The Dialog Model in the DeViSorGrid 2 Application

Dominik Göddeke

September 19th, 2001

This document describes the model we designed and implemented for the dialogs in the DeViSoRGrid 2 application. After a short introduction, we describe the docking mechanism we implemented for our dialogs. Special focus is set on implementational details to make sure the reader is able to add his own dialogs to the application whenever neccessary.

Contents

1 Motivation

When we designed the new version of the DeViSoRGrid 2 application, we had three goals in mind concerning dialogs:

- Dialogs should be dockable, this means it should be possible to *"glue"* dialogs to the border of the `MainFrame` so that they trail the `MainFrame`'s movements on the screen. Additionally, dialogs should reappear at their last position whenever they are displayed again.
- The whole application should provide a pluggable look and feel, thus the visible dialogs should change their look and feel whenever the main window does so.
- Dialogs should be displayed fast, we didn't want to wait all day while the GUI of the dialogs is dynamically created every time the dialog is displayed.

So, it appeared only naturally that we needed a dialog model which allows for easy access to all dialogs at any time. In a preliminary version, we implemented all dialogs as static, but this approach resulted in a lot of redundant code, thus not following the object oriented programming paradigm. The model now implemented uses one central storage class for all dialogs, only through which dialogs are accessible. Operations which apply to all dialogs are implemented only once in this class. For example, all dialog initialisation is collected in one method of this class, allowing for fast displaying time at the cost of a slightly longer startup time and slightly higher memory requirements, as all dialogs are stored in memory at all times. But this proved to be no major disadvantage.

2 Structure of the Docking Functionality

The idea behind our *Dockable Dialogs* is the following: Some functions require a dialog to be displayed at all times, for example when you are setting edge stati, you click on an edge, then change something in the dialog, click on another edge and so on. If the dialog would just be displayed centered on the `MainFrame`, it would disappear behind the `MainFrame` every time you click somewhere in the `MainFrame`, and if it were modal, this functionality would be impossible to achive. So, we wanted the dialog to appear at one fixed position (which the user can change of course), even after restarting the program. Resizing and moving of the `MainFrame` should result in a *trailing* movement of the dialog.

2.1 The Docking Model

When a dialog is displayed for the first time, it appears at a fixed initial position. When the user moves the dialog, it can be docked (or "glued") to the border of the `MainFrame`. To do so, the center of the dialog has to be moved anywhere near the border, the default dock margin is ten per cent of the `MainFrame`'s size in each direction. When the dialog is docked, all `MainFrame` movements result in a corresponding movement of the dialog as well, so that the relative position of the dialog to the `MainFrame` does not change. When a dialog is docked, it is even moved when it is not visible. Its last position is stored when the user saves his default options file and restored when it is displayed again. When a dialog is not docked, it will appear at its initial position, and it won't trail the `MainFrame`'s movements.

2.2 The Abstract Superclass

The *trailing* mechanism is implemented in an abstract superclass called `DockableDialog`. This class provides the neccessary methods which only need to be called from the `MoveListener` of the `MainFrame`. Technically, these methods are pretty straight-forward, although the implementation proved to be a bit tricky. We recommend taking a look at the code if you want to know more: First, the neccessary attributes are declared, followed by a couple of getters and setters. The actual docking methods are called `calcDockable()` and `calcPosition()`, and of course some functionality is put in the

`DockableDialogMoveListener`, an anonymous inner class. For usability reasons, we also added a method called `rePosition()`, which basically moves the dialog to its newly calculated position.

2.3 Subclass Structure

The subclasses which actually implement some application functionality must extend the `DockableDialog` abstract superclass. The only methods they must implement are `getInitialPosition()` which returns the dialog's initial position, and `refreshGUI()`, which might not even be neccessary. We added this method nonetheless, because we needed one central method to be called when a dynamic change of the dialog becomes neccessary. A little example will explain this: Assume that a dialog containing a combo box to select boundaries from is visible, and that the user presses the *new boundary* button in the toolbar. As a result, the combo box in the dialog would no longer contain valid data, so it has to be refreshed. By putting this "refreshment code" into the mentioned method, all that has to be done in the event handler of the *new boundary* operation is call this method for all dialogs affected.

