Assignment 3

Team number: 99
Team members

Name	Student Nr.	Email
C.T.T. Le (Thành)	2737387	lecungtuantranh@student.vu.nl
Shivangi Kachole	2634561	s.c.kachole@student.vu.nl
Kyle Schippers	2729563	k.a.schippers@student.vu.nl
Quang Tu Le	2723610	q.t.le@student.vu.nl

Diagrams.io link:
https://drive.google.com/file/d/1clDFlFqJk1aL9_QEQBhGq1hW0qoX5f2u/view?usp=sharing
(Open pages Class Diagram asg3, Object Diagram asg3, Sequence Diagram asg3, State Machine Diagram asg3)

Summary of changes from Assignment 2

Author(s): Tu, Thanh, Shivangi, Kyle

Class Diagram
- We added the multiplicities to the attributes that have lists
- We have added better explanations for each of the diagrams
- We have mentioned the functionality of each of the classes
- Each class has a single responsibility

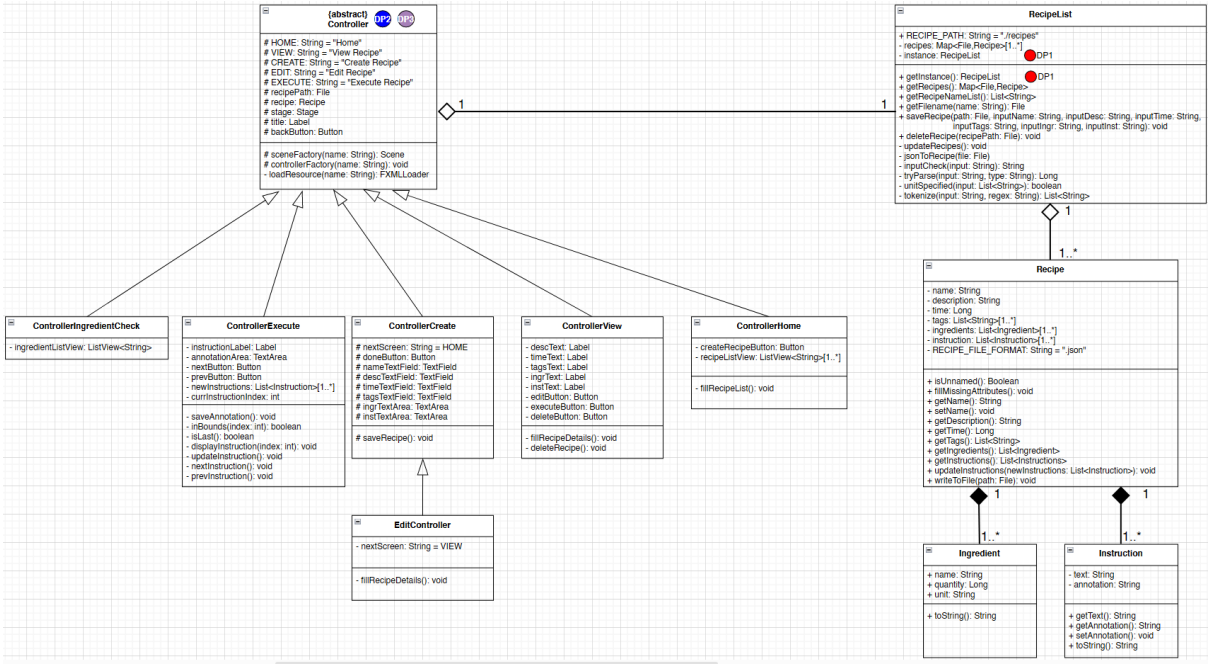
Object Diagram
- We showed the snapshot of the full execution by adding the instruction class as well.
- We also added the notes for each of our instructions

State Machine
- State machine diagrams are more complex than earlier so we did not need to create more diagrams.

Sequence Diagram
- Conformed with the operations mentioned in the class diagram
- Made the diagrams more representative of our implementation

Revised class diagram

Author(s): Tu, Thanh, Shivangi, Kyle



The changes we have made for each of the classes are listed below. However in general, the changes we made are that we added controller classes, since they handle our GUI part. Then we have removed Pantry, ShoppingList and IngredientList classes since it was making our system more complex.

