24.04.2024

PROJECT DOCUMENT – 300 KANBAN BOARD

1. Student Information

Behram ULUKIR – 1022757

Aalto University English BSc Programme - Data Science

3rd-year student

2. General Description

The kanban board is one of the tools that can be used to implement kanban to manage work at a personal or organizational level. Kanban boards visually depict work at various stages of a process using cards to represent work items and columns to represent each stage of the process. Cards are moved from left to right to show progress and to help coordinate teams performing the work [1]. By definition, a kanban board includes text-editable cards and the user is free to add new cards and lists. Mostly, the cards contain tags to filter and group along with a deadline attached to the card to keep track of the time left. In this project, the intermediate level requires that these cards can be archived by the user and can be returned from the archive pile as well. Also, the cards should be able to be moved to other lists by drag and drop motion performed by the users. Also, it requires that a single user can have multiple boards to manage. The boards should be saved to external files that are imported to recover the last status of the application in case the user launches the application.

3. User Interface

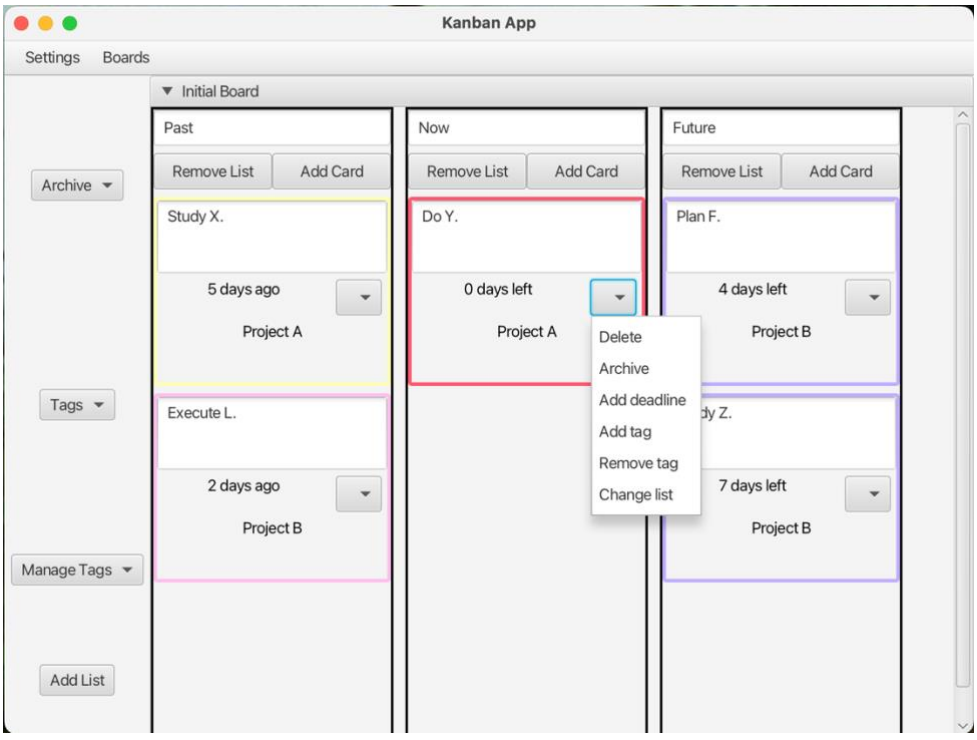

Figure 1: GUI of the application in example use case

The picture above is a screenshot of the GUI in an example use case. There are three main components of the application: a menu bar at the top, a control bar at the left, and a kanban board at the right side of the screen. From the top bar, users can access settings like changing the name of the current board, adding/deleting boards, and saving the current status of the application to external files. From the control bar at the left side, users can access the archive pile of cards and return archived cards back to their lists, see the list of tags and filter cards by the tags, create/delete tags from the board, and add new lists. From the kanban board which is on the right side of the screen, users can interact with lists and cards. They can remove lists or change their names. They can add cards to the lists and interact with the cards from there as well. They can change the card descriptions, delete cards, archive cards, add deadlines, add tags, remove tags, and change the list of cards. They are able to change the lists of cards by simply dragging and dropping them from one list to another.

The interaction between users and the application mainly occurs through buttons that users can click to activate some functions, such as adding a tag to a card or removing a list etc. Oftentimes, those buttons create alerts or dialogue pop-ups to get more information regarding the desired action. For example, when the user clicks on the add deadline button from the list of actions that can be reached from the down arrow placed on a card, a date-picking dialogue pops up on the screen where users can either type the deadline into the text box or they can choose the date from the calendar. Another example is deleting a board from the application. If users click on the delete board button under the settings list that can be reached from the top bar, a dialogue with a list of all boards pops up on the screen so that users can pick the board that they wish to delete. In cases where performing the desired action is not possible such as trying to delete a board when there are no boards at all, then an alert pops up on the screen to inform the user about the situation and about possible actions regarding the situation, as in this case, how to add a board to the application. There are many different cases like this where the user interacts with such pop-up dialogues and alerts. More example screenshots on this can be found in the appendices section.

