Action Manager 1.0 Component Specification

1. Design

The Action Manager component provides a general framework for executing actions. It also provides the undo/redo actions framework. The component provides the one place to be accessed when executing actions inside an application. All the GUI significant changes that affect the model of the application must be executed through this component. It will keep track of the undo / redo actions, also resetting the undo / redo manager if a non undoable action is executed, or leaving the undo / redo manager's state as is if a transient action is executed.

The heart of the component is the ActionManager class and is responsible for executing simple actions, transient actions, undoable actions and undo / redoes actions. The manager will maintain the undo / redo state (the undoable actions) in a javax.swing.undo.UndoManager instance. Since for getting the UndoableActions to be un-done/re-done are not directly exposed by the UndoManager, the design uses an extension of the javax.swing.undo.UndoManager class called AdvancedUndoManager defined in this component which provides methods for achieving this functionality by using the protected methods/members from the javax.swing.undo.UndoManager class.

This class also defines the Action interface which will be implemented by the specific components of the TC UML Tool which provides the handling for the specific actions which includes both GUI as well as non-GUI actions. All the actions will be executed using the ActionManager class which provides additional service of maintaining the actions can be re-done/un-done and allowing undo/redo of Action(s).

Additionally the component defines two marker interfaces namely TransientAction and UndoableAction. TransientAction marker interface defines that the execution of the action should not affect the state of the actions that are stored to be un-done/re-done. Undoable marker interface defines that the particular action can be un-done after execution re-done if after execution it was undone.

Finally the component also provides a couple of adapter classes namely TransientUndoableAction and CompondUndoableAction which adapts TransientAction and javax.swing.undo.CompondEdit respectively as an UndoableAction.

In addition to the core set of functionalities required by the component, the component also provides additional functionality of Logging which is pretty much useful and pertinent to such a component which is central component for handling the execution of the actions and handling the undo/redo operations. Logging and tracing the sequence of events that go on is pretty much useful and helps in detecting and correction of problem easily.

Note: It is highly recommended to go through the configuration file "Action_Manager.xml" before proceeding with the remainder of the document. This file is richly documented and defines what each property is meant for and its usage in the design. [Though only 3 configuration properties are defined in it]

Please configure the required components like Configuration Manager 2.1.5, BaseException 1.0 and Logging Wrapper 1.2.

1.1 Design Patterns

Adapter pattern is used by TransientUndoableAction and CompondUndoableAction to adapt TransientAction and javax.swing.undo.CompondEdit respectively as an UndoableAction.

1.2 Industry Standards

Java Swing Undo Framework

1.3 Required Algorithms

None. All the algorithms required for the various methods are pretty trivial involving just 2 to 3 steps and hence have been provided as implementation hints in the methods documentation wherever required. This section mentions about how the logging should be performed by ActionManager using the Logging Wrapper 1.2 component.

1.3.1 Logging:

The class ActionManager can be initialized by providing the Log instance from the Logging Wrapper component to be used for performing the Logging though this is optional as Logging involves some overhead. The Log instance to be used and the level of logging to be performed can be passed as parameter to the constructor during the creation of the instance of ActionManager or can be specified in the configuration file and the namespace from which the configuration properties needs to be read can be passed as parameter to the constructor during the initialization of ActionManager. The different levels of logging supported by the component and what needs to be logged at each level have been described as follows:

Five log levels are defined.

OFF:

No logging needs to be done.

ERROR:

1. Any exception that occurs during processing.
The log message should be "Exception caught: exception.getMessage()". The exception stacktrace should also be logged.

WARN:

In addition to the logging mentioned to be done for the ERROR level, logging should be done mentioning whether the processing will proceed after the exception is raised or will the processing be terminated after the exception is raised.

INFO:

In addition to the logging mentioned to be done for the WARN level, the following additional logging needs to be done:

1. Incoming Parameters
If the method takes parameters, all the parameters should be logged. If any argument is a List, iterate through the parameters list, and log each parameter in the list. The log message should be "Incoming parameter: parameter.toString()"
2. Some major steps in the processing being started or completed.