RecipeList

- RECIPE PATH
  - This attribute stores the path to the recipes folder that contains all the recipes in the json file.
- recipes
  - A private variable that is a map, where each file stores one recipe. The key is the file and the value is the recipe it stores.
- Instance
  - A private variable that is the instance of the RecipeList class, which is of the type RecipeList. This is for the singleton pattern.
- + getInstance(): RecipeList
  - This is a public method that creates instance attributes of the RecipeList class.
- + getRecipes(): Map<File,Recipe>
  - A public method that returns all the stored recipes.
  - Especially helpful when new recipes are either added or updated.
- + saveRecipe(path: File, inputName: String, inputDesc: String, inputTime: String, inputTags: String, inputIngr: String, inputInst: String): void
  - This is a public method that takes the input from the user and creates a new Recipe.
  - This new recipe is then written to a json file in the json format by calling a method called writeToFile().
- + deleteRecipe(recipePath: File): void
  - This is a public method that deletes a recipe. It takes the file as an input parameter, and checks if the file does indeed exist and then deletes it.
- - updateRecipes(): void
  - This is a private method that updates recipes that already exist in the recipelist.
  - Although this is a private method, it is accessed by public methods such as getRecipes().
- - jsonToRecipe(file: File)

    - This is a private method that reads all the json files and converts from the json format to a string.
    - This method is accessed by methods like updateRecipes(), since reading files and checking if the recipe does exist in the recipe list is important for the update.
    - It is private since we don't want any external classes being able to access this parsing method.
- - inputCheck(input: String): String
    - This is a private method that checks if the fields have been entered or not.
    - If the text fields are not entered, then it throws an exception.
    - This method is called in the public method saveRecipe() since that method is the one that saves a recipe to a json file.
- - tryParse(input: String, type: String): Long
    - This method is similar to inputCheck(), however this method checks if the numbers are inputted in the input fields where numbers are required (for example, the amount of time required).
- - unitSpecified(input: List<String>): boolean
    - This is a private boolean function which checks if the unit has been specified in the input fields, where the users are asked to fill in the units (for example: time field, quantity of an ingredient needed in a recipe).
- - tokenize(input: String, regex: String): List<String>
    - This private method removes all the irrelevant information such as whitespace, extra delimiters or empty tokens from the input of the user.
    - The input from the user is in the format of a string.
    - This method is called in the public method saveRecipe(), since this is where the user is asked to fill in the fields.
- In general the responsibility of this class is to handle the CRUD operations, which is an improvement since in the previous design we handled few of the operations on another class called User and few in this class.
- We need this class to make sure that the users get access to the recipes and work with the recipes (like update them, or delete them).
- This has a shared aggregation with the **Controller** class since both can exist independently as well.
- Design pattern singleton is applied to this class. Explained why we used it in detail later on.

Recipe

- + name: String
    - This is a public attribute that stores the name of the recipe.
- + description: String
    - This is a public attribute that stores the description of the recipe.
- + time: Long
    - This is a public attribute that stores the time of the recipe.
- + tags: List<String>[1..*]
    - This is a public attribute that stores a list of tags that are associated with the recipe. We have given it the multiplicity of [1..*], since there needs to be atleast 1 tag needed for a recipe and a recipe can have any number of tags.
- + ingredients: List<Ingredient>[1..*]
    - This is a public attribute that stores a list of ingredients that are needed for a recipe.We have given it the multiplicity of [1..*], since there needs to be atleast 1 ingredient in a recipe and a recipe can have any number of ingredients.
- + instruction: List<Instruction>[1..*]
    - This is a public attribute that stores a list of instructions that are needed for a recipe. We have given it the multiplicity of [1..*], since there needs to be atleast 1 instruction in a recipe and a recipe can have any number of instructions.

- - RECIPE_FILE_FORMAT: String = ".json"
    - This is a private attribute that keeps the format of the files as json.
- + isUnnamed(): Boolean
    - This is a public method that keeps a check on whether the recipe inside the json file does not have a name.
- + fillMissingAttributes(): void
    - This is a public method that fills in default values for the attributes in case they are empty.
- + updateInstructions(newInstructions: List<Instruction>): void
    - This is a public method that only updates instructions.
- + writeToFile(path: File): void
    - This is a public method that writes to a json file, in json format. This method is called in other public methods in the RecipeList class such as updateRecipes() and saveRecipe().
- In general the responsibility of this class is to handle the Recipe.
- We need this class to make sure that there is a way to store a Recipe and keep tabs on the appropriate attributes and data that needs to be stored.
- This has a shared aggregation with the **RecipeList** class since both can exist independently as well.

Ingredient

- + toString(): String
    - This is a public method that formats how the ingredients should be presented on the screen.
- In general the responsibility of this class is to store each ingredient.
- This has a composite aggregation with the **Recipe** class since a recipe cannot exist without an ingredient and the other way around. The multiplicities are that one recipe can have one or more ingredients. This is because a recipe must have atleast one ingredient.

Instruction

- + text: String
    - This is a public attribute that stores the name of the instruction itself.
- + toString(): String
    - This is a public method that formats how each instruction should be presented on the screen.
- In general the responsibility of this class is to store each ingredient.
- This has a composite aggregation with the **Recipe** class since a recipe cannot exist without an instruction and the other way around. The multiplicities are that one recipe can have one or more instructions. This is because a recipe must have atleast one instruction.