The application can be run by running the “App.scala” file under the “GUI” folder, and in the initial launch there is only one board named “Initial Board”. Then users can use the application as they wish and while closing the application, a confirmation dialogue confirms if the user wants to save the changes made during the session. During the next launches of the application, boards, cards, and all the related details will be restored from the external files that are saved. However, if users don’t save the changes then the changes will be lost and they are not going to be transferred to next launches.

4. Program structure

In a Kanban board code, the main classes are Board, Stage, and Card. The rest of the classes are either to store them, to help them, or to show them to the user visually. In the UML diagram below, you can find all of the classes and objects that exist in the Kanban Board application, both on the backend side and the GUI side.

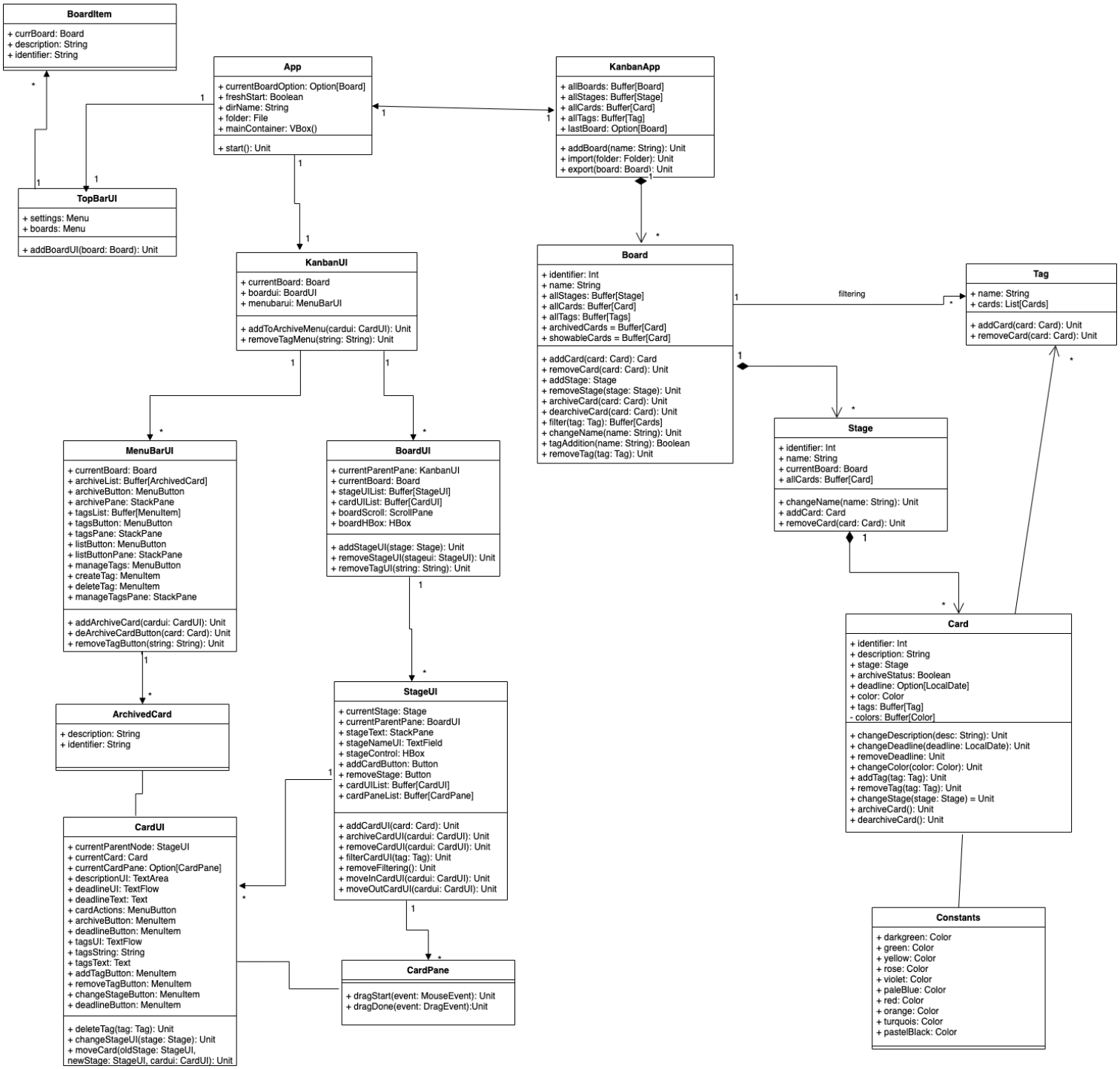

Figure 2: UML Diagram of the Kanban Board application

In this application, KanbanApp and App objects are the baseline of everything, as you can in the graph. They are the conjunction objects of the backend and GUI together. On the backend side several important methods such as board creation, import and export, and located at the KanbanApp object. Also, KanbanApp object hosts the storage of all Boards, Stages, Cards, and Tags that exist in the application. After that, the Board class comes. This class represents a kanban board and a user can store several boards in the KanbanApp object. At the Board class, there are storage containers for stages and cards, methods for adding/removing stages, and methods for managing and filtering cards by tags. After that, the Stage class comes, where the user adds cards to and removes cards from. The next class is the Card class, which is the most important class for the backend logic of the application. As you can see, most of the values are stored in the Card class and it also contains the majority of the methods. This is mainly because a card is the most interactive part of the application. It has a description variable to store a textual description of the card and methods to change it, a deadline variable to store the deadline and methods to change it, a colour variable and related methods, a tag container and tag addition/removal methods. As you can see, there are many important variables that the application stores in the Card class. This also creates a need for methods to control those variables. In general, most of the methods in the Card class perform similar tasks on the variables, adding, changing, and removing them if possible. The rest of the classes exist to help the Card class. The last class of the backend application is the Tag class. As you can understand from its name, it is there to tag cards and filter them accordingly. It has a textual description and stores the cards attached to it. As cards also store the tags that they are attached to, there is a two-step verification of the relationship between cards and tags. Tag class also interacts with the Board class to enable the app to filter cards based on their tags.