DEBUG:

```
In addition to the logging mentioned to be done for the INFO level, the following
additional logging needs to be done:

1. The entrance and exit of each method call.
For entrance, the log message should be "entrance of method: method signature"
For exit, the log message should be "exit of method: method signature".

2. Detailed return message
If the method has return type, assume that the returned object is result, the
log message should be "Return message: result.toString()".If the method return
void, the log message should be "Return message: void".  If the return type
is an array, then the message should be "Return message: Array: <class> Size:
<size>".
```

1.4 Component Class Overview

ActionManager [class]

This class is the main class of the component and is responsible for responsible for executing all types of Action's like simple actions, transient actions, undoable actions and undo / redo actions. The manager will maintain the undo / redo state (the undoable actions) in an AdvancedUndoManager extension of javax.swing.undo.UndoManager. It provides methods to:

1. Execute Action.

2. To redo / undo Actions that exists in the undo / redo manager [AdvancedUndoManager].

3. To query about the Actions to be undone / redone.

Thread-Safety: This class has immutable instance members but is not thread-safe as it provides methods for executing the Action and also methods for undo and redo of UndoableAction and the implementations of the Actions and the UndoableAction interface might or might not be thread-safe and since this class invokes methods of these interfaces, cannot be attributed to be thread-safe.

Action [interface]

This is the main interface defined by the component and defines the behavior of an action that can be executed to perform an operation. [The Action may be GUI based or non GUI based (for example: an action to perform printing)]. This interface defines a single method called "execute" which is supposed to perform the Action when this method is invoked. The Action is required as much as possible to be transactional in nature [though might not be always possible] and when the Action fails to execute should try to return to the state before the Action was executed. The execute() method of the Action is required to throw an ActionExecutionException if it fails to execute the Action.

Examples of Action: load a new project, close the existing project.

Thread-Safety: The implementations of this interface may or may not be thread-safe. This is because the Action implementation may make use of external resources and access to these cannot be made thread-safe as these external resources might be used by other process as well.

TransientAction [interface]

This interface extends the Action interface and is a marker interface and defines no methods as a part of the interface definition. This interface marks the class of Action(s) that should not affect the state of the undo/redo manager. This marker interface indicates to the ActionManager that on executing such an Action, no change should be done to the state of the AdvancedUndoManager.

Examples of TransientAction: print a diagram, copy an element to the clipboard, save the project.

Thread-Safety: This interface follows the Action interface with regards to the thread-safety requirements and the implementations of this interface may or may not be thread-safe.

UndoableAction [interface]

This interface extends the Action interface and the javax.swing.undo.UndoableEdit interface and is a marker interface and defines no methods as a part of the interface definition. This interface marks the class of Action(s) that can be un-done and re-done. When such an Action is executed using the ActionManager, the ActionManager will be add these Actions to the AdvancedUndoManager it maintains and will use it for un-doing/re-doing such Actions.

The undo() and the redo() methods from the javax.swing.undo.UndoableEdit interface which this interface extends from are not supposed to throw any exceptions as opposed to the runtime exceptions thrown by the undo() and redo() methods of the javax.swing.undo.UndoableEdit interface. The undo() and redo() methods will not throw any exception, according to their signatures, but the concrete classes should provide error handling either through logging, or by storing the last exception or both.

Examples of UndoableActions: add a class to the diagram, move an element on the diagram, remove an element.

Thread-Safety: This interface follows the Action interface with regards to the thread-safety requirements and the implementations of this interface may or may not be thread-safe.

TransientUndoableAction [class]

This class adapts a TransientAction, to be used as a non-significant UndoableAction. Its purpose is to allow the TransientAction to be used inside a CompoundUndoableAction. The adapter's canUndo() method will return true, and the undo() method will do nothing (it will not throw exceptions). This class follows the Adapter pattern.

Example of usage: the cut undoable action is composed by a transient copy action and an undoable remove action. In order to be able to create a single undoable action for the cut action, and avoid creating a new class, the action will be a compound undoable action, containing the non-significant undoable copy action and the undoable remove action.

Thread-Safety: This class has mutable instance members and no synchronized access is made to these and hence the class is not thread-safe. Also it provides methods for executing the TransientAction it is adapting and the TransientAction might not be thread-safe.

CompoundUndoableAction [class]

This class adapts a javax.swing.undo.CompoundEdit to be used as an UndoableAction. Its purpose is to allow the javax.swing.undo.CompoundEdit to be used as an UndoableAction. The adapter's method merely delegates calls to the instance of

javax.swing.undo.CompoundEdit it is adapting. This class follows the Adapter pattern.