Controller

- # recipePath: File
    - This attribute store the file path to the recipe.
- # title: Label
- # sceneFactory(name: String): Scene
    - This protected method takes care of different possible scenes that are possible and opens up the FXML file of that scene.
- # controllerFactory(name: String): void
    - This method switches the screen to a new controller.
- This class acts as a factory and also a controller class. This is a parent class to the rest of the other controllers. Each of the other controller classes have responsibility for each of the main functions that our system has to perform.
- Design patterns Controller and Factory applied for this class.

ControllerIngredientCheck

- - ingredientListView: ListView<String>
    - This is a private attribute that stores all the ingredients
- This class extends from the **Controller** class and has a generalisation relationship.
- The purpose of this controller class is to help the users check if the ingredients are present.

ControllerExecute

- - prevButton: Button
    - This is a button that takes you to the previous instruction
- - newInstructions: List<Instruction>[1..*]
    - All the new instructions are stored in this list.
- - currInstructionIndex: int
    - This attribute keeps track of the index of each of the instructions.
- - inBounds(index: int): boolean
    - This method checks if the given index is within the range of the list or not.
- - isLast(): boolean
    - This method checks if the current instruction's index is equal to the last index of the instruction list.
- - nextInstruction(): void
    - This method displays the next instruction in the instruction list. This continues to execute until the last instruction is reached.
- - prevInstruction(): void
    - This method displays the previous instruction in the instruction list. This continues to execute until the first instruction is reached.
- This class extends from the **Controller** class and has a generalisation relationship.
- The purpose of this controller class is to execute the recipe in a step by step manner.

ControllerCreate

- - nextScreen: String = VIEW
    - This private attribute ensures that the user is redirected to the Home screen after creating.
- # descTextField: TextField
    - This is a text field for the description of the recipe.
- # ingrTextArea: TextArea
    - This is a text field for the ingredients of the recipe.
- # instTextArea: TextArea
    - This is a text field for the instructions of the recipe.
- # saveRecipe(): void
    - This protected method saves the recipe from the GUI and saves it as a recipe object.
- This class extends from the **Controller** class and has a generalisation relationship.
- The purpose of this controller class is to create the recipe.

ControllerEdit

- - nextScreen: String = VIEW
    - This private attribute ensures that the user is redirected to the View screen after editing.
- - fillRecipeDetails(): void
    - This method edits the recipe and stores it.
- This class extends from the **CreateController** class and has a generalisation relationship.
- The purpose of this controller class is to edit the recipe.

ControllerView

- - descText: Label
    - This private attribute is the description text that stores the description of the recipe we are viewing.
- - ingrText: Label

    - This private attribute is the description text that stores the ingredients of the recipe we are viewing.
- - instText: Label
    - This private attribute is the description text that stores the instructions of the recipe we are viewing.
- - fillRecipeDetails(): void
    - This method shows the entire recipe on the screen.
- This class extends from the **Controller** class and has a generalisation relationship.
- The purpose of this controller class is to show the recipe to the user.

ControllerHome

- - fillRecipeList(): void
    - This method shows only the names of the available recipes.
- This class extends from the **Controller** class and has a generalisation relationship.
- The purpose of this controller class is to show only the names of the recipes to the user.

Application of design patterns

Author(s): Tu, Thanh, Shivangi, Kyle

	DP1
Design pattern	Singleton
Problem	Different parts of the system may modify the recipe list and create conflicts during runtime.
Solution	Use Singleton pattern to restrict the creation of RecipeList object.
Intended use	Singleton pattern ensures that only 1 list of recipes exist at any time during the system's execution. Different controller classes can access the same RecipeList using the getInstance() method.
Constraints	Method chaining can be verbose and hard to understand
Additional remarks	None