When it comes to the GUI, the App object is the critical part. It is where the GUI is started and it bridges the GUI with the backend due to its connection to the KanbanApp object. App object launches the GUI with its start function and also stores some critical values such as the current board that the user operates on. The App object is connected to two classes: TopBarUI and KanbanUI. TopBarUI class represents the top bar menu of the GUI and it has two menus, settings and boards. From the settings menu users can change the name of the board, add/delete boards, and save the application to external files. From the boards menu, users can switch between different boards. Here, boards are represented by the BoardItem class which is created for each board and it is an extension of MenuItem class of Scalafx [2]. On the other hand, KanbanUI class represents the more important elements of GUI. Along with some helper functions, it serves as a canvas for MenuBarUI and BoardUI. MenuBarUI contains some

buttons and menus to make some changes to the board and its view along with some helper functions that are needed for operations. It is also connected to the ArchiveCard helper class which is an extension of the MenuItem class of Scalafx and is used while representing archived cards under the archive menu. BoardUI is the visual representation of the Board class in the backend. It is the pane where different stages/lists are placed. With some UI variables added on top of each other, it can be scrolled vertically and horizontally. Then comes StageUI, which is the visual version of the Stage class. It simply visually represents the same function as the Stage class with quite a similar structure. The same can be said for the CardUI class, as it mostly contains buttons and text fields that are used to visually show and modify the variables in the Card class. It also has some helper functions that are used for connecting the backend logic with GUI. Finally, CardPane is a very simple helper class used while placing the cards in the StageUI. It is especially useful during the drag-and-drop motion since it would have been harder to implement that motion in the CardUI function which has a lot more additional elements.

5. Algorithms

In the application, many of the actions are simply manipulating variables in certain ways. Therefore, it is fair to say that there are not any complex algorithms or mathematical calculations. However, there was one thing that I also mentioned in the technical plan that I put some thought into: identification. Since the classes are so connected to each other and they should be saved/restored from external files, the connections between them should be built correctly to avoid problems. For that reason, I created a simple logic for creating unique identifiers for each element of the application such as cards, stages, and boards. The algorithm to create these identifiers will be as follows: 1. 1-digit int for the type of element 2. Date that element was created 3. 5-digit random number. For example, if a user creates board at 23.04.2024, an example identifier of that board element can be “1/2024-04-23/60036”. In the identifiers, 1 represents boards, 2 represents stages and 3 represents cards. These identifiers are registered to the class instances and they are saved to the external files as well. In many cases, I was able to make sure that operations were conducted on the correct elements by comparing the identifiers.