3 The `DialogManager` class

In this section, the central dialog managing and storing class be be explained in full detail. After this section, you will be able to follow the full-scale example presenting in the last section of this paper.

3.1 General Idea

To avoid redundant code at multiple locations throughout the application, we decided to implement one central storage and management class for all dialogs of the application. In this class, methods for the following tasks are provided: initialising the dialogs, extracting the docking information from the `Options` class, writing back docking information to the `Options` class, accessing single dialogs through a unique ID, and refreshing the GUI of all dialogs if neccessary.

3.2 Methods Provided by the `DialogManager` Explained in Detail

`Constructor` The constructor just initialises the array the dialogs are stored in and sets several references correctly. It is extremely important that the `DialogManager` is created only after the `ControlCenter` and Mainframe have already been created.

`initDialogs()` This method creates an instance of each dialog registered with the `DialogManager` and stores it in the array. Afterwards, the docking information is extracted from the `Options` class and each dialog's position is restored accordingly.

`hide(int)` **and** `hideAll()` These methods are the counterpart to the `initDialogs()` method, the docking information of a single or of all dialogs are written back to the `Options` instance.

`refreshAllGUI()` This method is pure convenience: It just broadcasts the refresh call to all dialogs registered with the `DialogManager`.

`refreshDialogs()` This method again is pure convenience: It hides all dialogs (thus storing docking information), recreates their GUI, and displays those dialogs again on the screen that were visible before. This is extremely practical when the look&feel of the application is changed.

4 Including Dialogs into the Application

In this final section, we will work through a full-scale example of creating a new dockable dialog and adding it to the application:

1. Create a new class (in the `devisor2.grid.GUI.dialogs` package). Without loss of generality, let us assume this class is called `ExampleDialog`. Make sure this class extends the `DockableDialog` superclass. Implement the two methods `getInitialPosition()` (note that the position is given relative to the `MainFrame`'s top left corner) and `refreshGUI()`, for which in this example an empty implementation will suffice. Feel free to create as much GUI as you want. Note that you have access to the `MainFrame` and the `ControlCenter` of the application via the `parent` and `cc` attributes respectively.

2. To register your new dialog with the `DialogManager`, there is not much to do:

   - Increment the `DIALOGCOUNT` variable by one.
   - Create a unique ID for the dialog by adding the following line of code:

     ```
     public static final int EXAMPLEDIALOG = xyz;
     ```

     where `xyz` is the successor of the last integer already used as an ID for a dialog.
   - Use this ID to add the initialisation of your dialog to the `initDialogs()` method:

     ```
     dialogs[EXAMPLEDIALOG] = new ExampleDialog(parent);
     ```
   - That's it, the dialog manager will take care of creating the neccessary entries in the `Options` class during runtime.

3. Whenever you want to display your new dialog, all you need is a method call like this:

   ```
   cc.dm.get (cc.dm.EXAMPLEDIALOG).setVisible (true);
   ```

   where `dm` is a reference to the global `DialogManager` instance of the application.

4. Hiding the dialog is just as easy:

   ```
   cc.dm.hide (cc.dm.EXAMPLEDIALOG);
   ```

5. If you need to access methods from the dialog, it should be obvious that this can be accomplished just like the call to the `setVisible()` method above. If you need to call methods unique in your dialog and not part of the `DockableDialog` interface, use a class cast like this:

   ```
   ExampleDialog dialog = (ExampleDialog)cc.dm.get (cc.dm.EXAMPLEDIALOG);
   dialog.myMethod (myParameter);
   ```

6. Last but not least, compile the application. Make sure to display the dialog once to check if it complies to your expectations. The docking information of the new dialog will automatically be saved to disk when you shut down the application.