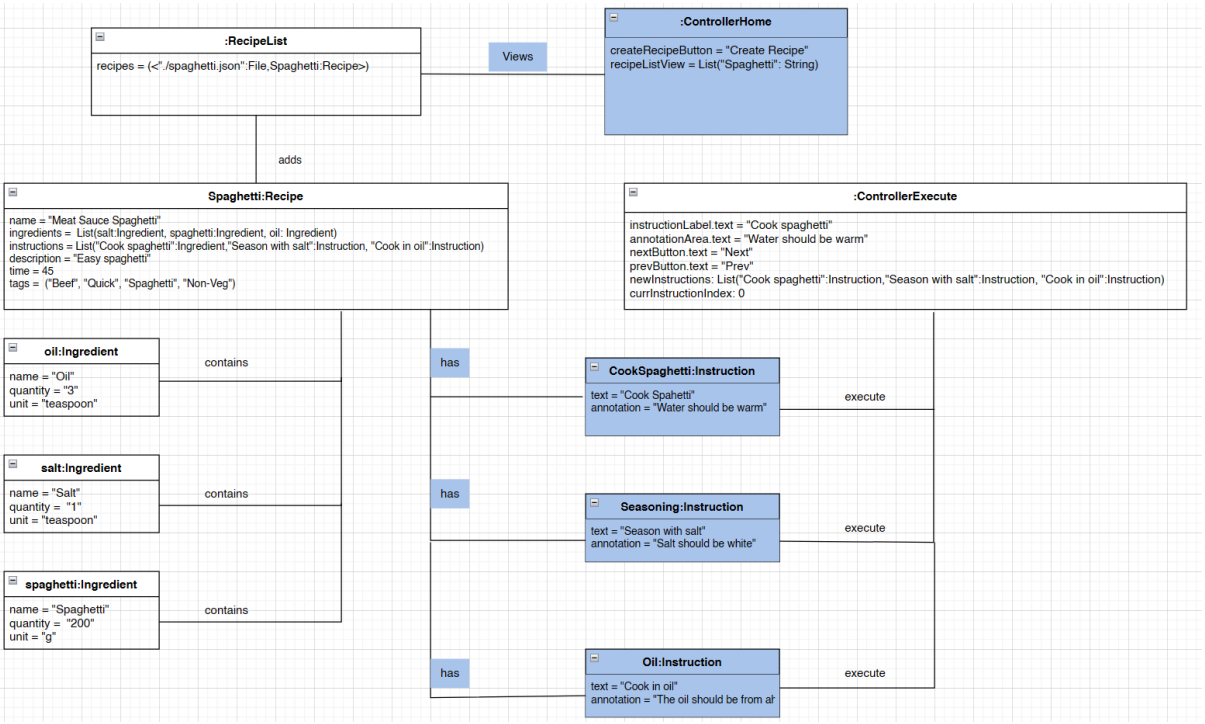
	DP2
Design pattern	Factory
Problem	There are multiple Controller types with different parameters in their constructors.
Solution	Use a controllerFactory() method to simplify the creation process, only the type is needed as an argument to the method.
Intended use	When switching screens, controllerFactory() is called inside the current controller, with the next screen controller's name as an argument.
Constraints	None

Additional remarks	None

	DP3
Design pattern	Controller
Problem	A simple interface is needed to handle requests and interactions of the system
Solution	An abstract Controller class and 1 controller subclass for each main functionality of the system
Intended use	Each Controller handles its respective screen's functionalities and can pass information to other Controllers
Constraints	Controllers must have strict rules to avoid spreading errors to other parts of the system.
Additional remarks	None

Revised object diagram

Author(s): Tu, Thanh, Shivangi, Kyle



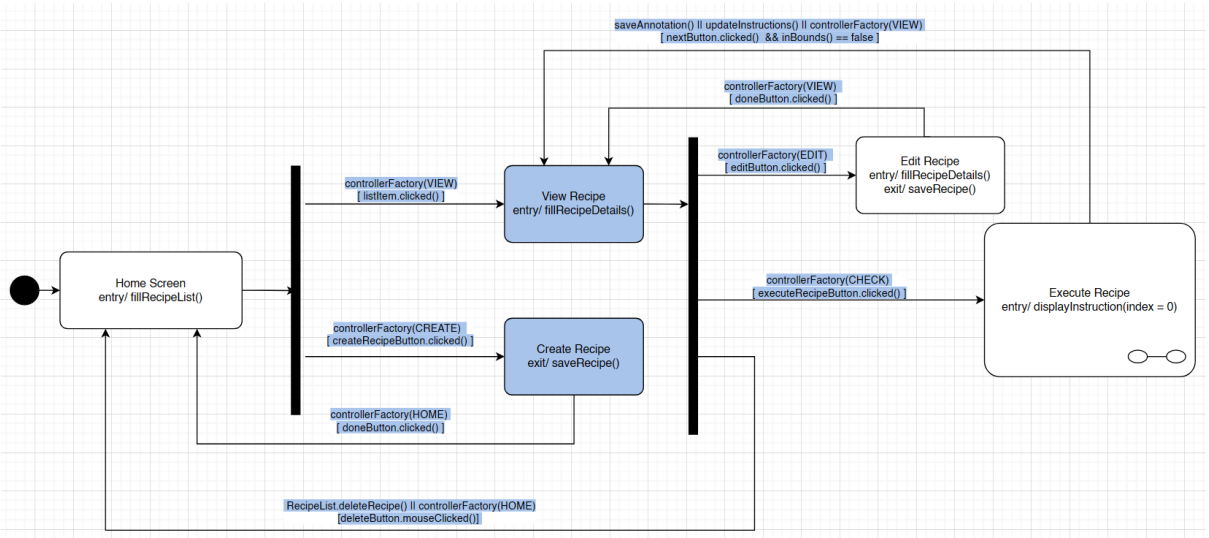
All the changes are marked in blue colour. We have started from the **HomeController** class, which shows our homepage.The other change from the previous diagram is the addition of

the **Instruction** classes.We have shown in this object diagram how each instruction is stored in the instruction class, to show the entire execution of the recipe. We have removed the Pantry, ShoppingList and IngredientList classes since we felt like it was making our system more complex.

