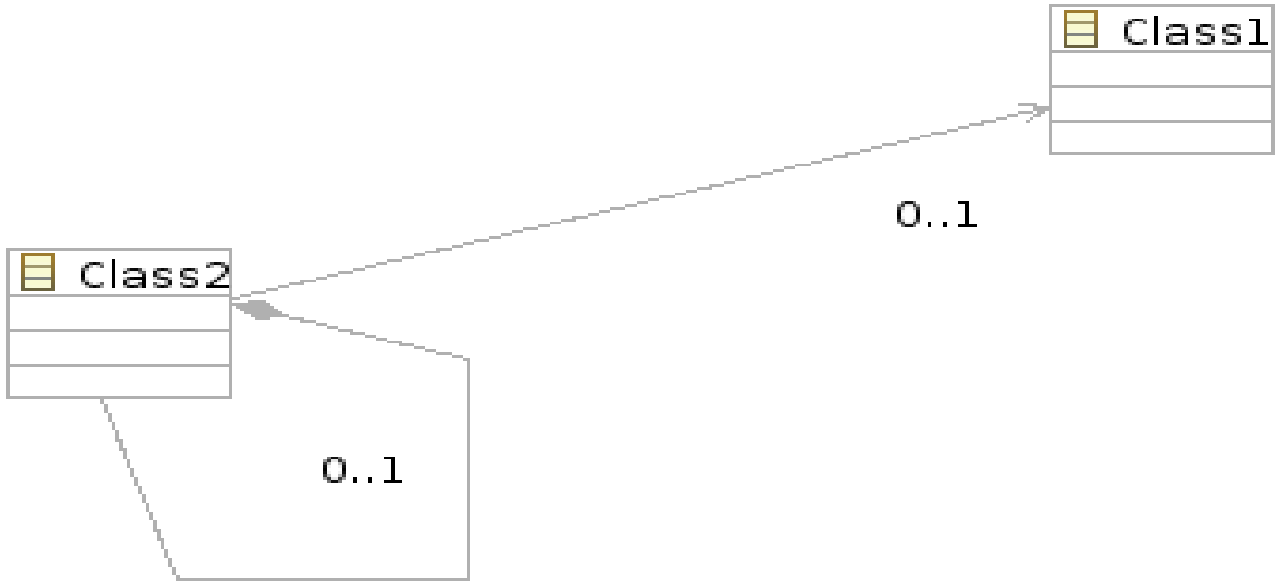
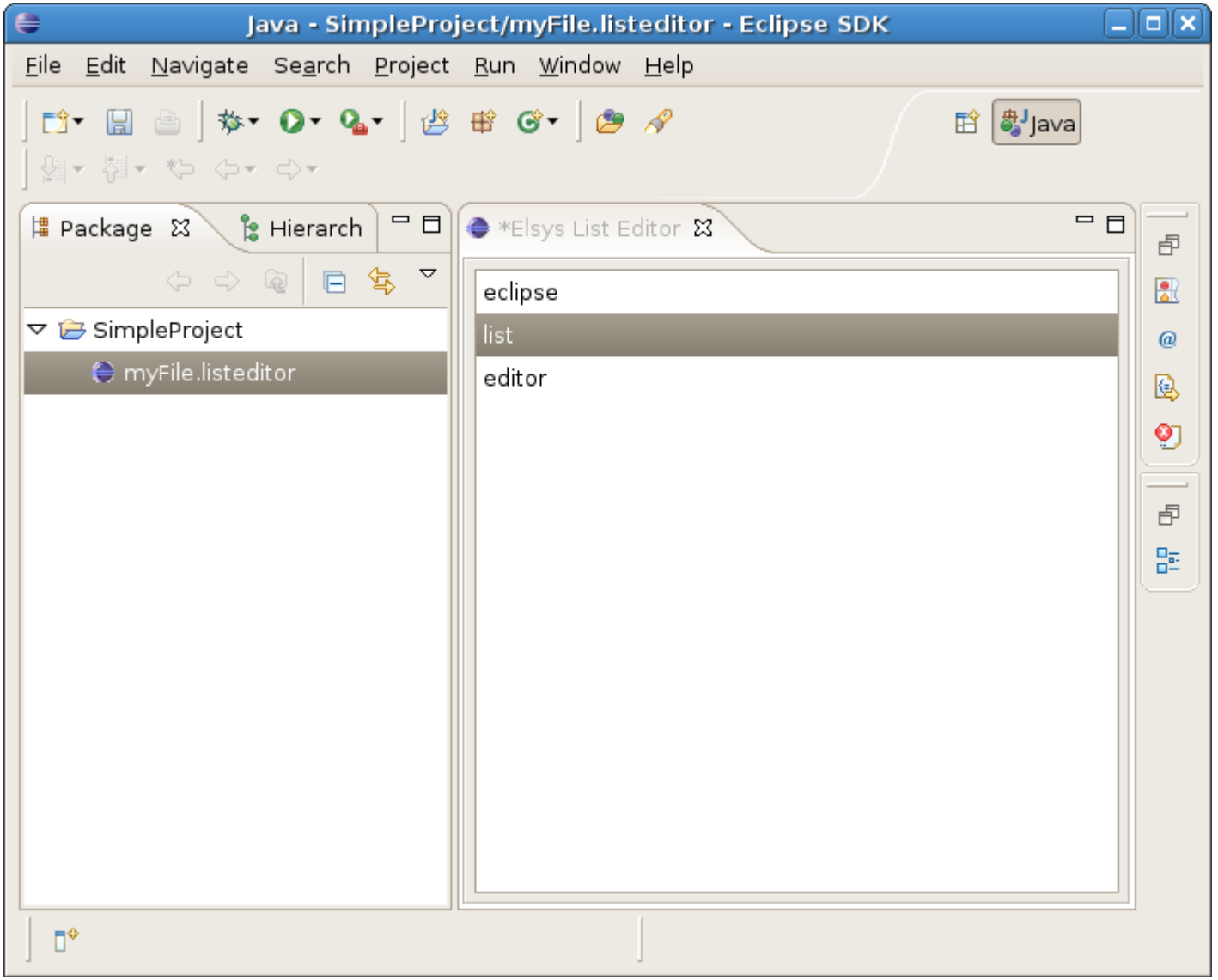