6. Data Structures

In this application, most of the information is in either String type or Int type and a few them in Boolean type. This was a result of a practical need since the main information related to elements (cards, lists etc.) should be stored in a written file in the directory. It is easy to process such files since there is a vast library with a variety of methods. Also, they fit perfectly

well with the comparison methods and if conditions, both of which I used extensively in this application.

For the containers, I preferred mutable Buffers. There are positive and negative sides to this choice. Former is Buffers are a rather flexible collection type. They are expandable, they are indexed, there are different methods available for processing buffers, and it is easy to transform a buffer to another collection type to name a few. However, there is a negative side too, just as I mentioned. Even though it is not a big deal, you need to import buffers explicitly, they are not originally available in the beginning. Still, Buffer is the collection type that I feel most comfortable using, so I wanted to use it. It is similar to Lists in Python, which is the language I use most, therefore Buffers seemed like a logical choice as a container.

7. Files and Internet access

One of the most important parts of this application was that users are able to save their changes to external files and restore them from external files as well. For that reason, I needed to come up with a way to store information in files. Pretty much all the data in the application can be represented as text, thus I thought standard text-based files would be a good option. As I presented in my technical plan, I was thinking of nested folders and text files, however, I changed my opinion and decided to use JSON files instead. In the final file structure, every board is a JSON file where the board, stages, and cards are objects under that. It also utilizes the arrays defined in JSON to represent the list of stages under the board and similarly cards under a stage. In Figure 3, you can see the basic example of JSON structure I used in this application.


```
{"boardID":"boardID",
  "boardName": "boardName",
  "boardTags": [
    "tag1",
    "tag2"
  ],
  "stages": [
    {
      "stageD": "stageD",
      "stageName": "stageName",
      "cards": [
        {
          "cardID": "cardID",
          "cardDescription": "cardDescription",
          "deadline": "deadline",
          "tags": [
            "tag1",
            "tag2"
          ],
          "archiveStatus": "T/F"
        }
      ]
    }
  ]
}
```


Figure 3: JSON file structure of an example board

There are quite many libraries in Scala to process the JSON files. However, I chose the Play framework due to its very easy-to-use Writer and Reader methods [2]. They work well with case classes I created and made it very easy to process JSON files with a very low risk of IO exceptions.

8. Testing

Since this application requires more work on the GUI side compared to the backend side, in my planning phase I have decided to use GUI in combination with test cases. This is in fact what happened during the implementation. I have written a test file by using Scalatest library with around 15 different cases to see if the base logic of the application such as managing boards, stages, and cards works as they are intended to [3]. In those unit tests, pretty much everything seemed to be working fine without any issues, after which I decided to start working on the GUI.

Testing the GUI is a different case since it is technically possible to write test cases for GUI methods as well, but it was not something that was either familiar or wanted. Therefore, I tested the GUI simply by using and interacting with it, many times trying to replicate real-life cases. As I am also an active user of similar products, I was already familiar with what users might want to do so I believe I did extensive testing for the GUI.

After coding the GUI, I started working on file-saving functions, which I again tested with unit tests. With that, I managed to test every part of the application either by unit tests or by using the GUI, making sure that things worked smoothly without any noticeable issues. I personally think if the users interact with something visually, then it is a better idea to test it visually as I did with the GUI. However, the backend of the application is not something that users see so it can be handled by writing test cases and seeing how they go. I tried to follow this idea, and I am happy with how the testing phase went for me.

9. Known bugs and missing features

In my plans, I stated my goal was to complete the project at the challenging level. Due to some scheduling issues on my side and lack of time, I decided that it was not realistic for me to go for the challenging level and instead, I completed the project at the intermediate level. Therefore, there are many missing features that I planned how to implement but, in the end, I did not. For example, I wanted to have checklists with progress bars, background images, attachments, templates, and some special actions that can be activated by pushing a button. For some of these, I even wrote the backend code but later in the process, I realized that I lacked time to implement GUI including those features and ended up leaving those elements out of the project completely. In the code cleanup, I have removed classes and methods related to

those features as well to avoid possible confusion about whether those features are part of the project or not.

In terms of the bugs, I believe it would be fair to say that this project is almost bug-free. The testing structure that I explained in the previous section made me find many bugs earlier in the process and when you have less amount of code, it is easier to spot bugs and fix them. However, there is one bug that I found but wasn't able to fix it. When the user clicks on the "Add deadline" button from the drop-down menu of the card, a date-picking dialog pops up on the screen allowing the user to choose a date. There, the user can either pick the date from the calendar or type the date in the text box presented. The bug happens when the user types a random string into the date-picking text box. Since the date picker, which is part of the Scalafx library, cannot convert that random string to a date, it gives an error. In practice, it doesn't affect much since the app can be used without any issues after that, however, I still realized the problem while looking at the console logs. I wasn't able to fix it mainly because the problem was part of the Scalafx library which I couldn't change much about [5]. I guess there is a workaround to fix the bug, but I didn't have enough time to dive deep into it as also because its effect is rather limited. Other than this, there aren't any known bugs in the application and it seemed to work without any major issues.