We are showing a snapshot of a part where the user is looking for a recipe. It starts from the **HomeController** class which shows all the recipes from the RecipeList class. The **RecipeList** class contains one recipe called *Spaghetti*, which is stored in a json file. The recipe is obtained from the **Recipe** class . Each recipe contains certain ingredients and certain instructions, which are both handled in **Ingredient** and **Instruction** and classes respectively. So in our example the ingredients would be *oil, salt and spaghetti* and the instructions are *CookSpaghetti, Seasoning and Oil*. Once all the instructions are obtained, the execution of the recipe begins. The execution of the recipe happens in a step-by-step manner. This is then handed over to the **ExecutionController** class which obtains instructions from the **Instruction** class and executes it in step-by-step and shows it on the screen to the user where the users are able to annotate. Finally, the most important change in our diagram is that we have reduced the number of objects that represent our execution in order to make our diagram less complex.

Revised state machine diagrams

Author(s): Tu, Thanh, Shivangi, Kyle

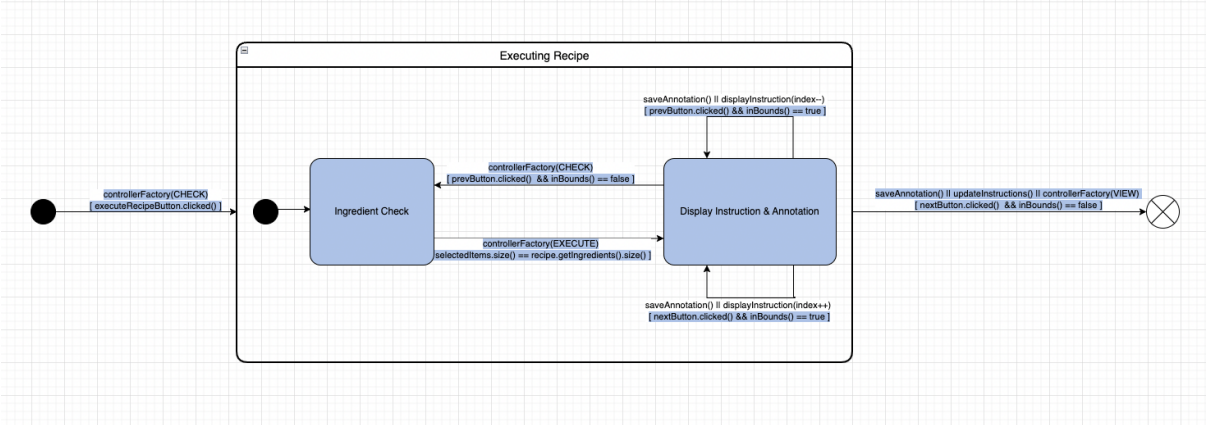


The changes are quite a bit from the previous state machine diagram. We have more events that we have added, since we have implemented the GUI as well. More specifically here we are representing the **Controller** class instead of the **RecipeList** class. All the changes are highlighted in blue. These changes give a more accurate representation of the internal states. In general our states have remained the same, however the way the transitions take place have been changed. First change is the transition from Home Screen, which has the pseudostate parallelization which goes to two other states *Create Recipe* and *View Recipe*, which are transitioned depending on the events that are triggered and if the guards are true

(controllerFactory(VIEW)[listItem.clicked()] or controllerFactory(CREATE)[createRecipeButton]). This is exactly how the class operates. Then the other change is that now from the *View Recipe* State we again have pseudostate parallelization which transitions to three different states *Edit Recipe*, *Execute Recipe*, and *Home Recipe* based on the events that are triggered that are listed below.

- *controllerFactory(EDIT) [editButton.clicked()],*
- *controllerFactory(CHECK) [executeRecipeButton.clicked()*
- *RecipeList.deleteRecipe() || controllerFactory(HOME)[deleteButton.mouseClicked]*

The other change we made is the state transition from *Edit Recipe* back to the *View Recipe* state.This can be achieved by the following event and guard *controllerFactory(VIEW) [doneButton.clicked()]*.This is a change we needed to make since we want the users to see the recipe that they made changes to, which is something we missed out in the previous diagram. The final change we made to this diagram again is the state transition from *Execute Recipe* to the *View Recipe* state. The reasoning is similar as the previous one, except that after the recipe is done executing the user should be taken back to the *View Recipe* state so the users can see their changes.