Editors @ Tues

Editors


Java - SimpleProject/myFile.listeditor - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Package
Hierarch
*Elsys List Editor
SimpleProject
myFile.listeditor
eclipse
list
editor

Действия

За добавяне и изтриване на елементи от списъка на редактора се използват действия. Действията може да се изберат от контексното меню. Логиката за добавяне и изтриване са отделени в:

- AddAction – действие за добавяне на елемент към списъка
- DeleteAction – действие за изтриване на елемент от списъка. Възможно е това действие да се казва RemoveAction или DeleteAction. Съгласно RFRS практиките:
  - Remove – елементът се премахва от избраното място, но продължава да съществува
  - Delete – елементът се прамахва и не продължава да съществува.

RFRS – Ready For Rational Software

http://www-304.ibm.com/jct09002c/isv/rational/rfrs_example.html

AddAction

```
public  class AddAction extends Action {

   private ListViewer fViewer;

   private ListEditor fEditor;

   private List<String> fContent;

   public AddAction(ListEditor editor, List<String>
content, ListViewer viewer) {
      super();
      setText("Add");
      fViewer = viewer;
      fEditor = editor;
      fContent = content;
   }
   public void run() {
      /* ... code missed ...*/
   }
}
```

AddAction

```
public  class AddAction extends Action
```

AddAction наследява `org.eclipse.jface.action.Action`. Action представлява имплементация по подразбиране за `org.eclispe.jface.action.IAction`. Обектите от тип IAction имат за цел да капсулират в себе си както визуалната информация така и логиката на действието.

```
public AddAction(ListEditor editor, List<String> content,
ListViewer viewer) {
        super();
        setText("Add");
        fViewer = viewer;
        fEditor = editor;
        fContent = content;
}
```

Конструкторът приема редактора, списъка който трябва да се редактира и viewer-а който трябва да се обнови след редакцията.

