1. Class Structure

In a kanban board code, the main classes are Board, Stage, and Card, respectively. The rest of the classes are either to store them or to help them. In the UML class diagram below, you can find all of the classes that exist in the Kanban Board application.

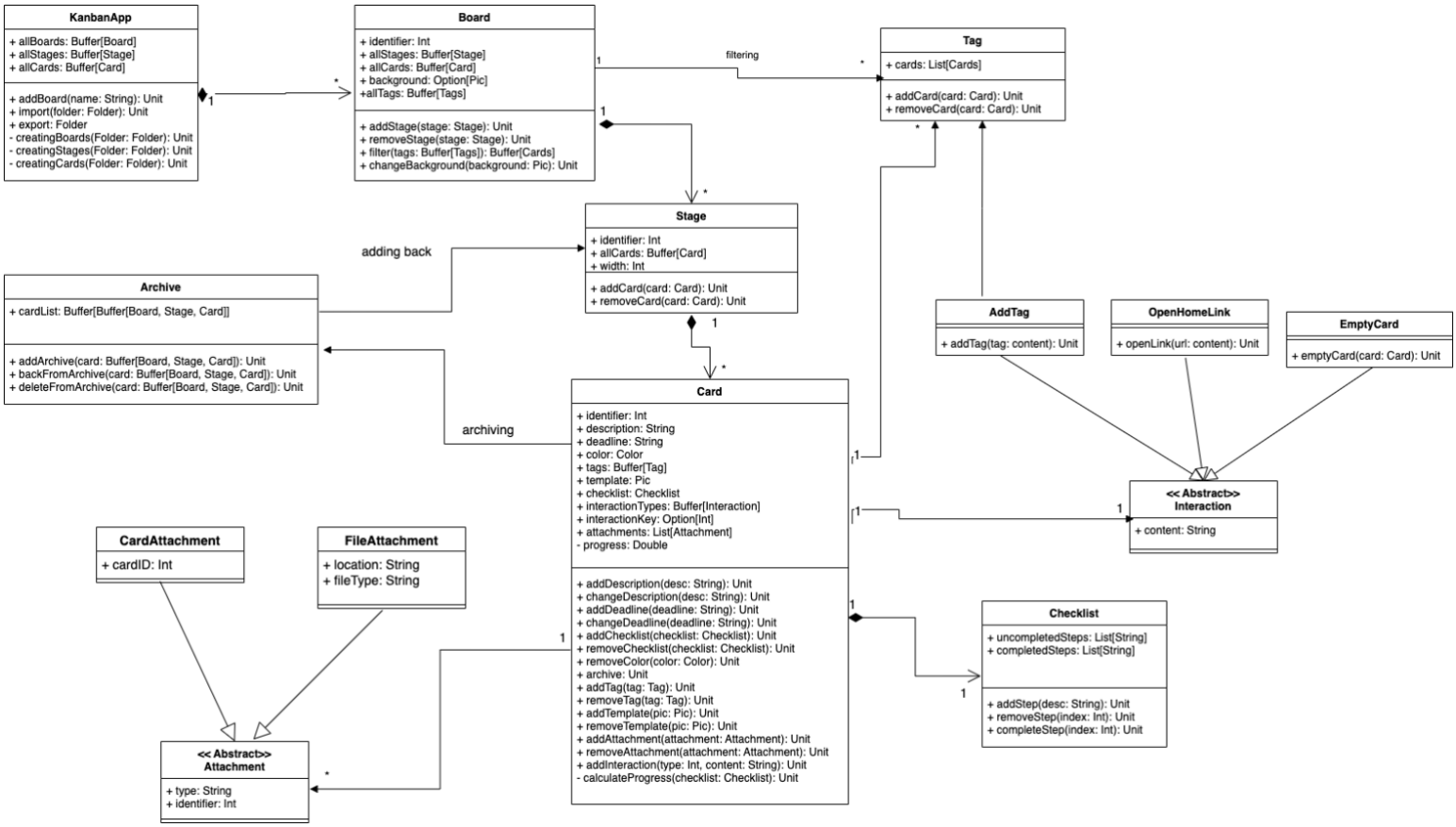

Figure 1: *UML Class Diagram of Kanban App*

In this application, KanbanApp class is the baseline, as you can in the graph. It is the final class that all other classes are linked to. Board creation, import, export, and several other helper methods are located. Also, KanbanApp class hosts the storage of all Boards, Stages, and Cards 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 class. At the Board class, there are storage containers for stages and cards, methods for adding/removing stages, method for filtering cards by tags, and as required in the challenging level a method to change the background. 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 of the whole 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 color variable and related methods, the tag container and tag addition/removal methods, a template variable to store the template if there is one and methods to change it, a checklist variable for a possible checklist, interaction related variables, attachment container variables and some more methods to control those variables. As you can see, there are so many variables that the application stores in the card. This also creates a need for methods to control those variables. In general, most of the methods at 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, except one.

The first helper class is the Tag class. As you can understand from its name, it is there to tag cards and filter them accordingly. It also interacts with the Board class to enable the app to filter cards based on their tags. Secondly, the abstract Interaction class and its inheritance classes come. These classes are for button interactions that users can add to their cards. Right now, there are 3 of them, AddTag, OpenHomeLink, and EmptyCard. Their functions are adding the pre-defined tag to the card, opening a general link that the user set, and deleting all the content of the card, respectively. With the flexibility that this implementation structure gives, the number of interactions can be increased in the future without significant effort. Another example of helper classes is the abstract Attachment class and its inheritance classes CardAttachment and FileAttachment. These two classes are the possible attachments of a card. An attachment can either be another card or a file such as an image file. Like the interactions, the number and type of the attachments can be increased easily. The last helper class is the Checklist class, which enables users to create checklists do step-by-step planning. Most of the helper classes don't have significant methods, as you can see. The only exception to this is Interaction classes, which exist to perform a task when the user presses a button.

