Einführung in Software Engineering WS 10/11

Fachbereich Informatik

Dr. Michael Eichberg
eichberg@informatik.tu-darmstadt.de

Assistent: Ralf Mitschke
mitschke@st.informatik.tu-darmstadt.de

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungsblatt 12 (10 Punkte): Design Patterns

Abgabeformat: Reichen Sie ihre Lösung per SVN ein. Jede Übung muss in einem eigenen Ordner ***ex<Number>*** *(<Number> = 01, 02, ...)* in Ihrem Gruppenverzeichnis eingereicht werden. Während der Übungsbearbeitung können Sie Ihre Lösungen beliebig oft in das SVN hochladen (per Commit). Wir prüfen die Zeit der Einreichung Ihrer Lösungen unter der Benutzung des SVN Zeitstempels.

Erstellen Sie für Lösungen der Aufgaben, die keinen Quelltext erfordern, eine PDF-Datei mit dem Dateinamen ***solution.pdf***. Dies gilt auch für alle UML-Diagramme, die Sie erstellen. Die Basisanwendung wird als Eclipse-Projekt vorgegeben. Ihr eigener Code muss entsprechend in den dafür vorgesehenen Verzeichnissen (***/src*** oder ***/test***) erstellt werden.

Abgabetermin: 16.02.2011 - 24:00 Uhr

Aufgabe 1 (4 Punkte)

Ziel: *Beurteilung eines objekt-orientierten Entwurfs*

Der folgende Text entstammt einem Artikel zum Thema Filtern einer Liste in Java Swing. Der Artikel stand von 2005 bis Anfang 2010 auf http://java.sun.com Online, ist aber heute nicht mehr verfügbar.

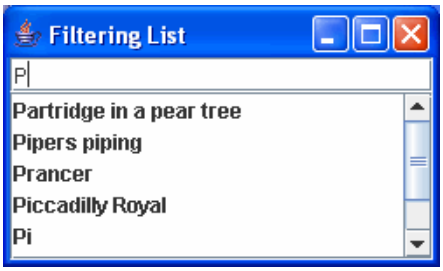
Studieren Sie den Artikel und beurteilen Sie den vorgeschlagenen Entwurf. Schauen Sie sich gegebenenfalls auch den Code weiterer hier verwendeter Klassen an, um deren Kontrakte und Funktionsweise zu verstehen. Machen Sie sich zuerst selber Gedanken über die Vorteile und Nachteile, die Sie an diesem Design erkennen. Diskutieren Sie Ihre Erkenntnisse und Schlussfolgerungen mit Ihrem Team.

Notieren Sie alle Punkte die Ihnen nachteilig erscheinen; geben Sie die entsprechenden Methoden und Code-Abschnitte an. Begründen Sie kurz warum/wann dies ein Nachteil ist. Achten Sie bei Ihren Überlegungen auch auf die Erweiterbarkeit und Wiederverwendbarkeit dieses Designs. Überlegen Sie sich hierzu weitere Belange, die Sie für eine angezeigte Liste haben können und wie sich diese mit dem gegebenen Design kombinieren lassen.

Providing a Filtering JList

[...] There is no feature in Java SE 6 for sorting and filtering a `JList`. However in this tip, you'll learn how to do something similar in J2SE 5.0 with a `JList`.

A common technique for prompting users to filter elements in a long list is to show a `JTextField` with the list. As the user types in the `JTextField`, the elements shown in the list are reduced to the set of matching entries.



The implementation of this type of feature for a `JList` requires two supporting elements: a model that filters its elements based on some text, and a text component that triggers the filtering action when the user types in a field.