AddAction - run

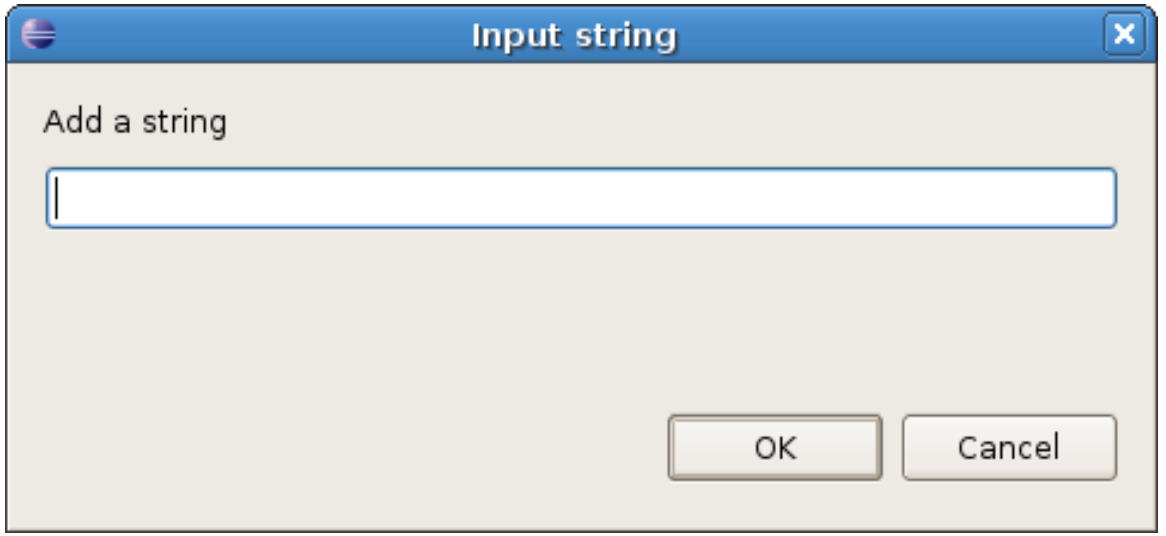
```
@Override
public void run() {
   InputDialog dialog = new
InputDialog(fEditor.getSite().getShell(),
         "Input string", "Add a string", "",
         new IInputValidator() {
            public String isValid(String newText) {
               return null;
            }
      });
   if (dialog.open() == IDialogConstants.OK_ID) {
      fContent.add(dialog.getValue());
      fViewer.refresh();
      fEditor.setDirty();
   }
}
```

AddAction - run

На потребителя се дава възможност да въведе произволен низ. За целта се използва диалоговя прозорец
`org.eclipse.jface.dialogs.InputDialog`

```
    InputDialog dialog = new

InputDialog(fEditor.getSite().getShell()...)
```



Създаването на нов диалог не предизвиква неговото отваряне.

AddAction - run

За да бъде визуално показан даден диалог е необходимо да се извика методът open().

```
if (dialog.open() == IDialogConstants.OK_ID) {
    fContent.add(dialog.getValue());
    fViewer.refresh();
    fEditor.setDirty();
}
```

Спътките, които трябва да се извършат след въвеждане чрез диалога са следните:

- `fContent.add(dialog.getValue());` **– промяна на модела**
- `fViewer.refresh();` **– обновяване на viewer-а.**
- `fEditor.setDirty();` **– промяна състоянието на редактора. Възможно е запаметяване**

DeleteAction

```
public  class DeleteAction extends Action {

   private ListViewer fViewer;

   private ListEditor fEditor;

   private List<String> fContent;

   public DeleteAction(ListEditor editor, List<String>
content,
         ListViewer viewer) {
   /* ... code missed ... */
   }
   public void run() {
   /* ... code missed ... */
   }
}
```

DeleteAction

За да се изтрие даден елемент е необходимо той първо да бъде избран. Следователно преди изпълнението на действието трябва да се определи кой е текущо избраният елемент във viewer-а.

