JIDE Diff Developer Guide

Contents

Purpose of This Document

In computing, **diff** is a file comparison utility that outputs the differences between two files. It is typically used to show the changes between a file and a former version of the same file. Diff displays the changes made per line for text files. The output is called a diff or a patch since the output can be applied with the Unix program patch. The output of similar file comparison utilities are also called a "diff". Like the use of the word "grep" for describing the act of searching, the word diff is used in jargon as a verb for calculating any difference. [1]

In modern applications, a diff utility is very useful too. In addition to the traditional development applications such as Java IDEs or source version control systems where a diff utility is necessary, applications in finance, computing, network management also need diff utility to compare text files, messages, xml files etc.

JIDE Diff brings a diff/merge component for Swing applications. It leverages *JIDE Code Editor* to view text files and provides compare/merge features on top of the CodeEditor. However, since the algorithm and the infrastructure layer is generic which means it can be customized to compare any type of objects.

---

[1] http://en.wikipedia.org/wiki/Diff

Features

Here are the main features of *JIDE Diff*.

- Compares text files side by side using CodeEditor.
- Clearly indicate the difference of the two files by linking the changes using the colored lines.
- Separate changes in the code editor using colors and lines.
- Allows editing of the final result file when comparing.
- Three way merging support

Diff Algorithm

The diff algorithm is implemented in a class called *Diff*. Thanks for Jeff Pace's contribute to the algorithm which he shared at http://www.incava.org/projects/java/java-diff/. We took it and used it in JIDE Diff with some minor modifications.

The diff algorithm can compare any two arrays. To compare two arrays, say from[] and to[], you just need to do

```
Diff<T> diff = new Diff<T>(from, to);
List<Difference> differences = diff.diff();
```

The type T is the data type of the arrays "from" and "to". *Difference* is a class that tells which elements in the arrays are changed (inserted, deleted or changed).

A *Difference* consists of two pairs of starting and ending points, each pair representing either the "from" or the "to" collection passed to *Diff*. If an ending point is -1, then the difference was either a deletion or an addition. For example, if *getDeletedEnd()* returns -1, then the difference represents an addition. The values of those points are the array index in the "from" and "to" array.

Merge Algorithm

Merging is the act of reconciling multiple changes made to different copies of the same file. Most often, it is necessary when a file is modified by two people on two different computers at the same time. Later, these changes are merged, resulting in a single new file that contains both sets of changes.

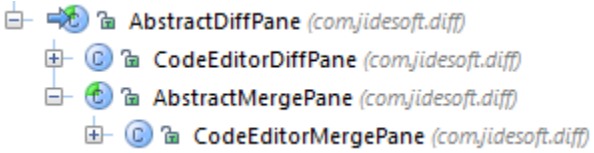
A two-way merge performs an automated difference analysis between a file 'A' and a file 'B'. This method considers the differences between the two files alone to conduct the merge and makes a "best-guess" analysis to generate the resulting merge. Consequently, this type of merge is usually the most error prone and requires user intervention to verify and sometimes correct the result of the merge prior to completing the merge event.

A three-way merge is performed after an automated difference analysis between a file 'A' and a file 'B' while also considering the origin, or parent, of both files (usually the parent is the same for both). This type of merge is more likely to be usable in revision control systems, which can guarantee that such a parent exists and is known. The merge tool examines the differences and patterns appearing in the changes between both files as well as the parent, building a relationship model to generate a merge of files 'A', 'B', and the parent 'C', to produce a new revision 'D'. Comparing with two-way merge, this merge is the most reliable and has performed well in practice. It has also required the least amount of user intervention, and in many cases, requiring no intervention at all (depending upon the complexity of the merge) making the process eligible for task automation.

In *JIDE Diff*, the merge algorithm is implemented in a class called *Merge*. It is a three-way merge.

JIDE Diff

JIDE Diff has two main components – *CodeEditorDiffPane* and *CodeEditorMergePane*. Both use *CodeEditor* as the text viewing component. In order to support other view component, we introduced abstract level class for each of the pane. Here is the class hierarchy.



AbstractDiffPane

AbstractDiffPanel is the base for the diff/merge panes. Not matter it is a diff pane or a merge pane, there are a toolbar, a status bar and content area. The content pane is a JideSplitPane. It has either 2 or 3 panes for diff and merge respectively. The createPane(int index) method is an abstract class that subclass can override to create the pane for the JideSplitPane. The createDiffDivider(int index) is used to create the divider between two panes. The divider has colored lines and shapes to show what the differences are.

The methods that subclasses can override are

- createPane(Object, int): override to use a different component to display the objects to be compared.
- createToolBar(): override to create a different component for the toolbar.
- createStatusBar(): override to create a different component for the status bar.
- createLegendBar(): override to customize the legend bar. It is part of the status bar.
- initLayout(JComponent, JComponent, JComponent)}: override to arrange the pane, the toolbar and the status bar.