10. 3 best sides and 3 weakness

Like all applications, this one has its best sides and weaknesses. I think the three main best sides of this application are:

- Pretty much all the requirements for the intermediate level were implemented
- The algorithm works without any major issues and the user experience is smooth
- GUI is easy to understand without any specific instructions

On the other hand, the three main weaknesses that this application has are:

- Application is not optimised in terms of time and memory efficiency
- GUI could have been decorated better to give it a more aesthetic view
- Some parts of the code are hard to understand due to variable/function names and some of them could have been implemented in more concise ways

It is natural for an application to have weaknesses, and there is always room to improve. I believe there are other fields that this application that can be improved but at the same time it has a good core user experience.

11. Deviations from the plan, realized process and schedule

As I mentioned before, the main deviation from the plan is that instead of the challenging level that I initially intended to complete the project, I completed the project in the intermediate level. That made all my planning regarding the requirements of the challenging level insignificant as they were not implemented in the final version.

Things didn't go as planned in terms of the schedule either. As I also wrote my thesis at the same time as this project, during the weeks before thesis draft deadlines I didn't have any time for the project at all as I was working on the thesis with my full power. That being said, for every sprint I had some new stuff implemented. In the first sprint, I was more focused on turning my legacy code for the project into something useful for this year. In the second sprint, I implemented test cases, designed the UI in SceneBuilder, and started to build the UI [6]. In the third sprint, I did most of the coding for GUI, finished the visual side of it and connected it with the backend. After the third sprint, I worked on file-saving functions and a few features left in GUI along with polishing the application overall. My initial plans were about working on the GUI earlier and implementing the backend code along with the GUI but the legacy code I had from my tries on this project from previous years made me change my idea. Even though the process didn't go the way that I planned, I still believe I had a good scheduling of the tasks and good distribution of work during the process. Now I learnt that before planning a project, it is better to first look at what you have in your hands and how useful they can be. That way, you can be more accurate with the schedule and the time estimations.

12. Final evaluation

I believe this project is quite good in terms of the core user experience and the smoothness in general. However, there is a clear room for improvement in the visual side of the application. I am personally not very good at design in general and to be honest, I don't really enjoy coding UIs. For that reason, I mostly relied on the classical view of things in Scalafx library [7]. So, in the future, this application can be mostly enhanced to give it a more aesthetic look. That being said, I think the application still looks okay. The application also can be optimized both in terms of time and memory. However, as the resources needed are not very high for such an application, the lack of optimization doesn't affect the user experience in any way.

I also learnt that if you put more time into planning and making a more reliable plan, it would be helpful later in the process. Otherwise, things might get confusing and the stress might be more affecting you. Anyway, I am super happy with everything and looking at the application makes me really happy. At the end of the day, creating something is the best feeling in the world!

13. Bibliography

[1] "Kanban (Development)," [Online]. Available: https://en.wikipedia.org/wiki/Kanban_(development). [Accessed 24 04 2024].

[2] ScalaFX, "MenuItem," [Online]. Available: https://javadoc.io/static/org.scalafx/scalafx_2.12/14-R19/scalafx/scene/control/MenuItem.html. [Accessed 24 April 2024].

[3] Play Framework, "JSON basics," [Online]. Available: https://www.playframework.com/documentation/3.0.x/ScalaJson. [Accessed 24 April 2024].

[4] Artima, "Scalatest," [Online]. Available: https://www.scalatest.org/. [Accessed 24 April 2024].

[5] ScalaFX, "DatePicker," [Online]. Available: https://javadoc.io/static/org.scalafx/scalafx_2.12/14-R19/scalafx/scene/control/DatePicker$.html. [Accessed 24 April 2024].

[6] Gluon, "Scene Builder," [Online]. Available: https://gluonhq.com/products/scene-builder/. [Accessed 24 April 2024].

[7] ScalaFX, "Scalafx," [Online]. Available: https://www.scalafx.org/. [Accessed 24 April 2024].