Всеки viewer имплементира `org.eclipse.jface.viewer.ISelectionProvider.` **Интерфейсът** `ISelectionProvider` **може да бъде имплементиран от всички класове, които по някакъв начин предоставят "начин за избор на елемент". Сруктурата на избраните елементи е напълно произволна. Може да бъде избран текст, произволно структуриране елементи, част от графика и тн.**

За да определим текущо избрания елемент ще използваме метода `org.eclipse.jface.viewers.ISelectionProvider:getSelection()`

DeleteAction - run

```
@Override
public void run() {
    IStructuredSelection selection = (IStructuredSelection)
                                fViewer.getSelection();
    fContent.remove(selection.getFirstElement());
    fViewer.refresh();
    fEditor.setDirty();
}
```

- `fViewer.getSelection();` - Определя се избрания елемен.
- `fContent.remove(selection.getFirstElement());` - премахваме елемента от модела
- `fViewer.refresh();` - обновяваме viewer-а.
- `fEditor.setDirty();` - променяме състоянието на редактор.

DeleteAction

```
public  DeleteAction(ListEditor editor, List<String>
content,ListViewer viewer) {
        super();
        setText("Delete");
        fViewer = viewer;
        fEditor = editor;
        fContent = content;
        fViewer.addSelectionChangedListener(new
ISelectionChangedListener() {
            public void
selectionChanged(SelectionChangedEvent event) {
                if (((IStructuredSelection)
fViewer.getSelection()).isEmpty() == false) {
                    setEnabled(true);
                } else
                    setEnabled(false);
            }
        });
}
```

DeleteAction

Когато не е избран елемен действието за изтриване може да се забрани. Да се изобрази в сив цвят и да не се позволява неговото изпълнение.
org.eclipse.jface.viewers.ISelectionProvider.addSelectionChangeListene() е метод който позволява регистрирането на обект наблюдаващ текущо избраните елементи.

```
fViewer.addSelectionChangedListener(new
ISelectionChangedListener() {
   public void selectionChanged(SelectionChangedEvent
event) {
      if (((IStructuredSelection)
fViewer.getSelection()).isEmpty() == false) {
         setEnabled(true);
      } else
         setEnabled(false);
}});
```

В зависимост от това дали е избран елемент се определя дали действието е позволено. За целта се използва `setEnabled(boolean)`.

IProgressMonitor

Обектите от тип `IProgressMonitor` са предназначени за осъществяване на обратна връзка към потребителя. Ако дадена операция има нужда от дълго време за изпълнение се предпочита потребителят да бъде уведомен за извършената до момента работа както и за оставащата. За предпочитане е потребителят да може да прекъсне операцията.

За целта се използват обекти от тип `org.eclipse.core.runtime.IProgressMonitor`.
При имплементиране на метод приемащ аргумент от тип `IProgressMonitor` сме задължени да извършим определени действия с този обект. Пример за такъв метод е `org.eclipse.ui.part.EditorPart:doSave(...)`.

`IProgressMonitor` е добър пример за това, че интерфейсът на даден обект не са само предоставените методи. От значение са също така моментите в които трябва да се извика даден метод, задължителен ли е даден метод или не, синхронизиран ли е методът или не. Всичко това може/трябва да може да се установи от документацията му.