This diagram represents the class ControllerExecute class since this controls the execution of recipe part. The Execute Recipe state is a composite state which consists of two internal states *Ingredient Check* and *Display Intruction and Annotation* states (similar structure as earlier except we have changed the states and the events and guard that trigger the transition).

The Ingredients Check state prompts the user to select all of the recipe's ingredients before continuing. When the guard *[selectedItems.size() = recipe.getIngredients().size()]* is true, the event *controllerFactory(EXECUTE)* is triggered to transition to a new state of displaying instruction and annotation.

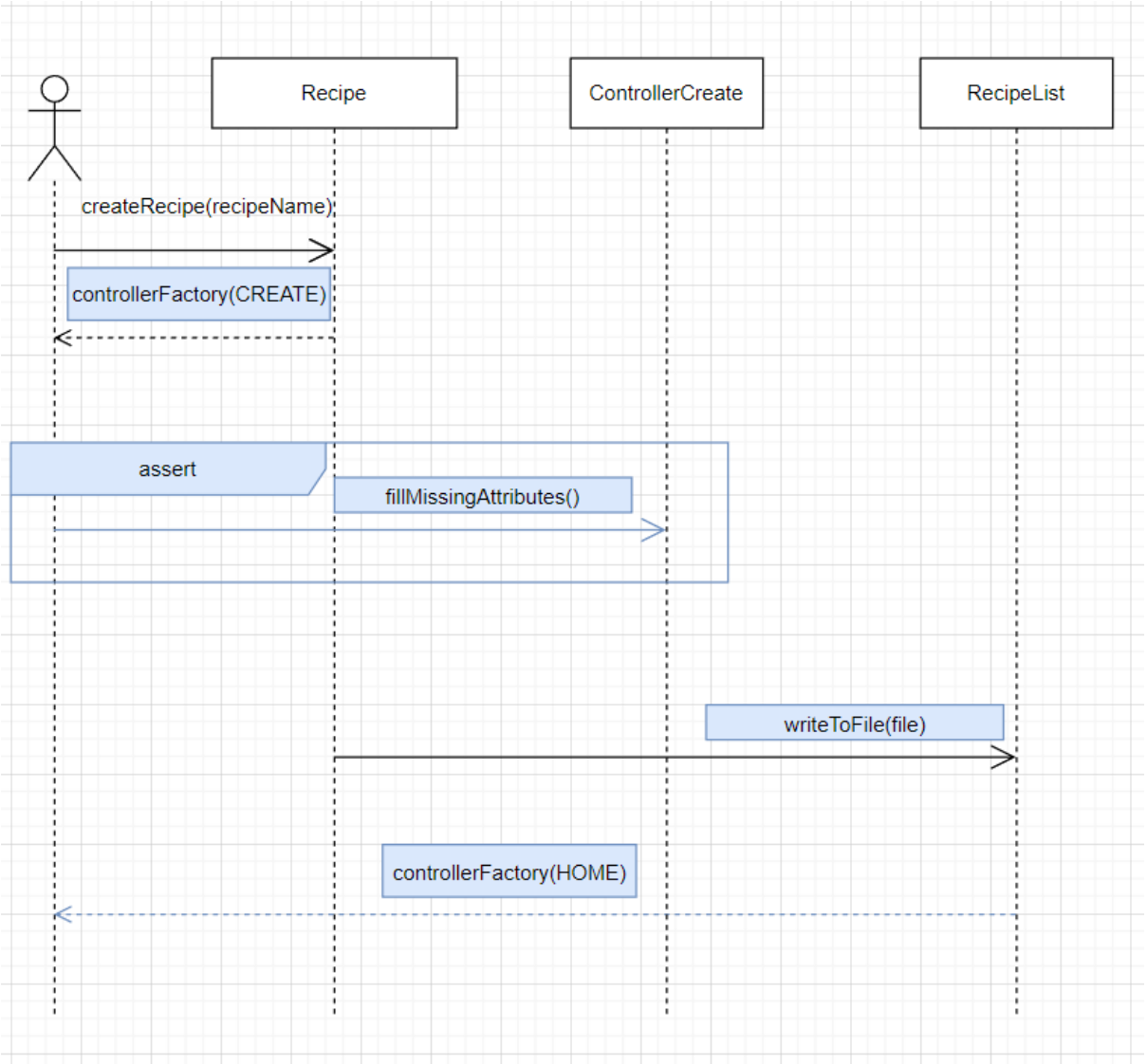
In this state, the first instruction is displayed immediately. The user can also switch to the previous or next instruction, checked by the guard *[prevButton.clicked() && inBounds() == true]* and *[nextButton.clicked() && inBounds() == true]*. inBounds() returns true if *[0 <= currentInstructionIndex < instructions.size()]* and returns false if otherwise.