Thread-Safety: This class has mutable instance members and no synchronized access is made to these and hence the class is not thread-safe. Also it delegates the methods to the corresponding methods on the CompoundEdit instance it is enclosing and since the class CompoundEdit is not thread-safe, this class is also not thread-safe.

AdvancedUndoManager [private inner class]

This class extends the javax.swing.undo.UndoManager class and provides additional methods which are required for servicing the functional requirements required by the ActionManager like fetching the List of UndoableAction(s) that can be un-done/re-done, the last UndoableAction that can be un-done/re-done and the methods that allow un-doing/re-doing all the UndoableAction(s) it maintains till/including the passed UndoableAction. These methods are as such not exposed/provided by the javax.swing.undo.UndoManager class and this class provides these functionalities by making use of the protected members and the protected methods from the parent class.

Thread-Safety: This class has no members of its own but accesses the members from the super class and allows the undo/redo operations of the UndoableAction and these operations are not as such required to be thread-safe and hence this class is not thread-safe.

AdvancedCompoundEdit [private inner class]

This class extends the javax.swing.undo.CompoundEdit class and provides an additional method which is required for servicing the functional requirements required by the CompoundUndoableAction like getting the array of UndoableAction/UndoableEdit that is stored inside a CompoundEdit and which is not exposed by the CompoundEdit class as such. This method is not as such exposed/provided by the javax.swing.undo.UndoCompound class and this class provides this functionality by making use of the protected members from the parent class.

Thread-Safety: This class has no members of its own but accesses the members from the super class which is not thread-safe and hence this class is not thread-safe.

1.5 Component Exception Definitions

IllegalArgumentException [From java.lang]

This exception is thrown in various methods if the given argument is null. It is also thrown in cases when the passed String parameters to a method are empty Strings. Refer to the documentation in Poseidon for more details.

NOTE: Empty string means string of zero length or string full of white spaces.

ActionExecutionException [custom]

This exception is thrown by the execute() method of the Action interface and re-thrown by the executeAction() method of the ActionManager class. This exception primarily indicates that there was a problem in the execution of the Action and that the execution of the Action could not be completed. The exception is also thrown by the Action, if the Action determines that it cannot be executed at that point in time [may be not having access to the required resources] or its execution could lead to problems.

UndoActionExecutionException [custom]

This exception extends the ActionExecutionException and is thrown by the undoActions() method of AdvancedUndoManager to indicate that the undo of the UndoableAction's till the passed UndoableAction could not be successfully completed. This could happen when the UndoableAction till which the UndoableAction's have to be un-done is not found to be present with the AdvancedUndoManager.

Note that the undo() method of the UndoableAction is not supposed to throw any exception [even runtime exception] and hence this will "not" act as a wrapper exception as the undo() method of UndoableAction does not throw any exception. Also it is worthwhile to note the undoActions() methods of ActionManager will "not" throw this exception as the undoActions() method of ActionManager do not throw any checked exceptions.

RedoActionExecutionException [custom]

This exception extends the ActionExecutionException and is thrown by the redoActions() method of AdvancedUndoManager to indicate that the redo of the UndoableAction's till the passed UndoableAction could not be successfully completed. This could happen when the UndoableAction till which the UndoableAction's have to be re-done is not found to be present with the AdvancedUndoManager.

Note that the undo() method of the UndoableAction is not supposed to throw any exception [even runtime exception] and hence this will "not" act as a wrapper exception as the redo() method of UndoableAction does not throw any exception. Also it is worthwhile to note the redoActions() methods of ActionManager will "not" throw this exception as the redoActions() method of ActionManager do not throw any checked exceptions.

ActionManagerConfigurationException [custom]

This exception is thrown from the constructor of ActionManager when it is initialized with a configuration namespace to read the required configuration properties from the configuration file and the required properties are either found to be missing or found to contain invalid values. It also acts as a wrapper exception over the exception thrown from the Configuration Manager component.

Basically this exception indicates that the initialization of ActionManager is not possible because of problems in the configuration file [missing properties or properties with invalid values] or the configuration of the Configuration Manager component.

1.6 Thread Safety

This component is not required to be thread-safe and is not thread-safe. The interfaces defined by this component like the Action, TransientAction and UndoableAction provide the flexibility to its implementation to be either thread-safe or not.

The ActionManager class has immutable instance members but cannot be called thread-safe since it provides methods for executing the Action and also methods for undo and redo of UndoableAction and the implementations of the Actions and the UndoableAction interface might or might not be thread-safe and since this class invokes methods of these interfaces, cannot be attributed to be thread-safe.