IProgressMonitor

```
public  void progressMethod(IProgressMonitor monitor) {
    try {
       monitor.beginTask("Tast name", 10);
       /* ... code missed ... */
    } finally {
       monitor.done();
    }
}
```

Ако ще се възползваме от подадения ни обект monitor то първият извикан метод трябва да е:

```
monitor.beginTask("Tast name", 10);
```

Визуално ще се изобрази започването на задача с име “Task name”.

След като сме приключили работа с monitor задължително трябва да извикаме `monitor.done()`. Най-удобно е да се използа `try/finally`, за да гарантираме условието и при възникване на изключения.

IProgressMonitor

След като е извършено определено количество работа може да изобразим това чрез `monitor.worked(...);`

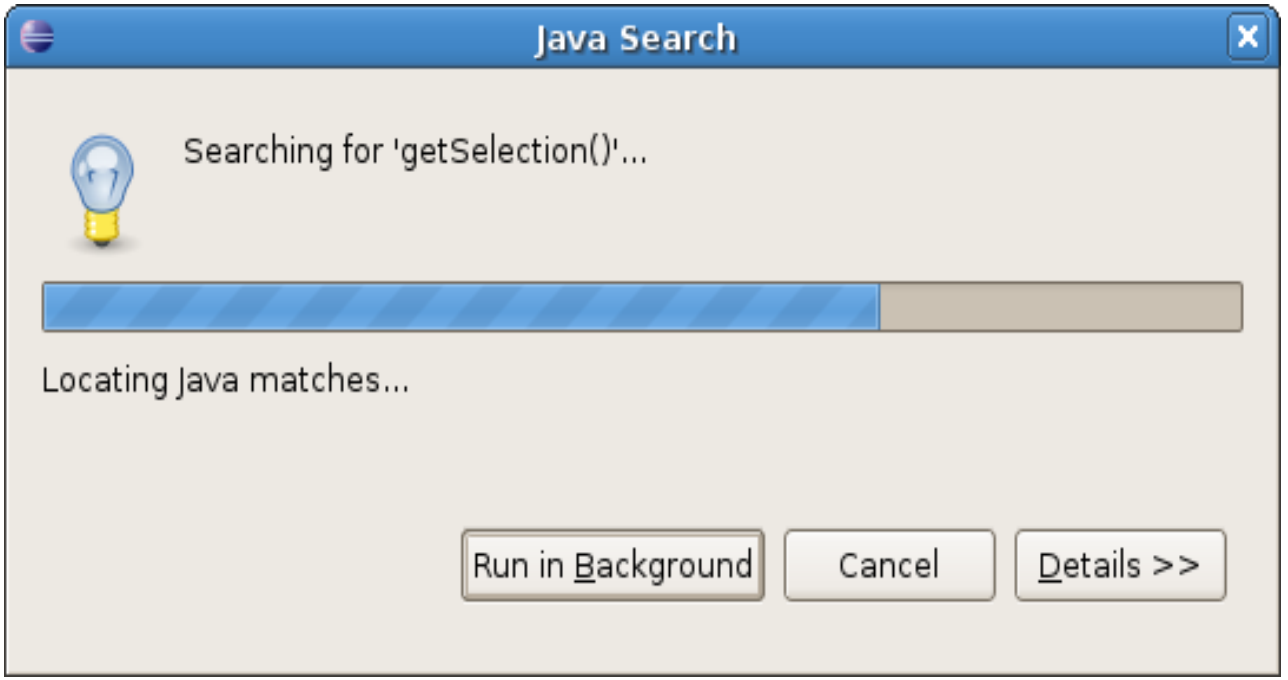
```
monitor.worked(2);
```

На потребителя се предоставя бутон “Cancel” с който може да се прекрати изпълнението на процеса. Методът `isCanceled()` проверява дали този бутон е натисна. Честотата, с която ще се извършват проверките трябва да е възможно най-висока, за да се осигури добро поведение на програмата. При прекъсване на операцията трябва да се хвърли изключение `org.eclipse.core.runtime.OperationCanceledException()`

```
if (monitor.isCanceled())
   throw new OperationCanceledException();
```