14. Appendices

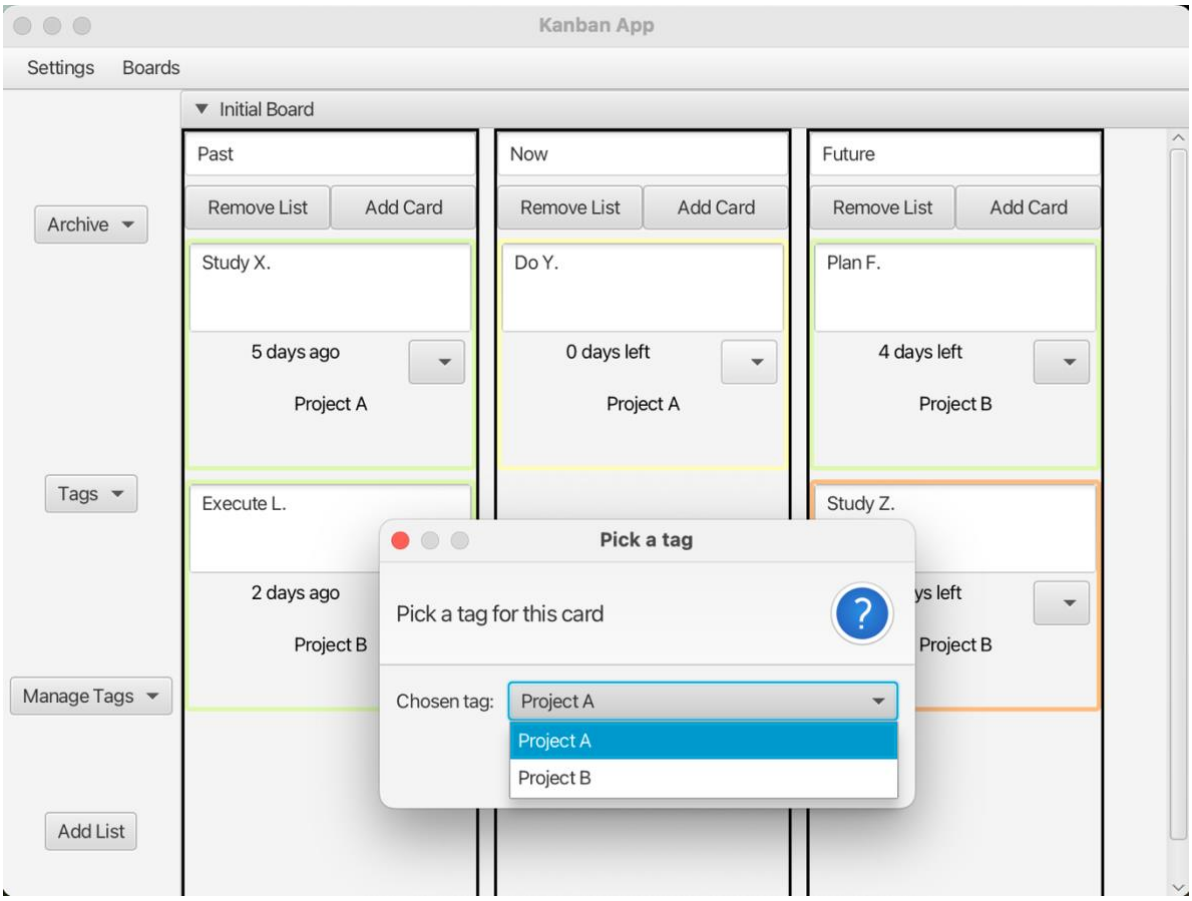

Figure 4: Adding a tag to a card

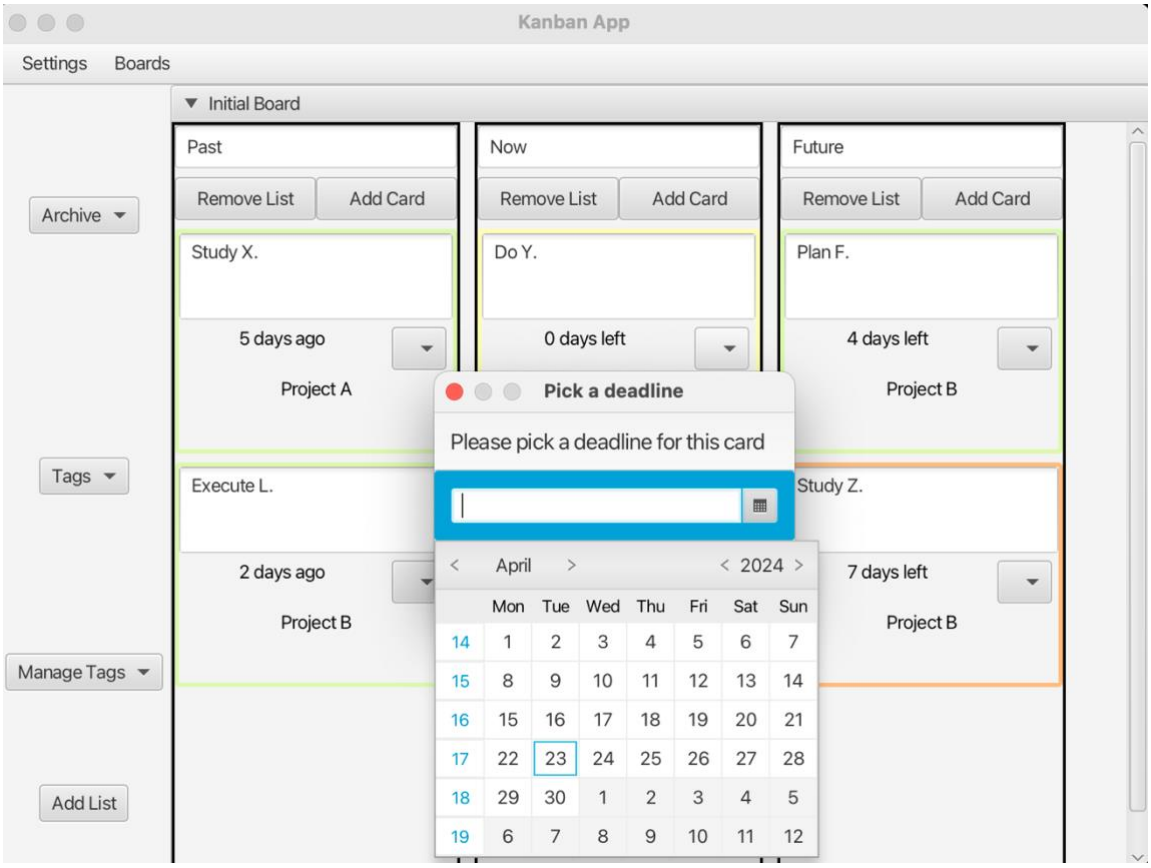