Implementing the input field is the easier of the two, so it will be shown first. With the Swing component set, the model for a `JTextField` is a Document. To monitor input to the Document, you attach a `DocumentListener` to the model. There are three methods for this listener, allowing you to react differently for input, removal, and change events:

```
public void insertUpdate(DocumentEvent event)
public void removeUpdate(DocumentEvent event)
public void changedUpdate(DocumentEvent event)
```

The changeUpdate() method is related to attribute changes in the model. These can be ignored. Because the same filtering action needs to happen for the two other methods, they just need to call the same method to be created on the custom model. Here's the full definition of the JTextField to be associated with the filtering JList:

```
JTextField input = new JTextField();
String lastSearch = "";
DocumentListener listener = new DocumentListener() {
  public void insertUpdate(DocumentEvent event) {
          Document doc = event.getDocument();
          lastSearch = doc.getText(0, doc.getLength());
          ((FilteringModel) getModel()).filter(lastSearch);
  }
  public void removeUpdate(DocumentEvent event) {
          Document doc = event.getDocument();
          lastSearch = doc.getText(0, doc.getLength());
          ((FilteringModel) getModel()).filter(lastSearch);
  }
  public void changedUpdate(DocumentEvent event) {
  }
};
input.getDocument().addDocumentListener(listener);
```

Instead of limiting the usage to a JTextField created by the JList, an installJTextField() method is provided that associates the listener to the given component, and an uninstall method for removing the listener. This allows the user of the filtering list to offer its own JTextField, instead of creating the default one.

```
public void installJTextField(JTextField input) {
  input.getDocument().addDocumentListener(listener);
}
```

```
public void unnstallJTextField(JTextField input) {
  input.getDocument().removeDocumentListener(listener);
}
```

The filtering model comes next, with its filter() method required by the DocumentListener. For this, you simply need to maintain two lists of elements: the source list and the filtered list. With the help of AbstractListModel, you need to implement a few methods, as follows:

- Constructor
- Adding method to add elements to the model
- getSize() to get its size
- getElementAt() to get an element back.

The constructor creates the two List objects. It doesn't matter what the elements in the List are, so create them as a List of Object types:

```
List<Object> list;
List<Object> filteredList;

public FilteringModel() {
  list = new ArrayList<Object>();
  filteredList = new ArrayList<Object>();
}
```

Adding elements to the model is done by adding them to the source model (list), and then telling the model to filter itself. This could be optimized to only filter the new element, however for now, adding an element calls the same filter() method as is called to filter the entire list. (Note that input into the Document through the DocumentListener calls filter() to filter the entire list.) So even though you are adding one element, it still clears the entire list, and adds each element that matches the last search term (which might be empty for a new list).

```
public void addElement(Object element) {
  list.add(element);
  filter();
}
```

The size of the list model is the filtered list size, not the source size:

```
public int getSize() {
  return filteredList.size();
}
```

As is the case for getting the size, getting an element fetches it from the filtered list, not the original source list. This works provided that you don't go past the end of the list:

```
public Object getElementAt(int index) {
  Object returnValue;
  if (index < filteredList.size()) {
          returnValue = filteredList.get(index);
  } else {
          returnValue = null;
  }
  return returnValue;
}
```

The final `filter()` method provides the bulk of the work. Because you won't know if the search string is expanding or contracting the result set, it is easiest to clear out the filtered list and add items from the source list that match the input field. Matching could be done from the start of the string or anywhere in the text. Here is an example of the latter search method. It allows "A" to find both elements that start with "A" or have a capitalized "A" anywhere:

```
void filter(String search) {
  filteredList.clear();
  for (Object element : list) {
          if (element.toString().indexOf(search, 0) != -1) {
                  filteredList.add(element);
          }
  }
  fireContentsChanged(this, 0, getSize());
}
```

The searching here is case-sensitive. In addition to changing the searching to the start of the string, you can also modify it to be case insensitive.

You can also sort the results after adding the elements to the filtered list. This requires you to know something about the contents of the model. At present, searching uses the `toString()` results -- it does not assume that the model contains elements of a Comparable type, like a String, that can also be sorted.