For the toolbar, you can customize it to add more buttons. Here are the steps.

1. Assign a unique name for the action/button.

2. Added entries to diff.properties following the existing entries.
3. Override createActions method. After calling super.createActions(…), create a new action and put it to _actions field which is a map mapping from the name to the action. Please make sure the action's actionCommand is same as the name.
4. Override customizeToolBar and call toolBar.add(createButton(_actions.get(name))).

AbstractMergePane

AbstractMergePane extends *AbstractDiffPane* to add one more pane since it is for a three-way merge. It also adds one more button to the toolbar using the way described in the section above.

CodeEditorDiffPane

Here is what a *CodeEditorDiffPane* looks like under Window L&F.

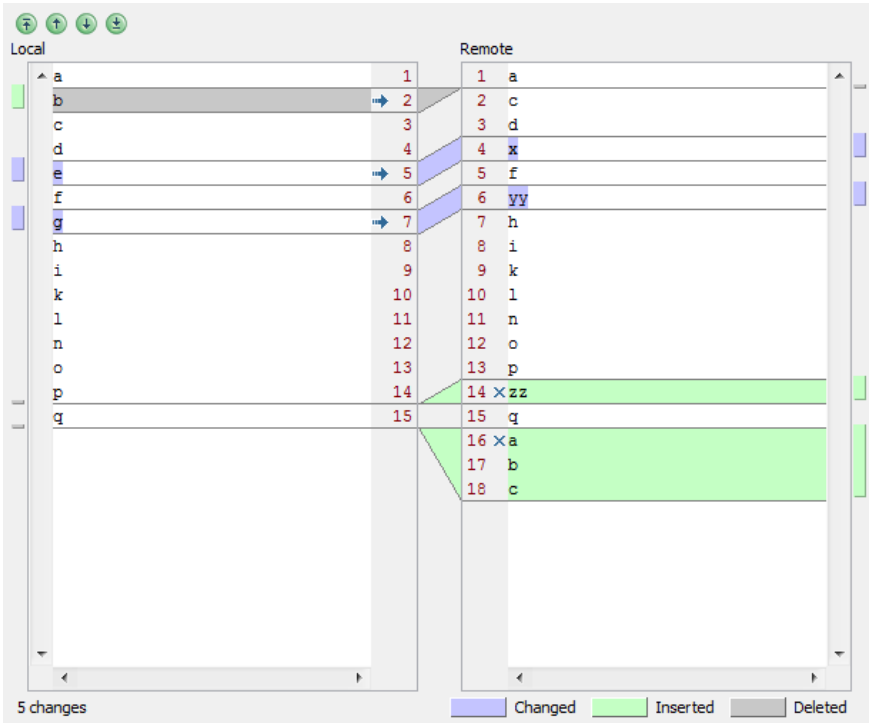


Figure 1 JIDE Diff Pane

CodeEditorDiffPane subclasses *AbstractDiffPane* and implemented *createPane* and *createDivider* methods. The *createPane* method uses *CodeEditor*. We used *CodeEditor* is because it supports color syntax, margin component and flexible highlight feature. In order to show the differences clearly, the highlight feature is very important. The margin component is used to place controls to accept or discard the change. Of course, if you have a 3rd party code editor component that meets the requirement, you can override *createPane* method to use it too.

The API for CodeEditorDiffPane is actually very simple.

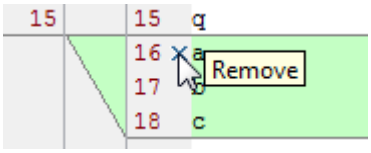
```
CodeEditorDiffPane diffPane = new CodeEditorDiffPane();
diffPane.setFromTitle("Local");
diffPane.setToTitle("Remote");
diffPane.setFromText(fromText);
diffPane.setToText(toText);
```

You need to have two pieces of text. The first piece of text is treated as “from” or “original” or “source”. The second piece of text is treated as “to” or “modified” or “destination”. You will set the text using *setFromText* and *setToText* methods. Then all you need to do is to call *diff()* to run the diff algorithm and show the differences on the UI.

```
diffPane.diff();
```

The diff() will return a list of differences. If you are interested in it, you can capture the return value. In most cases, the return value can be ignored.

CodeEditorDiffPane provides buttons on the margin area of the code editor to allow user to accept or ignore the changes. See below.



After user finishes the diff process, you call getToText() to get the final result and save it somewhere if user decides to keep it.

CodeEditorMergePane

Here is what a *CodeEditorMergePane* looks like under Window L&F with Office 2007 style.

