AcademiCS Application Framework

Specifications for the Third Set of Milestones and Deliverables

The third set of milestones/deliverables is concerned with the opening of documents and quitting within the application framework.

1. Glossary

Action	A task that is executed in response to user interaction with a GUI widget (e.g., a menu item or button).
Localization	The adaptation of a product to meet the requirements of a particular locale.
Worker	A task that can be executed in a thread other than the event dispatch thread.

2. Engineering Design

The relationships between the various classes that must be implemented for the first set of milestones/deliverables is illustrated in the UML class diagram (that is available as an SVG file). In addition to the specifications in that diagram, the classes/interfaces must comply with the following specifications.

2.1 The `AbstractLocalizedAction` Class

An `AbstractLocalizedAction` is a specialized `AbstractAction` that uses the default `Locale` and a `ResourceBundle` to set its `NAME`.

It uses the `name` it is given as the `ACTION_COMMAND_KEY` and uses the following conventions for the `SMALL_ICON` and `LARGE_ICON_KEY`:

```
SMALL_ICON = "/edu/jmu/cs/academics/resources/" + name + "-16x16.png"
LARGE_ICON_KEY = "/edu/jmu/cs/academics/resources/" + name + "-32x32.png"
```

Note that, for simplicity, this class does not set the `MNEMONIC_KEY` or `ACCELERATOR_KEY` since they would also need to be translated.

The `getString()` method returns the localized version of the given `key`.

2.2 The `AbstractDocumentAction` Class

This class encapsulates actions that can be performed on documents. The parameter `D` indicates the class of the document.

The `getPropertyChangeListenerNames()` method which must be implemented by concrete specializations gets the names of all of the action's `PropertyChangeListener` objects.

The `setDocumentManager()` method must: remove all of the action's `PropertyChangeListener` objects from the current `DocumentManager` (if there is one) referred to in the `manager` attribute, update the current `DocumentManager` to be the "new" `DocumentManager` (i.e., referred to by the parameter named `dm`), and must add all of the action's `PropertyChangeListeners` to the "new" `DocumentManager`.

2.3 The `Quit` Class

The `Quit` class is an encapsulation of an action that quits/exits an application. The parameter `D` denotes the class of the document that the application processes.

It must listen to `DOCUMENT_ACTIVATED`, `DOCUMENT_CLOSED`, and `DOCUMENT_EDITED` property change events in order to enforce the appropriate work flow (since the `Quit` action must only be enabled if the document is *empty* or *unmodified*). The `propertyChange()` method must enforce this work flow.

The `document` attribute must be used to keep track of the active document (when one is activated). It must be `null` when when the active document is closed.

This class must be a `WindowListener` because it must handle `windowClosing()` messages (generated by the user closing the window). If the user attempts to close the window when the action is disabled, the user must be presented with a message dialog containing the `WARNING_QUIT` message.

The `actionPerformed()` method, which contains the logic for the action, must quit/exit the application.

Note that this class must be localized because it presents information, directly or indirectly, to the user.

2.4 The `FileTypeFilter` Class

A `FileTypeFilter` can be used to filter files on the basis of on or more file types (i.e., the suffix after the last dot). For convenience, it implements both common `FileFilter` interfaces. It must provide the ability to accept directories so that the user of a file dialog can navigate through the file system using such a filter. This capability must be provided by default.

The `addAcceptableType()` method must only add the type to the collection of acceptable types if it is not `null`, not empty, and not already in the collection.

The `accept()` method must return:

- `false` if the `File` is `null`;
- `true` if the filter should accept directories and the `File` is a directory;
- `true` if the name of the `File` ends in a `.` followed by any of the acceptable suffixes;
- `false` otherwise.

The `getLastAccepted()` method must return the last type that was accepted in the most recent call to `accept()` that returned `true`. If the last call to `accept()` returned `false`, then `getLastAccepted()` must return `null`.

The `size()` method must return the number of currently acceptable types.

The `createFilterGroup()` method must create a collection of from an array of file types. The last element in the collection is a filter that accepts all of the file types. The other elements in the collection accept a single file type (and are sorted by type). The description of the last filter must be the given description. The description of the other filters must be the file type.

Note that this class uses a `Vector`(rather than, say an `ArrayList`) for the `acceptableTypes` attribute because the Vector class provides thread safety. While the required thread safety can be provided in other ways, this is the simplest way to provide it.

2.5 The `AbstractEditableReadingWorker` Class

An `AbstractEditableReadingWorker` is a worker that reads an `Editable`. The parameter `P` denotes the class of the `Product`, the parameter `F` denotes the class of the factory that knows how to produce `P`, and the parameter `S` denotes the class of the source used by the factory (or `Void`).

The `done()` method will be classed in the event dispatch thread when the task is "completed". Hence, if the task was not cancelled, it must assign the result of calling `get()` to the attribute named `result`. If this call results in an exception being thrown, it must display an error dialog containing the localized `String` for `ERROR_IO_OPEN`.

