COMP354 Software Engineering

Project Report:
Text Editor With Smart Undo
v1.0

Winter 2021

Group 12:
Paul Piggott, 27519451
Duc Minh Bui, 40073498
Trung Hieu Doan, 40082488
Andy Nguyen, 40042149
Thanh Tung Nguyen, 40097024
Ian Njoroge, 40079227
Hyder Wajid Hameed, 40060883

April 15, 2021

Dr. Rajagopalan Jayakumar

Contents

Schedule

Task	Description	Assignee	Deadline
Implementation of Model Module	The model module is the foundation of the system. This component must be implemented before any integration tests can be performed. This task is to implement the code of all components of the model module as per the specifications of our previously submitted reports.	Paul, Hieu	April 3
Component Tests for Model Module	Again, the model module is the foundation of the system. The purpose of this task is to implement and perform all component level tests for this module, so that it will be ready for integration testing when the implementation of other modules is finished. Component tests for the Edit and EditGroup should be performed before the component tests for the EditContainer and EditGroups components.	Paul, Hieu	April 4
Implementation of View Module	The purpose of this task is to implement the code of the components of the View module as per the specifications of our previously submitted reports.	Thanh	April 5
Implementation of Controller Module	The purpose of this task is to implement the code of the components of the Controller module as per the specifications of our previously submitted reports.	Andy, Minh, Ian	April 13
Components Test for View Module	The purpose of this task is to perform the component level tests for the module, so that it will be ready for integration testing. All component testing as described in Assignment 3 should be performed, including path testing and error testing.	Thanh, Hyder	April 13
Component Tests for Controller Module	The purpose of this task is to perform the component level tests for the module, so that it will be ready for integration testing. All component testing as described in Assignment 3 should be performed, including path testing and error testing.	Andy, Minh, Ian	April 13
Model Integration Testing	The purpose of this task is to perform “Integration Test 1”, to verify that the EditGroup and EditContainer components of the Model module work as expected. This testing must be performed, and the testing must be passed,	Paul, Hieu	April 5

	before other integration testing can go forward as planned.		
Controller - Model Integration Testing	The purpose of this task is to perform the integration testing as outlined in “Integration Test 3”.	Andy, Ian, Minh, Paul, Hieu	April 9
Controller - View Integration Testing	The purpose of this task is to perform the integration testing as outlined in “Integration Test 2”.	Ian, Any, Minh, Thanh, Hyder	April 9
Model - View - Controller Integration Testing	The purpose of this task is to perform the final integration testing as outlined in the “Integration Test 4”.	Everyone	April 10
UI / UX test usage + tweaking	The purpose of this task is for all members of the team to spend some time using and evaluating the integrated product. Criticisms, improvements and experiences should be noted. Based on the notes, final tweaks can be made to the product so that it operates as desired.	Everyone	April 12
Regression testing based on tweaks	The final task is to perform some regression tests once the tweaks noted above have been made, to ensure that everything is still perfectly functional. After tests, this will be the final product.	Everyone	April 13

Risk Analysis

Description of Risks

In this section we will outline the known and predictable risks associated with the implementation of TESU.

The first predictable risk is related to staff. The experience of the developers working the team is limited in regards to developing a GUI application in Java. Unfamiliarity with the nature of GUI's in Java could result in the development time for the Controller and View components taking longer than anticipated. This is a critical risk. The mitigation plan for this risk has been taken in the project scheduling phase. The time of work for each phase has been estimated and a deadline for completion has been set. These two modules of the system have been budgeted to allow for extra days for completion. Additionally, work on the model module will be completed before the target deadline for the other modules. This will allow the developers of the model module to aid in the development of the other modules if required.

The second predictable risk is also related to staff and the inexperience of developing a GUI in Java, but falls under the process definition category. It is possible that as a result of unfamiliarity with GUI, some parts of the architecture are lacking or have not been planned properly. This is a catastrophic risk. The mitigation plan for this risk has been taken, at least in part, in the project scheduling phase. Deadlines have been set for each part of the development process. However, the deadline for completion of the product is set to three days before its due date. These three days will act as buffer days. Any unplanned development can be scheduled and undertaken as needed. The time required for this unplanned development can result in the deadlines being pushed back for the corresponding amount of time without compromising the final release date of the application.

Under the developer environment category, there is the possibility that some of the developers are not trained on the tools we will use. Specifically, some members may be unfamiliar with the GIT version control software that will be used to control developed code. This

is a risk that has happened to some of us in the past on other Concordia projects. This is probable, but relatively inconsequential risk. The mitigation plan is any members unfamiliar with GIT will be assigned to do the introductory parts of the “Git & GitHub Complete Masterclass: Beginner to Git Expert” course of UdeMy that are available through Concordia.