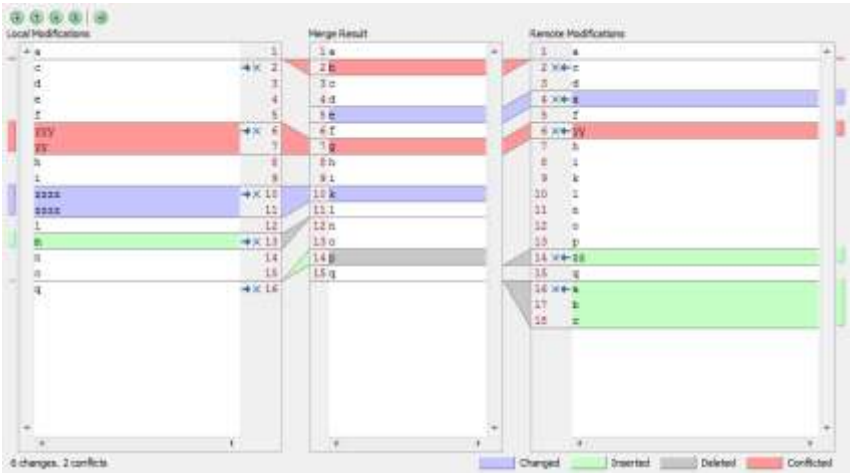


Figure 2 JIDE Merge Pane

As you can see, *CodeEditorMergePane* has three panes. In the real life, you need to merge because the file is modified at the same time by two people. For example, in the source code control system, two developers checked out the same file, the first developer modified it and checked in. The second developer is modifying the file at the same time. Now when he/she updates, he/she will have to merge the changes. Using this example, the left pane is the local modifications by the second developer. The middle pane is the base without the two developers' modifications. The right pane is the modifications from the first developer. Here is what the code looks like for this scenario.

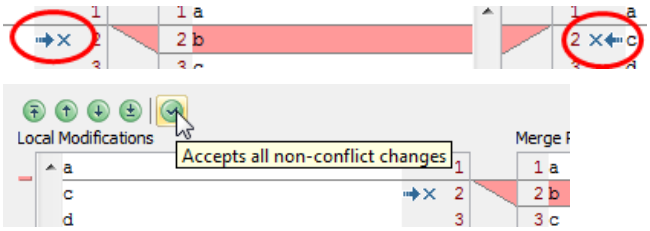
```
CodeEditorMergePane diffPane = new CodeEditorMergePane(..., ..., ...);

diffPane.setFromTitle("Local Modifications");
diffPane.setToTitle("Merge Result");
diffPane.setOtherTitle("Remote Modifications");
```

Then you call merge() to run the three-way merge algorithm and show the differences and/or conflicts on the UI.

```
diffPane.merge();
```

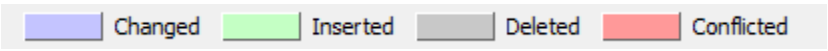
CodeEditorMergePane provides buttons on the margin area of the code editor to allow user to accept or ignore the changes. There is also a toolbar button to accept all non-conflicting changes. For the conflicts, user has to review it and decide what to do. See below.



After user finishes the merging process, you call getToText() to get the merge result and save it somewhere if user decides to keep it.

Color of Legends

On the bottom of the diff pane and the merge pane, you can see the color boxes. We used different color to indicate the meaning of the changes.



Those colors can be customized through UIDefaults. All four UIDefaults are defined in a class called *DiffUIDefaultsCustomizer*. You can create your own UIDefaultsCustomizer and add it to LookAndFeelFactory if you want to change it for all diff and merge panes.

```
public class DiffUIDefaultsCustomizer implements LookAndFeelFactory.UIDefaultsCustomizer {
   public void customize(UIDefaults defaults) {
      Object[] uiDefaults = new Object[]{
            "Diff.changed", new ColorUIResource(196, 196, 255),
            "Diff.deleted", new ColorUIResource(200, 200, 200),
            "Diff.inserted", new ColorUIResource(196, 255, 196),
            "Diff.conflicted", new ColorUIResource(255, 153, 153),
```

```
        };
        defaults.putDefaults(uiDefaults);
    }
}
```

If you just want to change the colors for a particular diff/merge pane, you can call *setChangedColor*, *setInsertedColor*, *setDeletedColor*, or *setConflictedColor* to adjust the colors.

Internationalization Support

All Strings used in *JIDE Diff* are contained in one properties file called diff.properties under com/jidesoft/diff. Some users contributed localized version of this file and we put those files inside jide-properties.jar. If you want to support languages other than those we provided, just extract this properties file, translated to the language you want, add the correct postfix and then jar it back into jide-properties jar. You are welcome to send the translated properties file back to us if you want to share it.