The adapter class of TransientUndoableAction has mutable instance members and no synchronized access is made to these and hence the class is not thread-safe. Also it

provides methods for executing the TransientAction it is adapting and the TransientAction instance might not be thread-safe. Similarly the adapter class of CompoundUndoableAction has mutable instance members and no synchronized access is made to these and hence the class is not thread-safe. Also it delegates the methods to the corresponding methods on the CompoundEdit instance it is enclosing and since the class CompoundEdit is not thread-safe, this class is also not thread-safe.

The AdvancedUndoManager class has no members of its own but accesses the members from the super class and allows the undo/redo operations of the UndoableAction and these operations are not as such required to be thread-safe and hence this class is not thread-safe.

Finally the AdvancedCompoundEdit class has no members of its own but accesses the members from the super class which is not thread-safe and hence this class is not thread-safe.

Hence on the whole the component is not thread-safe. The component can be used in a thread-safe manner by firstly making sure that the Action implementations that are passed/used with this component are thread-safe and accessing the methods of ActionManager, TransientUndoableAction and CompoundUndoableAction in synchronized blocks with the synchronization done on the instance on which the methods are being called [basically "this" instance].

2. Environment Requirements

2.1 Environment

- Development language: Java1.5
- Compile target: Java1.5

2.2 TopCoder Software Components

Configuration Manager 2.1.5

It will be used in ActionManager to load configuration values from configuration file.

Base Exception 1.0

The exception ActionExecutionException and ActionManagerConfigurationException defined by this component extends from BaseException from the BaseException 1.0 component. All other exceptions defined by this component extend from ActionExecutionException.

Logging Wrapper 1.2

It is used for performing logging and tracing as mentioned in section 1.3.1.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.actionmanager [Main package of the component. No other sub-packages are required for this components design]

3.1 Configuration Parameters

No	Property	Description
1	Name MaximumUndoableActions Required true Configuration Namespace com.topcoder.util.actionmanager	This property is a required property and a single-valued property and specifies the maximum number of UndoableAction(s) that should be maintained in the UndoManager [AdvancedUndoManager] instance for performing the Undo-Redo operations, and once the limit is crossed, the UndoManager will flush out the old entries to maintain the size of the entries in the UndoManager. The value has to be an integer value greater than 0.
2	Name LoggerName Required false Configuration Namespace com.topcoder.util.actionmanager	This property is an optional property and a single-valued property and specifies the name of the instance of the Log from the Logging Wrapper 1.2 component to be used for performing the logging.
3	Name LoggingLevel Required false Configuration Namespace com.topcoder.util.actionmanager	This is an optional property and a single valued property. If the value is not specified or an invalid value is specified then it defaults to the value of "DEBUG". This property can take the value "OFF", "INFO", "WARN", "DEBUG" or "ERROR" and specifies the level of logging to be done by the ActionManager. Note, if other value is specified, then it will result in an exception.

The configuration file "Action_Manager.xml" defines the configuration required for this component. This file is provided in the "conf" and in the "docs" folder of the distribution and is quite well documentation with respect to the properties and the value it can take and what is the need for the property. This sample configuration file has been provided below:

```
<?xml version="1.0" ?>

<!--

This is the configuration file for this component and defines the set of configuration
properties for the component. These configuration properties define the information
like maximum number of UndoableAction(s) that should be maintained in the UndoManager
[AdvancedUndoManager] instance for performing the Undo-Redo operations, and once the
limit is crossed, the UndoManager will flush out the old entries to maintain the size
of the entries in the UndoManager.
```

```
It also contains the configuration properties that define the instance name of the
Log to be used for performing the logging using the Logging Wrapper 1.2 component.

Note: This is a sample configuration file. It does not contain valid configuration
data and cannot be used for testing. It's purpose is to be used as a reference for
writing other config files.

-->

<CMConfig>

  <Config name="com.topcoder.util.actionmanager">

<!--

This property is a required property and a single-valued property and specifies the
maximum number of UndoableAction(s) that should be maintained in the UndoManager
[AdvancedUndoManager] instance for performing the Undo-Redo operations, and once the
limit is crossed, the UndoManager will flush out the old entries to maintain the size
of the entries in the UndoManager. The value has to be an integer value greater than
0.

-->

    <Property name="MaximumUndoableActions">

      <Value> 10 </Value>

    </Property>

<!--

This property is an optional property and a single-valued property and specifies the
name of the instance of the Log from the Logging Wrapper 1.2 component to be used
for performing the logging.

-->

    <Property name="LoggerName">

      <Value>DefaultLog</Value>

    </Property>

<!--

This is an optional property and a single valued property. If the value is not specified
or an invalid value is specified then it defaults to the value of "DEBUG". This property
can take the value "OFF", "WARN", "DEBUG" or "ERROR" and specifies the level of logging
to be done by the ActionManager.

-->

    <Property name="LoggingLevel">

      <Value>DEBUG</Value>

    </Property>

  </Config>

</CMConfig>
```