Figure 5: Adding a deadline to a card

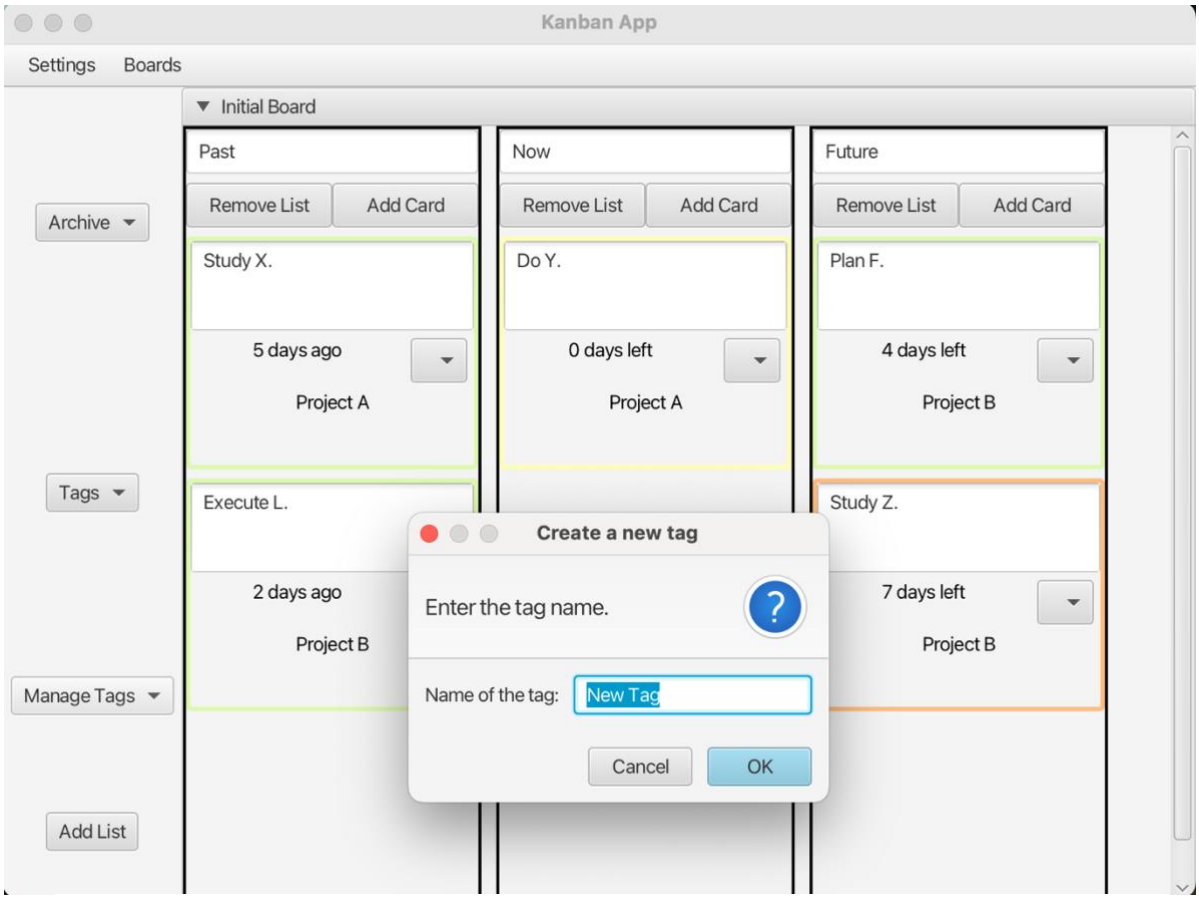

Figure 6: Creating