Under the staff risk category, there is the possibility that a lack of communication could greatly affect the development of the project. We have segmented ourselves into groups, where each group will work on one of the model, view and controllers modules. At the end, these modules must be integrated. A lack of communication during the development of the modules could result in modules that are harder to integrate, and may have to be refactored even before testing takes place.

Risk Table

Risk	Category	Probability	Impact
Software Architecture Lacking	Process Definition	30%	1
Development time takes longer than expected	Staff Experience	70%	2
Lack of communication	Staff Risk	50%	1
Lack of synchronization between members’ availability	Staff Risk	60%	3
Medical risk	Staff Risk	20%	4
Lack of training on GIT	Development Environment	80%	5

Implementation

Mitigation of Risks

Software Architecture

The software architecture inevitably did have some design errors that was not accounted for during the design of the architecture, specifically the communication between the two modules: views and controllers. To elaborate, a user that uses an undo feature on the menu bar should be letting the undo panel controller do the undoing instead of having its own separate implementation to ensure a more robust and loosely coupled architecture. This issue was quickly recognized and placed as a high priority. Thankfully, following the MVC pattern itself saved us an immense amount of time and this was easily resolved by making sure specific controllers have a reference to the undo panel controller (which manages the undo panel view and models). We funnelled the undo implementations onto the undo panel controller so that any sort of change in the undo implementation should be within one module only. Besides that, we also had to add new private helper members and methods that were not specified within the UML diagram to make code more organized and loosely coupled. We also replaced the Undo button on the menu bar to “New Group” as we felt it makes more sense being in the same action category.

These design errors were a critical risk. They were mitigated through an agile process. The onus was placed on the software engineers who discovered these issues during development to come up with a way to fix them.

GIT Training

We unfortunately did encounter GIT issues working on this project and realistically due to inexperience. Throughout most of our development, there was only one branch ever created and a majority of the development seems to take place in the main branch. This was a mistake and was quickly bought up by the team leader to ensure everybody is on the same page when it came to working on different branches. Luckily, all members were working on the project at different time intervals so there were no conflicts. Nevertheless, this risk could potentially cause devastating consequences if this was not prevented earlier. Additionally, those that have

problems using GIT bash were suggested to use Github desktop, an app that offers a more visible and convenient way of managing the project. While this was a mistake, it was a negligible that did not have much impact on the development of the project.

Communication Risk

As the above description and table shows, this risk was a catastrophic risk. Unfortunately, no mitigation steps were taken. There ended up being a lack of communication between the teams working on the different modules. There was a bit of a lack of understanding on how the different modules would communicate and integrate. These questions were not raised until it was too late. With hindsight, an effective mitigation strategy would have been to meet regularly to allow the teams to update each other on their progress and ask for any clarification that is needed. Alas, there were only three meetings between the different teams during the entire development process. The result was a chaotic scramble close to the deadline to try to make everything fit and work together.

Unexpected Development Time

As the above description and table shows, this risk was also a catastrophic risk. And similarly to the other catastrophic risk of a lack of communication, no mitigation steps were taken for this risk, either. The model module was the only component that was completed on time. The view module was finished a day before the deadline, and the controller module was being worked on at the time of this writing (night of the deadline). In retrospect, an effective mitigation strategy would have been similar to the one mentioned above. Meeting often (once every 1 or 2 days) to allow the teams working on each module to update each other would have allowed for any lack of progress to become apparent. This way, developers who finished working on their own parts could have been reassigned to help on parts that were falling behind schedule. This was a catastrophic risk that has jeopardized our final product.

Implementation and Tests

Model

The Model classes implementation consist of the base models and their test cases. We decided to use JUnit as our testing framework.

For the base modelling, we have four main components, including Edit, EditContainer, EditGroup and GroupContainer.

The Edit component was implemented with methods to generate properties of edits. In this component, edit's IDs are created and managed, as well as making sure edits are in the right position. It has the properties nextID, id, text, position, next, previous. The nextID property is used to generate ID for the edit. The id, text, position, next and previous properties are used to construct the edit. Here, id, text, position are the id, text, position of the edit. Previous and next are used to manage positioning for the edit. They also help with checking for the existence of the next and previous edits. The split method splits the edit at its index, returning the next edit at the next ID. The method merge that takes in a String text merges the text to the Edit's text unless the text that needs to be merged does not match the given text. The mergeDeleted method combines a sequence of deleted edits.

EditGroup is implemented to manage a group of Edits created by the user. To construct this component, we include the name of the group and an array list of EditView that contains all the edits chosen by users.The component has a function to set name for the group of edits, a function to get the name of the group. It can also add a list of relevant edits to an existing group and remove a list of edits out of a group using IDs.

