ADDNODE`.
- In the `GridListener`, we added the call to `notifyUndo` at the end of the node adding code. In the `UndoManager` class, a corresponding case block was added to the mentioned method.
- After that, we added two corresponding case blocks to the `performUndo` and `performRedo` methods: In the UNDO method, add we needed to do was to push the operation over to the redo stack, and to call `delNode` with the node passed as parameter to the container instance removed from the top of the stack. In the REDO method, we called `addNode` respectively.
- Note that we did not need the third parameter of the `UndoContainer` for adding nodes. To illustrate how this parameter can be used, consider implementing some affine transformation. The transformation matrix has to be saved after performing the operation, because its inverse is needed to perform the UNDO. So, we passed it as third parameter of the `notifyUndo` method.

We hope that this example will enable the reader to add the UNDO operation for every new feature added to the DeViSoRGrid2 application. We recommend taking a look at the code, especially in the `GridListener` and `UndoManager` classes.