The `doInCallersThread()` method must call the `readInCallersThread()` method.

Note that this class must be localized (since it will, directly or indirectly, present information to the user).

2.6 The `StringReadingWorker` Class

A `StringReadingWorker` is a worker that reads a `String` representation of a document. The parameter `P` denotes the class of the `Product`, and the parameter `F` denotes the class of the factory that knows how to produce `P`.

The `readInCallersThread()` method must read all of the (assumed to be UTF-8) characters from the attribute named `file` into a `String` and then use the factory to create the product.

The `doInBackground()` method must call the `readInCallersThread()` method.

2.7 The `AbstractModalDialog` Class

The `AbstractModalDialog` class is an abstract modal `JDialog` that can be specialized to create dialogs for specific purposes. It must be possible to include either a cancel button, or an OK button, both, or neither.

The `createMainPanel()` method will be implemented by concrete specializations. In those specializations it will create the `JComponent` that constitutes the main panel of the dialog.

The `showDialog()` method must set the owning object's location (centered on the owner) and then make the owning object visible. It must return `JOptionPane.CANCEL_OPTION` if the user clicks on the cancel button or closes the dialog. Otherwise it must return `JOptionPane.OK_OPTION`.

It must be an `ActionListener` because it must respond to `ActionEvent` objects generated by the user clicking on the cancel button or OK button. It must be a `WindowListener` because it must respond to `WindowEvent` objects generated by the user closing the window.

Note that this class must be localized (since it will present information to the user).

2.8 The `BackgroundTaskDialog` Class

A `BackgroundTaskDialog` is a modal `JDialog` that can be used to cancel and show the status of a background task (i.e., a task being executed using a `SwingWorker`). The parameter `T` denotes the result type returned by a `SwingWorker` object's `doInBackground()` and `get()` methods. The parameter `V` denotes the type used for carrying out intermediate results by a `SwingWorker` object's `publish()` and `process()` methods.

Visually, it must contain a `JProgressBar` and a cancel button. The `JProgressBar` may have its `indeterminate` attribute set, since it will be difficult to make accurate estimates of how long tasks will take and how complete they are.

It's internal `TimeoutTask` class is used to cancel the task if too much time has elapsed.

It's `execute()` method must execute the background task, start the countdown timer (if the `timeout` attribute is greater than 0), and show itself (unless the task has already completed). After the dialog is made invisible this method must cancel the countdown timer. If the dialog became invisible because the cancel button was pressed, it must also cancel the task

This class must implement the `PropertyChangeListener` interface because it must observe the task. Specifically, the `propertyChange()` method must determine if the task has completed and, if so, must cancel the countdown timer and make the dialog invisible.

2.9 The `AbstractOpen` Class

An `AbstractOpen` object is an action that reads reads a representation of a document from a `File` (in a background/worker thread), after prompting the user to select the `File` using a `JFileChooser`. The parameter `D` denotes the class of the document to be read/created, the parameter `F` denotes the class of the factory that knows how to create `D`, and the parameter `S` denotes the class of the source used by the factory (or `Void`).

It must listen to `DOCUMENT_ACTIVATED`, `DOCUMENT_CLOSED`, and `DOCUMENT_EDITED` property change events in order to enforce the appropriate work flow (since the `AbstractOpen` action must only be enabled if the document is *empty* or *unmodified*. The `propertyChange()` method must enforce this work flow.

The `Configuration` attribute named `configuration` must be used to determine the current directory/folder (using the key `"Current_Directory"`) and it must be updated whenever it changes (e.g.,as a result of the user selecting a `File` with a file dialog).

The `actionPerformed()` method, which contains the logic for the action, must display a file dialog and handle the user's response. If the user enters a `File` that does not exist, it must display a message dialog that displays `ERROR_IO_OPEN`. Otherwise, it must invoke `readUsingWorker()` for the `File`.

The `readUsingWorker()` method must create an appropriate reading worker (using the `createReader()` method), create a `BackgroundTaskDialog` that uses that reading worker, execute the dialog, dispose of the dialog, record the result of the reading worker, and inform the `DocumentManager` that a document has been created.

The `readUsingCallersThread()` method must create an appropriate reading worker (using the `createReader()` method) and invoke its doInCallersThread() method.

The `setFileTypes()` method must add "choosable" filters to the `JFileChooser`. It must add one filter for each file type in the array and one filter that accepts all of the file types in the array. The filter that accepts all of the types must be the default. Note that the `FileTypeFilter` class has a factory method that will be useful in this regard.

2.10 The `OpenString` Class

An `OpenString` object is an action that reads reads a `String` representation of a document from a `File` (in a background/worker thread), after prompting the user to select the `File` using a `JFileChooser`. The parameter `D` denotes the class of the document to be read/created and the parameter `F` denotes the class of the factory that knows how to create `D`.

The `createReader()` method simply returns a `StringReadingWorker` for the given `file` and `factory`, using the owning object's parent (obtained using its `getParent()` method) as the parent component for the `StringReadingWorker`.