3.3 Dependencies Configuration

- Please review the components "Configuration Manager 2.1.5", "Base Exception 1.0", and "Logging Wrapper 1.2" to make it work.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

- The required components are configured correctly and are available at runtime.

4.3 Demo

Note: The handling of the exceptions has not been shown in the demo.

The demo is shown from the Customer usage of the TC UML Tool and how the TC UML Tool will be using to this component corresponding to user interaction with the TC UML Tool.

1. Starting of the TC UML Tool:

On the starting of the TC UML Tool, the TC UML Tool will need to create an instance of ActionManager and cache it further usage. The creation of an instance of ActionManager can be done in any one of the following ways:

```
// ActionManager with no logging to be done and with max UndoableAction limit set to 10
a. ActionManager manager = new ActionManager();

// ActionManager initialized from the configuration values specified in the passed namespace
// in the configuration file.
b. ActionManager manager = new ActionManager( "com.topcoder.util.actionmanager" );

// ActionManager created by explicitly specifying the max UndoableAction limit to 50 and
// specifying the Log instance to be used and the Level of logging to be performed to ERROR.
c. Log logger = LogFactory.getInstance().getLog("MyLog");
      ActionManager manager = new ActionManager( 50, logger, Level.ERROR );
```

2. Execution of Actions corresponding to the users usage of the TC UML Tool:

Consider the user of the TC UML component does the following set of operations:
1. Opens an existing Project.
2. Adds some Classes in the model and attributes to these classes.
3. Deletes a set of classes together.
4. Moves the set of the Classes from one location to another.
5. Prints the Class Diagram of the Project.
6. Save the project.
7. Closes this project and opens a new project.
8. Adds some classes in the project.
9. Reverts the action he has performed.

Corresponding to each of these steps the different types of actions that will be created and what happens when these actions are executed is mentioned as follows:

a. Execution of Simple Action:

```
// A Simple Action will be created for Steps 1, 7 i.e. Opening of a Project and Closing of a
// Project. Creation of such an Action and execution of the Action is shown below.
// The name of the Action used here is for demo purpose and assumes a no-argument
// constructor to be present and the actual Action names from the other TC UML component
will
// be different.

// Creating the Action for Opening a Project.
// Assuming a no-argument constructor for this demo. The actual Action will be taking
// parameters, but not known at this time.
Action openProject = new OpenProjectAction();

// Executing the "openProject" Action. This should clear the state of the
AdvancedUndoManager
```

```
manager.executeAction( openProject );

// Creating the Action for Closing a Project.
// Assuming a no-argument constructor for this demo. The actual Action will be taking
// parameters, but not known at this time.
Action closeProject = new CloseProjectAction();

// Executing the "closeProject" Action. This should clear the state of the
AdvancedUndoManager

manager.executeAction( closeProject );
```

Note: "openProject" and "closeProject" should not be instance of TransientAction or UndoableAction. (Basically the instanceof operator for these Actions against TransientAction and UndoableAction should return false).