There is also an Archive class, which is neither a so-called main class nor a helper class. It is a class designed for archiving the cards and returning them back to their previous position. For that reason, there is a circular relationship between Card, Archive, and Stage classes. While you can archive a card from its "archive" method when you want to turn it back from the archive you use the "backFromArchive" method which commands a stage to add a card. You can also delete cards from Archive's storage which will lead to permanent deletion of the card.

2. Case Structure

A typical usage scenario of a Kanban Board application is that the user opens the application and starts to create a project timeline from scratch. In this application, the user gives all commands from the graphical interface. So, the first thing that the user will do is giving a

name for the empty kanban board. Right now, there is only one board stored in the application. Then by clicking the plus button at the lower-left corner of the screen, the user adds different stages and gives them names. This action will create a number of stages that are stored on the board. Then, the user creates different cards for those stages by clicking the plus button located at the bottom of each stage, write textual descriptions for those cards and add attachments, a checklists, or a deadline, as he/she wishes. The cards will be created by the command given to stages and stored in the card containers while card additions are created by the methods of the Card class and stored by them. The user can also define a button for cards, which will do the same pre-defined task when the user presses this button. The user always has the chance to change the task or remove the button completely. If the user wants so, he/she can archive some of the cards and as wished can return them back to their previous location with all extensions preserved. All those actions can be done by using the graphical interface only as described. In fact, the user initially doesn't have to create a stage since all boards come with one empty stage. However, in reality, kanban boards include multiple stages and generally, users create multiple stages first. To order cards and stages, the user can drag and drop them to their new locations. If the user doesn't specify an order, cards and stages will be ordered according to their creation date, last created at the last position.