IProgressMonitor

Пример за това как визуално изглежда диалог оправляван от обект от тип IProgressMonitor:



Undo/Redo

Поддръжката на Undo/Redo е задължителна за един съвременен редактор. Undo/Redo функционалността се базира на Command Pattern http://en.wikipedia.org/wiki/Command_pattern

Всяка промяна от страна на потребителя се капсулира в отделна команда. Преди изпълнението си комантата запаметява текущото състояние на приложението. След изпълнението се командата може да бъде отменена – undo. След изпълнението си всяка команда се добавя към списък от изпълнение команди. По този начин се запазва историята на редактиране. Всяка команда може да възобнови състоянието на редактора такова каквото е било преди нейното изпълнение.

За реализацията на Undo/Redo ще използваме следните класове:

- `org.eclispe.core.runtime.IUndoableOperation`
- `org.eclipse.core.runtime.IUndoContext`
- `org.eclipse.core.runtime.IOperationHistory`
- `org.eclipse.ui.operations.UndoActionHandler`
- `org.eclipse.ui.operations.RedoActionHandler`

AddOperation/DeleteOperation

Към разработвания редактор добавяме два нови класа

- AddOperation
- DeleteOperation

Това са класовете, който ще капсулират в себе си действията по добавяне е изтриване. AddAction и DeleteAction ще създават и изпълняват обекти от тип AddOperation/DeleteOperation.

AddOperation/DeleteOperation имплементират `org.eclipse.core.command.operations.IUndoableOperation` и наследяват неговата имплементация по подразбиране - `org.eclipse.core.commands.operations.AbstractOperation`

AddOperation

```
public  class AddOperation extends AbstractOperation {

   private ListEditor fEditor;
   private String fAddedValue;

   public AddOperation(ListEditor editor) {
      super("Add");
   }

   public IStatus execute(IProgressMonitor monitor,
IAdaptable info)
         throws ExecutionException {}
public IStatus redo(IProgressMonitor monitor, IAdaptable
info)
         throws ExecutionException {}
   public IStatus undo(IProgressMonitor monitor,
IAdaptable info)
         throws ExecutionException {}
}
```

AddOperation - undo

Методът `execute()` се вика когато операцията трябва да се изпълни за първи път. В случая на AddOperation това означава добавяне към списъка:

```
   public IStatus execute(IProgressMonitor monitor,
IAdaptable info)  throws ExecutionException {
      InputDialog dialog = new
InputDialog(fEditor.getSite().getShell(),
            "Input string", "Add a string", "", new
IInputValidator() {
                public String isValid(String newText) {
                   return null;}});
      if (dialog.open() == IDialogConstants.OK_ID) {
         fAddedValue = dialog.getValue();
         fEditor.getContent().add(dialog.getValue());
         fEditor.getViewer().refresh();
         fEditor.setDirty();
      }
      return Status.OK_STATUS;
   }
```

AddOperation - undo

Методът `undo()` се вика когато операцията трябва да се отмени. В случай на AddOperation това означава изтриване от списъка.

В общият случай операцията трябва да запамети състоянието на редактора преди изпълнението си и в `undo()` да може да върне редактора към това състояние. AddOperation запаметява добавената стойност, така че при `undo()` да може да я премахне от списъка.

```
   public IStatus undo(IProgressMonitor monitor,
IAdaptable info)
          throws ExecutionException {
       fEditor.getContent().remove(fAddedValue);
       fEditor.getViewer().refresh();
       fEditor.setDirty();
       return Status.OK_STATUS;
   }
```

AddOperation - redo

Методът `redo()` се вика когато операцията трябва да се изпълни за пореден път. Разликата между redo и execute и, че в execute() трябва да се запамети състоянието на редактора и след това да се извърши промяната. При redo() директно се извършва промяната. В случай на AddOperation това означава добавяне към списъка.

```
   public IStatus redo(IProgressMonitor monitor,
IAdaptable info)
          throws ExecutionException {
       fEditor.getContent().add(fAddedValue);
       fEditor.getViewer().refresh();
       fEditor.setDirty();
       return Status.OK_STATUS;
   }
```