b. Execution of Transient Action:

```
// A Transient Action will be created for Steps 5 and 6 i.e. Printing a class diagram and Saving
a
// Project.

// Creating the Action for Printing a class diagram.
// Assuming a no-argument constructor for this demo. The actual Action will be taking
// parameters, but not known at this time.

Action printAction = new PrintClassDiagramAction();

// Creating the Action for saving a project.
// Assuming a no-argument constructor for this demo. The actual Action will be taking
// parameters, but not known at this time.

Action saveAction = new SaveProjectAction();

// (printAction instanceof TransientAction) and (saveAction instanceof TransientAction) should
// return true.

// Executing these TransientAction. These should not change the state of the
// AdavancedUndoManager within the ActionManager.

manager.executeAction( printAction );
manager.executeAction( saveAction );

// Creation of the Undoable wrapper for these TransientActions.

UndoableAction undoableWrapperPrintAction = new TransientUndoableAction( printAction,
"PrintAction", null); // Created without specifying the Log instance.
UndoableAction undoableWrapperSaveAction = new TransientUndoableAction( saveAction,
"SaveAction", logger); // Created with the Log instance obtained in step 1.c.

// The calls to the undo() and redo() methods will be not do anything. Basically the following
// calls will not do anything.

undoableWrapperPrintAction.undo();
```

```
undoableWrapperSaveAction.redo();

// However the following calls will affect the state of the AdvancedUndoManager within the
// ActionManager.

manager.executeAction( undoableWrapperPrintAction );
manager.executeAction( undoableWrapperSaveAction );

// Getting the last exception if any had occured in the execution of TransientUndoableAction.
ActionExecutionException exception =
((TransientUndoableAction)undoableWrapperPrintAction).getLastException();
```

c. Execution of UndoableAction:

```
// An UndoableAction will be created for steps 2,3,4 and 8 mentioned above. Note that all the
// Action instances created here shall return true when they are checked to be instance of
// UndoableAction and should return false when checked to be instance of TransientAction.

// Creation of an UndoableAction for the Adding a class to the diagram.
// Assuming a no-argument constructor for this demo. The actual Action will be taking
// parameters, but not known at this time.

UndoableAction addClassAction = new AddClassDiagramAction();

// Creation of an UndoableAction for the Adding a members to the class.
// Assuming a no-argument constructor for this demo. The actual Action will be taking
// parameters, but not known at this time.

UndoableAction addClassMemberAction = new AddClassMemberAction();

// Executing these Actions. Should change the state of the AdvancedUndoManager when
these
// when these actions are executed.

manager.executeAction( addClassAction );
manager.executeAction( addClassMemberAction );

// Creating UndoableActions for deleting a set of classes and adding them to a
// CompoundUndoableAction so that these are executed as a unit and can be
un-done/re-done
// as a unit.

UndoableAction compoundAction = new CompoundUndoableAction();

// Create a DeleteClassAction [not from this component, but a similar action (may be with
// different name) will be present in the other component of the TC UML Tool, for each of the
// class to be deleted. [Only one shown here, others to be done similarly]

UndoableAction deleteClassAction = new DeleteClassAction();
compoundAction.addEdit( deleteClassAction );

// Execute the compoundAction. All actions inside it will be executed and the state of the
// AdvancedUndoManager will be changed.

manager.executeAction( compoundAction );
```

```
// Getting the last Action that can be undone.
// The "action" instance returned should be "compoundAction" instance.
UndoableAction action = manager.getUndoableActionToBeUndone();

// Getting the LastException that might have occured in executing the "compoundException".

Exception exception = ((CompoundUndoableAction)compoundAction).getLastException();

// Performing the undo of the UndoableActions till "compoundAction". In this case, since
// "compoundAction" is the last one and so only the "compoundAction" will be undone.

manager.undoActions( compoundAction );

// Getting the last Action that can be redone.
// The "action" instance returned should be "compoundAction" instance.
UndoableAction action = manager.getUndoableActionToBeRedone();

// Performing the undo of the UndoableActions till "compoundAction". In this case, since
// "compoundAction" is the last one and so only the "compoundAction" will be undone.

manager.redoActions( compoundAction );

// Getting the List of UndoableAction(s) that can be Undone.

List undoActions = manager.getUndoableActionsToBeUndone();

// Getting the List of UndoableAction(s) that can be Undone.

List redoActions = manager.getUndoableActionsToBeRedone();
```

3. Getting the Presentation Names for the UndoableActions:

```
  // Getting the Presentation names for the UndoableActions. Using the
// "undoableWrapperPrintAction" created in step 2.b for this.

String presentationName = undoableWrapperPrintAction.getPresentationName();
String undoPresentationName = undoableWrapperPrintAction.getUndoPresentationName();
String redoPresentationName = undoableWrapperPrintAction.getRedoPresentationName();
```

4. Marking an UndoableAction not to be used further:

```
  // Using the "undoableWrapperPrintAction" created in step 2.b for this.

undoableWrapperPrintAction.die();

// After this call the following call will thrown an ActionExecutionException.
manager.executeAction( undoableWrapperPrintAction );
```

5. Future Enhancements

None at this moment.