As you can see, the tasks in the application mostly happen "from outer to inner" kind of order. Besides the Archive and Tag classes where the former archives cards with the command of Card class and returns them back by commanding to the stage class, and the latter notifies Board's tag container to update information as a new Tag was created or a Card is tagged with a new tag, there isn't any reference that skips class order, which is Board-Stage-Card.

3. Algorithms

While designing the application I spent most of my time finding the best way to organize files. Since the app files should be importable/exportable, as I stated before, I thought that building a matryoshka-doll-like file structure would be appropriate. Doing that requires the app to distinguish each element from the others. For that reason, I designed the "identifier" variable for boards, stages, and cards. This will be a unique identifier in integer type by which the application can understand which exact element it is trying to save or read at the moment. The algorithm to create these identifiers will be as follows: 1. 1-digit int for the type of element 2. Date-time that element was created 3. 5-digit random number. For example, if a user creates board at 18.02.2022 14.56, the identifier of that board element will be "1180220221456*****", where '*'s will be random numbers. In the identifiers, 1 represents boards, 2 represents stages

and 3 represents cards. Attachments of cards will have the same identifier with the card that they are linked to, plus one more random number in the end. By reading file names with the help of Java libraries, the application will distinguish elements. One important thing here is the folder of Board in the directory will include folders of the stages that it contains and the same applies for stages – cards. This will prevent any confusion in the classification of elements, like assigning a card to a wrong board/stage.

This design choice requires high interaction with directory folders/files, hence Java libraries designed for that reason will be quite useful for that. Private methods in KanbanApp class will handle the names of those folders/files and handle them with string processing methods. These methods are designed to be inaccessible from out of the KanbanApp class and they will command public methods to actually create elements.

For card extensions like buttons and attachments where users have multiple choices, like a file or card attachment and different button operations, the application will distinguish types with an identifier integer. So, when the user wants to attach something to a card, the application will first ask which type of attachment it is, card or file. Depending on the answer, the application will use different algorithms. Such as, if the user wants to attach a file the app will store the file with the name given according to the description above and store its location. If the user wants to attach a card, the application will store the identifier of the card which is attached. The same mentality applies to buttons, where the application first asks for the button type and continues according to the answer. This means, in this application the match-case structures will be used.

4. Data Structures

In this application, most of the information is in either String type or Int type. This is mainly because the main information related to cards is stored in a text file in the directory. It is easy to process text files since there is a vast library with a variety of methods. Also, they fit perfectly well to the match-case structures and if conditions, both of which I plan to use extensively in this application.

For the containers, I preferred 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, 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. This might be because we worked with Buffers so much during O1, and if we have given a choice I always chose it.

5. Files

As explained in the general plan in detail, boards and stages/columns are going to be represented as a folder while the cards are structured as text files and attachments are saved in their own file type. Like said before, the matryoshka doll would be a good example to imagine this structure. Every folder (board or stage) will be named in its own unique way just like the cards which were put inside them with their attachments. Cards, the most important part of the application, will be built as text files in the following way:

1. Stage: *StageName*
2. Text: *TextDescription*
3. Tags: *Tag1, Tag2, Tag3*
4. Color: *R: x, G: x, B: x*
5. Deadline: *DD.MM.YYYY*
6. ChecklistItems:
   Item1 – Status
   Item2 – Status
   Item3 – Status
   Item4 - Status
7. CardActivity: *ActivityKey*
8. LinkedItems: *ItemCodes*

Here, all the information would be saved as text and mostly in the form of identifier codes, which connects different parts of the application. This is a harder way to build things in the beginning but it minimizes the risk of some serious problems: multiple items with the same name, scalability issues, and the program having a hard time reading some of the names due to unique characters.