The complete filtering `JList` is shown next with its `ListModel` as an inner class. The custom `ListModel` is also the `DocumentListener` for the text component. This might seem odd at first because the filtering is localized to the model, but it appears to be the best place to define the behavior.

```
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import java.util.*;

public class FilteringJList extends JList {
  private JTextField input;

  public FilteringJList() {
          FilteringModel model = new FilteringModel();
          setModel(new FilteringModel());
  }

  /**
   * Associates filtering document listener to text component.
   */
  public void installJTextField(JTextField input) {
          if (input != null) {
                  this.input = input;
                  FilteringModel model = (FilteringModel) getModel();
                  input.getDocument().addDocumentListener(model);
          }
  }
```

```
/**
 * Disassociates filtering document listener from text component.
 */
public void uninstallJTextField(JTextField input) {
	if (input != null) {
		FilteringModel model = (FilteringModel) getModel();
		input.getDocument().removeDocumentListener(model);
		this.input = null;
	}
}

/**
 * Doesn't let model change to non-filtering variety
 */
public void setModel(ListModel model) {
	if (!(model instanceof FilteringModel)) {
		throw new IllegalArgumentException();
	} else {
		super.setModel(model);
	}
}

/**
 * Adds item to model of list
 */
public void addElement(Object element) {
	((FilteringModel) getModel()).addElement(element);
}

/**
 * Manages filtering of list model
 */
private class FilteringModel extends AbstractListModel implements
		DocumentListener {
	List<Object> list;
	List<Object> filteredList;
	String lastFilter = "";

	public FilteringModel() {
		list = new ArrayList<Object>();
		filteredList = new ArrayList<Object>();
	}

	public void addElement(Object element) {
		list.add(element);
		filter(lastFilter);
	}

	public int getSize() {
		return filteredList.size();
	}

	public Object getElementAt(int index) {
		Object returnValue;
		if (index < filteredList.size()) {
			returnValue = filteredList.get(index);
		} else {
			returnValue = null;
		}
		return returnValue;
	}

	void filter(String search) {
		filteredList.clear();
		for (Object element : list) {
			if (element.toString().indexOf(search, 0) != -1) {
				filteredList.add(element);
			}
		}
		fireContentsChanged(this, 0, getSize());
	}
```

```
        // DocumentListener Methods

        public void insertUpdate(DocumentEvent event) {
                Document doc = event.getDocument();
                try {
                        lastFilter = doc.getText(0, doc.getLength());
                        filter(lastFilter);
                } catch (BadLocationException ble) {
                        System.err.println("Bad location: " + ble);
                }
        }

        public void removeUpdate(DocumentEvent event) {
                Document doc = event.getDocument();
                try {
                        lastFilter = doc.getText(0, doc.getLength());
                        filter(lastFilter);
                } catch (BadLocationException ble) {
                        System.err.println("Bad location: " + ble);
                }
        }

        public void changedUpdate(DocumentEvent event) {
        }
  }
}
```

At this point you need a test program. The key part of the program is the following six lines. The lines create the `JList`, add it to a `JScrollPane`, and then add that and the associated input field to the screen. The bulk of the test program adds elements to the model. [...]

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Filters {
  public static void main(String args[]) {
        Runnable runner = new Runnable() {
                public void run() {
                        JFrame frame = new JFrame("Filtering List");
                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                        FilteringJList list = new FilteringJList();
                        JScrollPane pane = new JScrollPane(list);
                        frame.add(pane, BorderLayout.CENTER);
                        JTextField text = new JTextField();
                        list.installJTextField(text);
                        frame.add(text, BorderLayout.NORTH);

                        String elements[] = { "Partridge in a pear tree",
                        "Turtle Doves", "French Hens", "Chi", "Psi", "Omega" };

                        for (String element : elements) {
                                list.addElement(element);
                        }

