TEXTEDITOR REPORT

ACO class by Noel Plouzeau, Fall Semester 2021

Authors: Johann TOUTA | Niklas BARTH
johann.touta@etudiant.univ-rennes1.fr | niklas.barth@etudiant.univ-rennes1.fr
Submitted on: 20.12.2021
Supervisor: Adrien LEROCH | adrien.leroch@univ-rennes1.fr

This document serves as a software project report for the ACO text editor project carried out by the above mentioned authors.

Table of Contents

Launching different versions

Version	GUI?	Run the following command
V1	no	<git checkout V1>
V2	no	<git checkout V2>
V3	yes	<git checkout V3>

Note: Versions come with tests

- ★ **The master branch contains all mandatory informative documents (Report, UML, Javadoc). The code is contained in the three branches mentioned above**.
- ★ **The V2 tests have been updated in the V3**.

Architecture

The following presents the project architecture. Further, important design decisions are explained.

UML class diagram and Javadoc

- ★ **The UML diagram is directly provided at the root of both master and V3 branches, to avoid resizing entailing a quality loss**.
- ★ **The Javadoc is also provided at the root of both master and V3 branches**.

Design Decisions

What	Comments
General Command	A selection [start:end] includes *start* and excludes *end*. When defining a selection and running a command accordingly, the selection remains in its state. So if a selection [0; 3] is made on “Hello world” and the text cutted, the new selected text is “lo world”. **The same holds true after applying other functions that alter the buffer**.
V1 - Command design pattern choice	In the command design pattern context, the method arguments are stored in the invoker. Thus, only one concret command instance is required for each ConcretCommand class.
V2 – Memento Design decision	In the Memento design pattern context, each concret command is an originator. Therefore, the interface “Originator is added” to enable a concret command to be both a command and an originator. The resulting class is named “CommandGlobal” and extends both “Command” and “Originator” interfaces.
V2 – Replay functionality	Recording text writing without changing the selection text implies to run the replay function at the current cursor position. Therefore, when replaying, the result in the GUI is reversed because the successive insertions are always made in the same place. This behavior is not a malfunction, it’s expected given the code. Opening: To enhance the general text editor behavior, it could be considered to include an updating of both begin and end selection indexes while playing a command.
V2 – Replay decisions	Deep copies of both command and state lists are made, because while replaying, the successive executed commands and states are added to those lists, which results in an infinite loop. Then, the last commands and states are removed, the buffer erased and the commands replayed from the beginning.
V3 – Undo functionality	Used method: Saving every command and command state to create a history. When undo is executed the last command is removed from the history and the history is replayed from the beginning after erasing the buffer.
V3-Redo decisions	In the undo operation implementation, both command and state to be removed are added to two other lists in anticipation of a future redo. The redo process simply consists in executing the last command saved inside the “futureCommands” list, and removing it when executed.

GUI	**Not any JFrame functionality is used**: ◦ Writing in the TextArea field is made through the text editor functionalities. **The text area content is replaced by the buffer content automatically at any time**. ◦ Removing text (backspace) is not made through the JFrame removing function. By pressing the backspace touch, the DeleteCommand (Command design pattern) is used, and the text area content is replaced with the buffer content. ◦ Selection, Copy paste and cut: In order to demonstrate the text editor abilities, we make available dedicated functions to avoid using the JFame pre existing ones (refer to the explanations below to use it). Choices: ◦ After each keystroke on the keyboard, the begin and end indexes are replaced at the end. ◦ When pasting an element, the begin and end indexes are automatically replaced at the end, to paste the content at the end although the selection has not been replaced at the end. ◦ As the cursor is systematically replaced at the end when writing on the keyboard, the redo function has been updated to replace the cursor at position 0, to avoid indexOutOfBoundException. ◦ Only simple punctuation special characters are accepted.
Other design choices	● Personalized exceptions have been created to accurately cover the different exception cases. ● The package fr.istic.aco.utileFunctions contains static functions to factorise the code. The functions aim to make a deep copy of ArrayList by using iterators.

Using the text editor

The following explains how to launch the GUI, and the functionalities of the text editor.

The GUI does not put forward all the text editor functionalities. The main client is not the GUI, but the JUnit tests.

Starting the GUI

1. Open the package named "GUI".
2. Run the Java file "Launcher.java".

Functionalities

Functionality	Explanation

Selection	Please, do not use the classical selection. Two spinners are provided to define the begin index (left spinner) and the end index (right spinner). Then, click on the "Select" button to confirm the selection.
Copy, Past & Cut	Please, do not use the classical shortcuts (Ctrl + c, Ctrl + x…). **Any content added to the buffer in such a way will automatically be removed from the buffer**. 1. Define a selection as explained above. 2. Press the “Select” button.
Record	1. Click on "Start recording". 2. Do some actions (writing on keyboard, defining selection, cutting…). 3. Click on "Stop recording" and then "Replay".

Issues: Pasting a text is not immediately visible, a new letter should be written before seeing the result. This behavior is not due to the text editor, but is due to the fact that we avoid all JFrame native functionalities to replace it by the text editor functionalities only.

Synthesis of the test results

The JUnit tests are executable and confirm that the required functionalities are working. At the same time the test achieve a high code coverage:

While running the coverage estimation on the entire project, the resulting percentage is worth 80%. This result is pulled down by the GUI class which is not tested to the extent that it only uses already tested frontend features.
Independently considered, all other classes have a coverage between 90% and 97%.

Element	Coverage	Covered Instructions	Missed Instructi
TextEditor	80,9 %	3 773	891
src/src	80,9 %	3 773	891
fr.istic.aco.GUI	0,0 %	0	689
fr.istic.aco.tests	94,7 %	2 920	162
fr.istic.aco.Exceptions	0,0 %	0	14
fr.istic.aco.Commands	97,9 %	333	7
fr.istic.aco.Recorder	92,7 %	76	6
fr.istic.aco.UndoRedo	97,1 %	203	6
fr.istic.aco.Memento	81,8 %	18	4
fr.istic.aco.utileFunctions	92,7 %	38	3
fr.istic.aco.editor	100,0 %	133	0
fr.istic.aco.Selection	100,0 %	52	0