When prevButton.clicked() and the system is already showing the first ingredient, the system will call controllerFactory(Check) and move back to Ingredient Check screen.

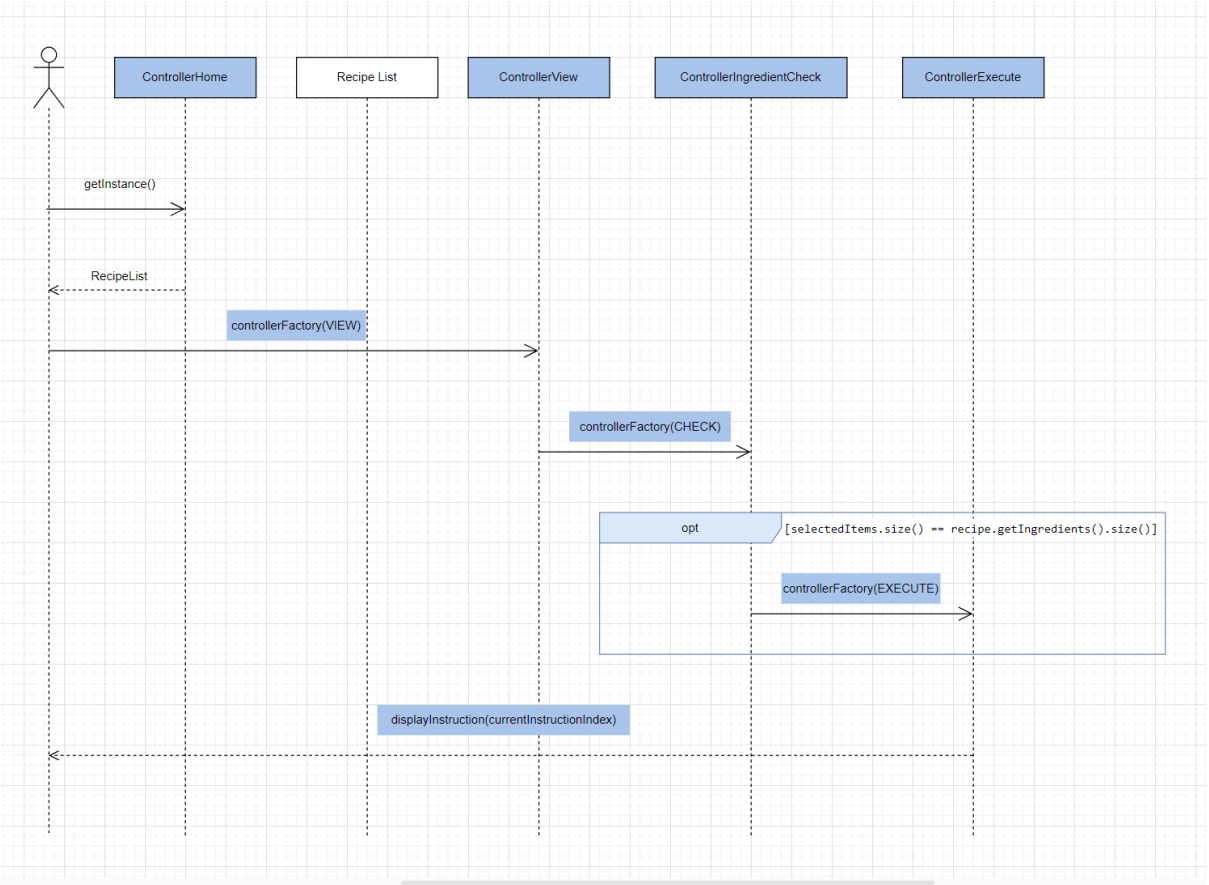
A significant change we made is the event and guard that trigger the termination state: when *[nextButton.clicked() && inBounds() == false]* guard is true, *controllerFactory(VIEW) and updateInstructions()* are called, and the user is brought back to the View Recipe screen, displaying the recipe with new instructions.

Revised sequence diagrams

Author(s): Tu, Thanh, Shivangi, Kyle



The main improvement in this sequence diagram is removing the loop fragment as we are not looping and prompting for each instruction/ingredient as initially planned. Instead, now the user gets an interface (the CreateController) where the user needs to insert all the necessary information for the corresponding recipe he/she wants to create. We also added an assert fragment where the user needs to fill in the missing attributes since all attributes must be filled in in order to create the recipe. Finally, we improved the messages and replies to conform with the operations specified in the class diagram.