DeleteOperation

Работата на `DeleteOperation` следва същия ход на действие.

```
public  class DeleteOperation extends AbstractOperation {

   private ListEditor fEditor;
   private String fSelectedString;

   public DeleteOperation(ListEditor editor) {
      super("Delete");
      fEditor = editor;
   }
   public IStatus execute(IProgressMonitor monitor,
IAdaptable info)throws ExecutionException {}
   public IStatus redo(IProgressMonitor monitor,
IAdaptable info)  throws ExecutionException {}
   public IStatus undo(IProgressMonitor monitor,
IAdaptable info)  throws ExecutionException {}

}
```

DeleteOperation

При изпълнението на DeleteOperation текущо избраният низ се запаметява в fSelectedString и след това се изтрива от съдържанието на редактора. При повторно изпълнение – redo – низът вече е определен и трябва само да се изтрие.

```
public IStatus execute(IProgressMonitor monitor, IAdaptable info)
        throws ExecutionException {
    IStructuredSelection selection = (IStructuredSelection) fEditor
            .getViewer().getSelection();
    fSelectedString = (String) selection.getFirstElement();
    return redo(monitor, info);
}

public IStatus redo(IProgressMonitor monitor, IAdaptable info)
        throws ExecutionException {
    fEditor.getContent().remove(fSelectedString);
    fEditor.getViewer().refresh();
    fEditor.setDirty();
    return Status.OK_STATUS;
}
```

DeleteOperation - undo

За да се отмени промяната на DeleteOperation изтритият низ трябва отново да се добави към списъка.

```
   public IStatus undo(IProgressMonitor monitor,
IAdaptable info)
          throws ExecutionException {
      fEditor.getContent().add(fSelectedString);
      fEditor.getViewer().refresh();
      fEditor.setDirty();
      return Status.OK_STATUS;
   }
```

UndoableListEditor

До момента бяха изградени само операциите за добавяне и изтриване. Все още обаче никой не ги изпълнява. За целта трябва да се променят класът на редактора `ListEditor` и класовете на действията `AddAction` и `DeleteAction`.

```
public  class UndoableListEditor extends ListEditor {

    private IUndoContext fUndoContext;
    private IOperationHistory fOperationHistory;
    private UndoActionHandler fUndoActionHandler;
    private RedoActionHandler fRedoActionHandler;

    public IUndoContext getUndoContext() {
        return fUndoContext;
    }
    public IOperationHistory getOperationHistory() {
        return fOperationHistory;
    }
}
```

UndoableListEditor

Капсулираме поддръжката на Undo/Redo от страна на класа на редактора в нов клас наречен UndoableListEditor.

```
public  class UndoableListEditor extends ListEditor {
}
```

В него ще добавиме следните полета:

- `fUndoContext` – една операция може да се изпълни в повече от един контекст. Това позволява елегантно отделяна на редактора от операцията, като дадена операция може да се изпълнява за няколко контекста.
- `IOperationHistory fOperationHistory` – списъкът от досега изпълнени операции
- `UndoActionHandler fUndoActionHandler` – обект от тип IAction който може визуално да се покаже в менюто. При изпълнението му се извиква undo на последната операция в `fOperationHistory`
- `RedoActionHandler fRedoActionHandler` – има за цел да извика redo на текущата операция в `fOperationHistory`

UndoableListEditor - init

Необходимата инициализация се извършва в метода init(). Тъй като UndoableListEditor наследява ListEditor е задължително да извикаме `super.init(site, input);`

```
   public void init(IEditorSite site, IEditorInput input)
         throws PartInitException {
      super.init(site, input);
      fUndoContext = new UndoContext();
      fOperationHistory =
OperationHistoryFactory.getOperationHistory();
   }
```