                        frame.setSize(250, 150);
                        frame.setVisible(true);
                }
        };
        EventQueue.invokeLater(runner);
  }
}
```

Compile the `FilteringJList` and Filters classes. Then run Filters. You should get a filtering `JList`.

Provided that your list elements have "good" `toString()` representations, this approach for a filtering `JList`, with its associated `JTextField`, works well. For more complex list elements, consider defining a Filter interface that could be passed into the model for the filtering operation.

One thing not covered in this example is management of selection. By default, the `JList` doesn't alter the selection when the list model content changes. Depending on the desired behavior, filtering might try to preserve the selected element or always reset it to the first item in the list.

Although the underlying `JList` component doesn't support filtering directly, it does provide a smart type-ahead option. If you don't like the default behavior, you can override the `getNextMatch()` method (that method was added in J2SE 1.4).

Aufgabe 2 (6 Punkte)

Ziel: *Anwendung des Decorator Patterns*

Die Flashcards-Anwendung soll erweitert werden, so dass die Liste der Flashcards gefiltert und sortiert werden kann. Die Filterung soll über ein Suchfeld erfolgen. In der Liste sollen dann nur noch Karten angezeigt werden, deren Frageseite den im Suchfeld eingegeben Text enthält. Die Sortierfunktion soll verschiedene Kriterien Unterstützen, wie z.B. Sortieren Anhand des Erstellungsdatums oder Anhand des Datums an dem man eine Karte das letzte Mal erfolgreich gelernt hat. Abb. 1 zeigt einen Vorschlag, wie dies in der GUI umgesetzt werden kann.

Nutzen Sie im Folgenden das Decorator Pattern, um die Liste der Flashcards zu Filtern und zu Sortieren.

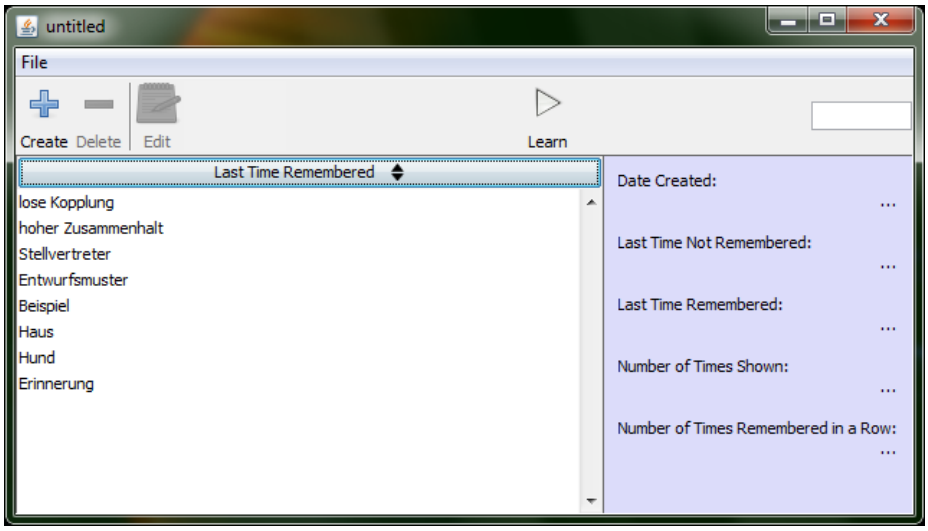


Abb. 1 Beispiel Flashcards-Anwendung mit Suchfeld zum Filtern und Sortierfunktion

a) Filtern der Liste (2 Punkte)

In dieser Aufgabe soll das Suchfeld in die Flashcards-Anwendung integriert werden. Sobald etwas in das Suchfeld eingegeben wird, soll sich sofort die Anzeige der Liste der Flashcards ändern. Es sollen nur noch Flashcards anzeigt werden, deren Vorderseite den im Suchfeld eingegebene Text enthält.

Erweitern Sie die Anwendung um ein Suchfeld. Sie können dazu Sie den unten stehenden Code verwenden. Dekorieren Sie anschließend die `FlashcardSeries` so, dass diese anhand der Sucheingabe gefiltert wird.

Code Beispiel für die Konfiguration der GUI (zu Teilaufgabe a)

Sie können das folgende Codebeispiel zur Erstellung eines Suchfelds verwenden:

```
public FlashcardsWindow(DefaultFlashcardSeries flashcards) {
[…]
    searchTextField = new JTextField(8);
    searchTextField.putClientProperty("JTextField.variant", "search");
    searchTextField.setMaximumSize(searchTextField.getPreferredSize());
    searchTextField.getDocument().addDocumentListener(new DocumentListener() {

            public void removeUpdate(DocumentEvent e) {
                    String searchText = searchTextField.getText();
                    // filter the list of flashcards
            }
            public void insertUpdate(DocumentEvent e) {
                    String searchText = searchTextField.getText();
                    // filter the list of flashcards
            }
            public void changedUpdate(DocumentEvent e) {
                    String searchText = searchTextField.getText();
                    // filter the list of flashcards
            }
    });
[…]
    toolbar.add(searchTextField);
[…]
}
```

b) Sortieren Anhand verschiedener Kriterien (2 Punkte)

In dieser Aufgabe soll die Flashcards-Anwendung um eine Sortierfunktion erweitert werden. Erweitern Sie die graphische Oberfläche der Anwendung um die entsprechende Funktionalität. Sobald eine Sortieroption gewählt wird, soll sich sofort die Anzeige der Liste der Flashcards ändern. Die Flashcards sollen in der Reihenfolge angezeigt werden, die der gewählten Sortierung entspricht.

Dekorieren Sie die `FlashcardSeries` anhand der ausgewählten Sortieroption. Die Anwendung soll die folgenden drei Kriterien als Sortieroption anbieten:

1) Erstellungsdatum, wobei die neueste Karte oben angezeigt wird
2) Datum des letzten erfolgreichen Lernens, wobei das älteste Datum oben steht
3) Wie häufig eine Karte erfolgreich gelernt wurde, wobei weniger oft gelernte Karten oben stehen

Code Beispiel für die Konfiguration der GUI (zu Teilaufgabe b)

Sie können das folgende Codebeispiel für eine Auswahlleiste der möglichen Sortieroptionen verwenden:

```
public FlashcardsWindow(FlashcardSeries flashcards) {
[…]
    // create menu item for one sorting category
    final JMenuItem dateCreatedMenuItem = new JMenuItem("Date Created");
    dateCreatedMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                    // sort the flashcard series
            }
    });
[…]
    // create a popup menu for choosing different sorting categories
    sortOrderPopupMenu = new JPopupMenu("Sort Order");
    sortOrderPopupMenu.add(dateCreatedMenuItem);
    sortOrderPopupMenu.add(lastTimeRemeberedMenuItem);
    sortOrderPopupMenu.add(rememberedInARowCountMenuItem);
    sortOrderPopupMenuDimension = sortOrderPopupMenu.getPreferredSize();
    // create a button for sorting categories
    sortOrderButton = new JButton("Date Created");
    sortOrderButton.setVerticalTextPosition(SwingConstants.CENTER);
    sortOrderButton.setHorizontalTextPosition(SwingConstants.LEFT);
    sortOrderButton.putClientProperty("JButton.buttonType", "roundRect");
    sortOrderButton.putClientProperty("JComponent.sizeVariant", "small");
    sortOrderButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                sortOrderPopupMenu.show(sortOrderButton,
                    sortOrderButton.getWidth()/2 - (sortOrderPopupMenuDimension.width/2),
                    sortOrderButton.getHeight()/2 - (sortOrderPopupMenuDimension.height/2)
                );
            }
    });
[…]
    // create the left side panel with sort button and list
    JPanel listPanel = new JPanel(new BorderLayout());
    listPanel.setOpaque(true);
    listPanel.setBackground(UIManager.getColor("Spinner.background"));
    listPanel.add(sortOrderButton, BorderLayout.NORTH);
    listPanel.add(listScrollPane);
[…]
    splitPane.setLeftComponent(listPanel);
}
```

c) Simultanes Filtern und Sortieren (0,5 Punkte)

Das Filtern und das Sortieren der Flashcards sollen natürlich auch gleichzeitig durchgeführt werden können. Verbinden Sie hierzu die beiden Decorator aus Aufgabe a) und b).

d) Dokumentation des Decorator Patterns (1,5 Punkt)

Erstellen Sie ein UML Klassendiagramm welches Ihre Implementierung des Decorator Patterns dokumentiert. Die Klassen sollten alle für das Pattern relevanten Methoden zeigen. Weitere Methoden sind nicht aufzuführen. Notieren Sie welche Klassen welche Rollen des Patterns inne haben (entweder im Diagramm, siehe „Design Pattern – Introduction“ Folie 27, oder extern in Ihrem Lösungsdokument).