The EditContainer component was implemented to manage the edits. It is like an interface point to the individual Edits which manages the creation of edits, as well as process the undo commands. It has the properties first, last, size, updated, mostRecent and containerSingleton. The first and last properties are of type Edit, which represents the reference to the first and last Edits in the EditContainer. The updated property is to check whether the method markSeen is used, which is to ensure all existing groups are always up to date. The size property returns the size of the elements in the current EditContainer. The create method is used to insert an edit made by the user into the list of Edits. The sync method guarantees the right positioning of the edits every time each operation on EditContainer is performed. The asText method returns the text representation of the document in its current edited state. The delete method that takes in an ID from an Edit will find the edit with the given id and remove it, unless the edit's id was not in the container in the first place. The view method returns a corresponding list of the text representation of those edits. The undo method that does not take in any parameter will perform undo to the most recent edit by looking at its id. The undoById performs the same functionality but with a specific id given and the undoByIds method removes the list of edits with corresponding ids given in the integer list as the parameter. The method empty resets the first and last edit and sync the container.

The GroupContainer is implemented to manage user created groups of groups. It is responsible for creating and deleting individual EditGroups, as well as managing the addition and removal of edits to those groups. It has the properties groups that store all the groups of edits, a defaultGroup property that acts as a list of all the edits that are not in a user-managed group. It also has editContainer property that is a reference to the created instance of the EditContainer and containerSingleton property. The getContainer method initializes containerSingleton, which prevents more than one GroupContainer from being created. The create method takes in a string, creates a new EditGroup and assigns the string as the group's name. The removeGroup method that takes in a string as the group's name, removes the group then add the edits back to the default group, unless group name does not exist in groups. The removeAndDeleteGroup method that takes in a string as the group's name, deletes the name and all the edits of that group based in their ids, unless group name does not exist in groups. The undoGroup method that takes in a string as the group's name, undos all the edits in the group based on their ids. The add method that takes in group name and an array list of ids adds specified list of Edits to the specified EditGroup based on their ids. The add method that takes in group name and an id adds specified edit to the specified EditGroup based on its id. The undoEditsInGroup method performs undo to a specified group with the edits given using their ids, obtained in the integer list as the parameter. The undoEditsInDefaultGroup method performs undo to the default group with the edits given using their ids, obtained in the integer list as the parameter. The EditGroup is synced by removing any edits that are no longer valid. The updateDefaultGroup method is use to add more specified edits in the default group, unless they were already in there. The update method makes sure all existing groups are not out of date. The viewEditsInGroup and viewEditsInDefaultGroup are textual representation of the Edits that comprise an EditGroup.

Tests for the model component were of two types. First, JUnit tests were written for the EditContainer and Edit components. These unit tests covered the data, interface, boundary and exception handling tests that were described in Assignment 3. In total, 11 unit tests were implemented for the EditContainer component, and 4 tests were implemented for the Edit component. The second kind of testing that was performed was a hybrid integration/component test. It is partly an integration test because it combines the EditContainer and GroupContainer components of the model module. It is not a true integration test, because it does not involve components from other modules. For this test a driver was created. This driver accepts input from the user (in this case the developer who is testing it). This driver is meant to simulate the interaction that the actual users (and the controllers that perform actions based on the interaction with the software) will have with the model components. After each interaction, the

contents of the EditContainer and the GroupContainer are printed to the screen. In this way, any type of interaction can be visually debugged. So this interactive test driver was developed as both a testing tool and a way for other developers to understand how user actions were reflected in the model.

View

The implementation of the View classes were strenuous due to the lack of knowledge using Java Swing, our basis for our software user interface. Research on how to create the software user interface from pure coding and use of libraries required long research on being familiar with their API. For example, JPanel is one of the many components offered by Swing to act as a container. However, customizing the format and style of the JPanel meant needing to know the multitude of different layouts the JPanel uses. These layouts then have their own unique APIs that behave differently from other layouts. That being said, Java Swing is quite outdated and not optimal for modern visuals in comparison to the softwares we have today. Outside of learning the library, the implementation was fairly simple and easy and does not require any extensive implementation to make one UI component show up. Once the user interface was created with a well defined JFrame, all that was left was to add the event handlings to any interactive buttons, menu bar, selections in the panel. For this task, action listeners, action performed, and event handlers were implemented by using anonymous classes, as they are ideal to use in these types of cases for the UI. Some buttons were having an issue with not updating the panel once they were pressed but this was easily resolved by calling the updateUI method after each action was listened to and performed. Lastly, we decided to make two requirement changes for our software. Previously, we limited the number of edit groups to exist at once as ten and we decided to lower it down to seven instead. This change in the user interface design allowed the software to look more sleek and less convoluted for the user. The second feature we decided to remove due to time constraint is the expanding collapsing the undo panel view. However, the impact for this removal this is minimal.