UndoableListEditor - createActions

Създаването на действията се извършва в метода `UndoableListEditor:createActions()`. За целта `ListEditor:createActions()` е деклариран като абстрактен, защитен метод, който задължително трябва да имплементираме в `UndoableListEditor`. Методът `fillContextMenu` ще добави желаните действия към менюто.

```
    protected void createActions() {
        fAddAction = new AddAction(this);
        fDeleteAction = new DeleteAction(this);
        fUndoActionHandler = new
UndoActionHandler(getSite(), fUndoContext);
        fRedoActionHandler = new
RedoActionHandler(getSite(), fUndoContext);
    }
    protected  void fillContextMenu(IMenuManager menuMgr) {
        menuMgr.add(fUndoActionHandler);
        menuMgr.add(fRedoActionHandler);
        super.fillContextMenu(menuMgr);
    }
```

AddAction

Добавянето на елемент към списъка се извършва с помощта на `AddOperation`. `AddAction` има за цел да се покаже в менюто и при избор от страна на потребителя да изпълни обект от тип `AddOperation`

```
public  class AddAction extends Action {

   private UndoableListEditor fEditor;

   public AddAction(UndoableListEditor editor) {
      super();
      setText("Add");
      fEditor = editor;
   }

   @Override
   public void run() {
   /* ... code missed ... */
   }
}
```

AddAction - run

```
@Override
public void run() {
    AddOperation operation = new AddOperation(fEditor);
    operation.addContext(fEditor.getUndoContext());
    IProgressMonitor monitor =
    fEditor.getEditorSite().getActionBars()
          .getStatusLineManager().getProgressMonitor();

    try {
       fEditor.getOperationHistory().execute(operation,
                                   monitor, null);
    } catch (ExecutionException e) {
       e.printStackTrace();
    }
}
```

Конструира се нов обект от тип `AddOperation`. Задава се контекстът, в който трябва да се изпълни операцията. Обектът се подава на `fEditor.getOperationHistor()`, за да бъде изпълнен.

DeleteAction

Аналогична е логиката на работа на DeleteAction.

```
public  class DeleteAction extends Action {
    private UndoableListEditor fEditor;
    public DeleteAction(UndoableListEditor editor) {
        super();
        setText("Delete");
        fEditor = editor;
        fEditor.getViewer().addSelectionChangedListener(
                new ISelectionChangedListener() {
                    public void selectionChanged(SelectionChangedEvent
event) {
                        if (((IStructuredSelection) fEditor.getViewer()
                                .getSelection()).isEmpty() == false) {
                            setEnabled(true);
                        } else
                            setEnabled(false);
                    }
                });
    }

    public void run() {
    /* ... code missed ... */
    }

}
```

DeleteAction - run

```
    @Override
    public void run() {
        DeleteOperation op = new DeleteOperation(fEditor);
        op.addContext(fEditor.getUndoContext());
        IProgressMonitor monitor =
                fEditor.getEditorSite().getActionBars()
                .getStatusLineManager().getProgressMonitor();
        try {
            fEditor.getOperationHistory().execute(op,
monitor,
null);
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
```

Упражнение

Проектът org.elsys.pluginsample.undoredo предоставя редактор подържаш Undo/Redo.

Задачата е използвайки този проект да се реализира Undo/Redo в org.elsys.pluginsample.

Стъпки:

- Реализация на UndoableListEditor наследяващ ListEditor
- Промяна на plugin.xml, така че асоциираният редактор да е UndoableListEditor.
- Реализация на AddOperation, DeleteOperation.
- Промяна на AddAciton, DeleteAction.

Предоставеният org.elsys.pluginsample.undoredo не работи. Съществуват два пътя на работа

- да се оправи org.elsys.pluginsample.undoredo
- да се допълни org.elsys.pluingsample

Copyright