6. Schedule

After creating classes in general, I plan to build the core of the application first and other detailed classes later. That means I plan to code Card, Stage, Board, and KanbanApp classes first. Helper classes and the Archive class will be implemented after building the core of the application. Considering the deadline at the end of the April and my plans to start at the

beginning of March, I have 8 weeks for this project. Hence, my plan for this project is as follows:

a. Weeks 1-2: Initial start of the project, implementation of the Card, Stage, Board, KanbanApp classes. I don't plan to implement file storage and export/import methods at this stage. This stage will take approximately 20 hours.
b. Weeks 3-4: Main elements of GUI. I plan to build GUI in general at this stage but there will be unimplemented parts like Archive and helper classes. Here, I will also simulate a real-life case to see how the classes and methods I coded work with GUI. This stage will take 25 hours.
c. Weeks 5-6: I plan to build file storage, import/export functions, Archive class, and helper classes. Their GUI side will not be implemented at this stage. Building import/export functions will also give me the ability to test all internal functions at once, which I will do with different files. This stage will take approximately 30 hours.
d. Weeks 7-8: GUI implementation of lately build classes and functions, final testing, polishing, and debugging. This stage will take 15 hours.

According to the plan above, I expect to spend the most time while implementing file storage and import/export functions. This is mainly because I am not too familiar with file management and while doing that, I will use Java libraries which I will learn from scratch. After that, the first phase of GUI implementation comes. This is also because of my lack of knowledge and experience in GUIs. However, I expect to spend less time at the second stage of the GUI implementation since I already plan to have a general structure of GUI built at that time.

I also reserved some free time at the final stage, just if I face some unexpected big errors there. I hope not to and to avoid such cases, I will always try the classes and methods I implemented. In that sense, holistic tests that I will conduct after building export/import methods will play a key role.

7. Testing Plan

The design of my code and some methods required by the course enables me to conduct unit testing in 2 different ways easily. The main reason for this is the import function. Import function requires the application to build a complete Kanban Board from scratch according to the given file without using any GUI element -in fact, the user will use a simple import button but this is not something crucial in this testing case. So, I plan to conduct system testing in both ways. I will create 2 different scenarios and test one of them by using GUI and the other one by using test cases. In both cases, the core elements of the application will be tested. That

means KanbanApp will create Board(s), the Board will create Stage(s), and the Stage will create Cards with different properties.

In different testing methods, the application should handle error situations in different ways. For example, GUI should prevent error actions, reject applying them, and create visible error messages while testing by test case created with the help of import method should stop when it faces with error situation, mainly because of the design of the import function. Of course, it is possible to write test cases without using the test cases but I will not prefer to do that, mainly because GUI and its drag-drop functionality have a key role in the application. Therefore, I believe the system testing should be conducted either by using GUI or using the import method.

However, I believe that test cases written without using the import method are really useful for unit testing. I can use them for testing several functionalities. The first one is the very basics of the application, Board, Stage, and Class creation. They are interdependent classes therefore they should be tested together. The second part of the unit testing is the Card helper classes. All of them can be tested with test cases written with reference to the Card class. In these tests, the main input will be commands to create application elements -classes- with different properties and expect the program to give them as output, add them to the correct collections. The third usage area of test classes is unit testing of the Archive class. It is a part of the program which doesn't really affect the general system if we take them out. That means it is perfect for unit testing. While conducting the unit testing of the Archive class, the test case will command the program to move some cards to Archive and move some cards from the Archive back to the actual program. After the test, it is expected that both Archive and actual program collections will contain correct Cards.

8. References and Links

https://trello.com

https://docs.oracle.com/javase/7/docs/api/java/io/File.html

https://otfried.org/scala/gui.html

https://www.javatpoint.com/java-swing

https://alvinalexander.com/scala/how-to-list-files-in-directory-filter-names-scala/

https://stackoverflow.com/questions/48932630/scala-creating-directory-and-file

https://stackoverflow.com/questions/28947250/create-a-directory-if-it-does-not-exist-and-then-create-the-files-in-that-direct

https://stackoverflow.com/questions/6214703/copy-entire-directory-contents-to-another-directory

https://gist.github.com/rfries/6ca2c2fd002e79365df0e712458b3a42