In regards to the testing of the View classes, we used Unit Testing to check if the state of the user interface is correct after user actions such as adding new edit groups and deleting them. There are two components that needed testing: UndoPanelView and EditGroupView. The process of the first component involved testing the edit groups like creating new edit groups, deleting existing edit groups, expanding edit groups, and collapsing edit groups. While the second related to the edit list required testing adding new edits, undo edits, and undoing all edits. During the unit testing phase, we considered inputs such as empty strings, null values,

indexes that are out of bounds, and input types. We then check the output to ensure they match the expected value such as the correct size of a list, number of total edit groups, or the correct edit group is expanded even after deleting other groups. We were able to spot a few errors that were not considered and made the appropriate changes to consider these invalid inputs. The testing implementation and debugging went smoothly as the tests were neatly organized to be easily readable and traceable.

Controller

The Controller classes implementation consist of the base classes and their test cases. The base controller consists of the ControllerInterface and three main controllers: the UndoPanelController, the MenuBarController, and the TextBoxController.

First of all the ControllerInterface consist of 8 different actions of void type which are addNewEdit(int,String,int,boolean), addNewGroup(int), undoEdits(int,int), undoGroupEdits(int), deleteRandomEdits(int), deleteGroup(int), deleteAllGroups() and finally the updateView()

Secondly, the UndoPanelController has 3 main variables namely undoPanelView, groupContainer and editContainer. The undoPanelView variable is the object of class type UndoPanelView, groupContainer variable is the object of class type GroupContainer and editContainer variable is the object of class type EditContainer. UndoPanelController will implement the interface from the ControllerInterface. First of all the addNewEdit() consist of 4 variables which are groupIndex of type int, text of type String, editIndex of type int and isAddition of type boolean. The main purpose of this function is to add new edit to the edit container. Secondly, the addNewGroup() is a function which consists of one single variable of type int namely groupIndex. It's purpose is to add a new group to the group container. UndoEdit() is a function which has two variables of type integer: groupIndex and editIndex. It will undo the requested edit and update on the view. The undoGroupEdits consists of one variable of type integer which is the groupIndex. They will search for the index of a group of edits which is requested by the user then undo it and show it on view. Moreover, the deleteGroup() function has one integer variable represented for the group index. It will search for the index of a group which the user wants to delete and do its task. The deleteRandomEdits has one variable which is the integer type name groupIndex. Its purpose is to randomly delete any group from the group container using a random number. Lastly, for the deleteAllGroups() function, it will loop through the size of the group edit container and remove all of them.

The MenuBarController has two main variables for a MenuBarView object and a UndoPanelController object. On this object all the actions were implemented in the

ControllerInterface. It will take the request from the MenuBarView and call the actions that were implemented on the UndoPanelController object.

Last but not least the TextBoxController also implements functions from the ControllerInterface. It has 3 different variables which are TextBoxView object, GroupContainer object and an integer representing the index of edit.

The undoPanelController had three tests which assessed the component's capacity to add edit groups, add individual edit items into different groups, and the ability to undo edit items from a group. By focusing on these functions, the component's core functionality is evaluated to ensure proper use once integration will be done as well as to be certain that it will work well within the final product.

The first test - testAddNewEdit() - simulates five groups which then have a random number of edit items added to each group. To do this, a group is created then the edit items are inserted. For each insertion a statement is made declaring that the edit has been added to the specified group.

The second component test - testAddNewGroup() - ascertains that a group is created. This is done by first creating then adding the group and then ensuring that the index provided to create it matches the name of the most recently created group.

The final component test - testUndoEdit() - takes in aspects of the first two tests however it also tests that all specified edits in a group are undone. To do this, the test once again simulates five groups to be added. These groups then add a random number of edit items which are subsequently undone following the addition of all of the edit items. A message is declared upon the successful undoing of each edit item.

Component Integration

The integration between components was arguably the most challenging part of our development. Our plan was to complete the model, then the controllers, and finally the view all in order. Because of this co-independency, where ideally one has to be completed before the other, we had to be flexible on how we worked together during the development. This is mostly due to the very small time window we had to complete this project. While normally two weeks are enough, it doesn't consider the schedule of every member. As such, instead of waiting for another team to complete their part, we made sure everybody does prior research, take notes, and write pseudocode (i.e. "// todo tell controller do this here") to speed up the development time. Because we were following the MVC pattern, separation between the modules and

components were easy to manage. We followed our bottom-top integration strategy and first started with the Model and Controller modules. We ran the tests again and no errors were found. Then we connected the controllers with the view modules and it went as planned. All parts were able to communicate with each other and respective method calls were being made when appropriate. Unfortunately due to lack of communication between members, the team responsible for the text box controller was not able to complete their tasks by the deadline and we had to submit what is currently working. Overall, the integration process went smoothly for the working parts and there were only a few minor bugs.