a tag at a board

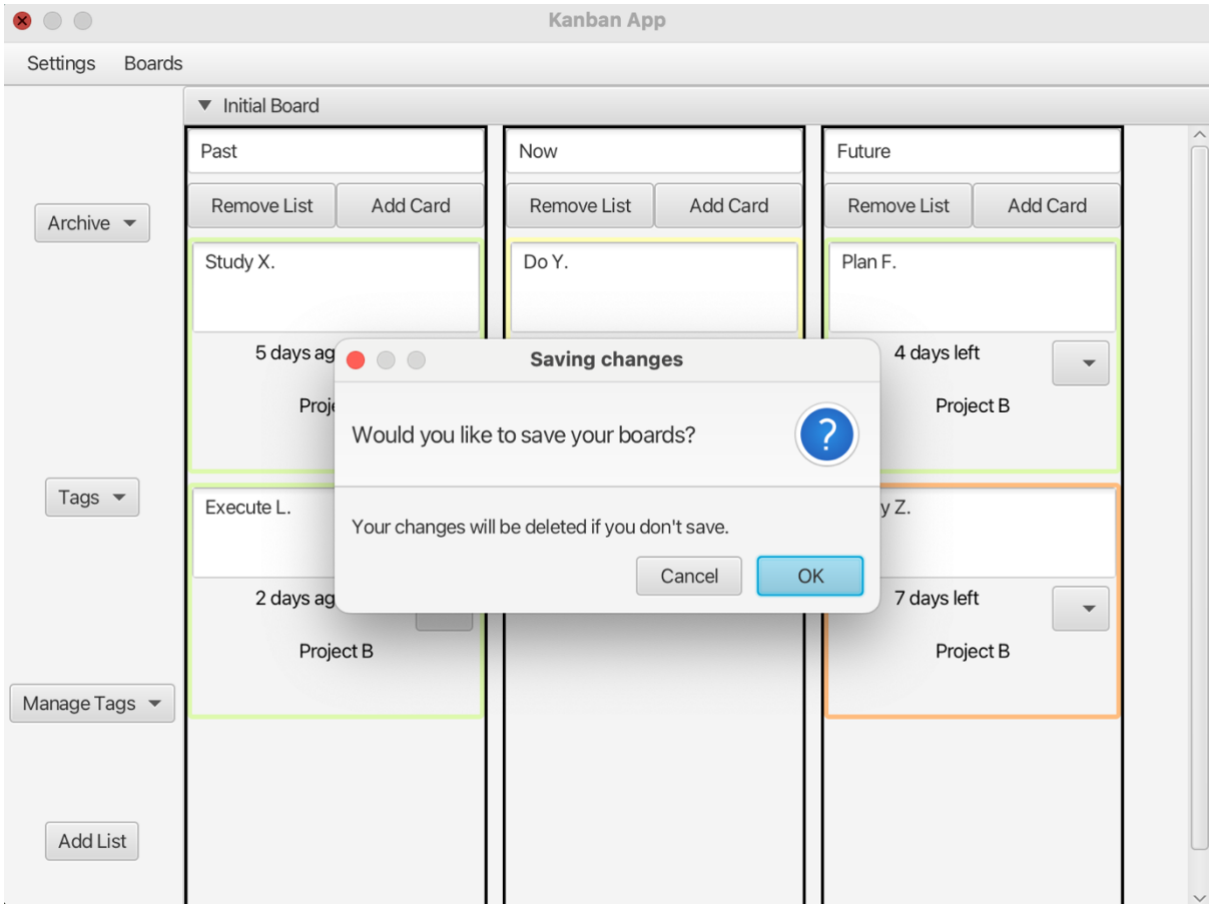

Figure 7: Saving question while quitting the application