In this sequence diagram there have been a lot of changes. Primarily we removed the pantry and shopping list class from the sequence diagrams since we did not implement these in the end. We also did not end up implementing the bonus filtering feature so, this has also been removed. Some key improvements have been the controller classes the user interacts with. At first the user is presented with the RecipeList by the ControllerHome. Then when the user clicks a recipe the controllerFactory shows the ControllerView where the user can edit,execute and view a recipe. The user clicks execute and the controllerFactory shows the ControllerIngredientCheck. Here we added an opt fragment that checks if we have all the ingredients necessary for the corresponding recipe. When all the ingredients have been checked off then the user is presented with the first instruction of the recipe.

Implementation

Author(s): Tu, Thanh, Shivangi, Kyle

Demo video (3 minutes): https://youtu.be/47aikYfCxr4

We did not have a fixed strategy when moving from the UML models to the implementations, as we implemented the system and modified the UML models in conjunction. By first brainstorming ideas for the diagrams, we were able to start building the system's basics fairly quickly. By implementing functionalities of the system's different modules, we were able to model their interactions more accurately in UML. We kept repeating this process until both the system and the UML models are complete.

To make the system as **independent from the recipe json files** as possible, we store recipes inside a Map data structure with each entry being <File, Recipe>. By doing this, a recipe can have a different name from its json file. For example, a recipe with the name “test recipe 2” can be stored inside the file abcdefg.json. This also means that recipes can be modified without changing the json file name.

To parse json to Recipe and the reverse, we used the Gson library to simplify the process. If any attribute of the recipe is missing or blank after parsing to Recipe, a placeholder value is generated using the fillMissingAttributes() method in the Recipe class.

To make sure that the recipe list is consistent throughout the system’s execution, we used the Singleton design pattern for the RecipeList class.

To create a simple GUI, we made use of JavaFX and its components.

Most of the system’s behaviour is handled inside the Controller class and its subclasses. There is one subclass for each main state of the system.

Each screen of the system (Home, Create, Edit, Execute, View) is a state, and they all have the following in common: a large title text on the top of the screen that can be changed depending on the state, and a back button that redirects the user to the previous screen.

On startup, **the Home Screen** contains:

- a ListView containing all the current recipes’ names stored in .json files inside the /recipes directory.
- Each ListItem has an ActionListener that redirects the user to the View Recipe screen when clicked.
- a Create Recipe button that redirects the user to the Create Recipe screen when clicked.
-

The Create Recipe Screen contains:

- 4 TextFields for the recipe’s name, description, time, and tags.
- 2 TextAreas for the recipe’s ingredients and instructions.
- a Done button at the bottom of the screen submits the user’s input for input validation and recipe saving, handled by the RecipeList class. These are the rules for the input fields:
    - All fields and areas have to be filled
    - Time has to be a Long (64-bit integer)
    - Tags are separated by commas
    - Ingredients and instructions are separated by newlines
    - Ingredient format is: name, quantity, unit.
        - Quantity has to be a Long (64-bit integer)
        - Unit is optional, but name and quantity are required
    - There are no restrictions on instructions

In case of an error or an invalid input, the error is thrown by the RecipeList class, and caught by the Controller class. The screen’s title is set to the error message to inform the

user. When all inputs are valid and a new recipe is created, it is written to a json file with the same name and stored in the /recipes directory.

The View Recipe Screen contains:

- 3 buttons on the right side of the screen: Edit, Execute, and Delete. When clicked on, they redirect users to their respective screens, **except for Delete, which directly deletes the recipe**.
- a 2x3 GridPane:
    - First column contains the Tags, Ingredients, and Instructions labels
    - Second column contains the corresponding contents. Ingredients and instructions are stored inside a ScrollPane to make sure the user can scroll vertically or horizontally in case there are too many characters or lines.

The Edit Recipe Screen has the same interface as the Create Recipe screen. Upon entry (from the View Recipe screen), the current recipe's details are populated to the appropriate fields, and the user can fill out new details. When the Done button is clicked, the current recipe is overwritten by a new recipe with the new details filled out by the user. **The json file name stays the same** (important, for independence from json).

When the Execute button is clicked, **the Ingredient Check screen** is displayed before the Execute screen. This screen contains:

- a Listview containing the recipe's ingredients and prompts the user to check (select) all the ingredients that they have before continuing to execute the recipe.

The system automatically transitions to the Execute Recipe screen when the number of selected items equals the number of ingredients in the recipe.

The Execute Recipe screen contains:

- a 2x2 GridPane:
    - First column contains the Instruction and Annotation labels
    - Second column contains the instruction text (displayed in a Label) and annotation (displayed in a TextArea, modifiable)
- prev/next button that displays the previous/next instruction and annotation.

When entering this screen, currInstructionIndex is set to 0 and the first instruction/annotation is immediately displayed. The user can edit the annotation inside the designated TextArea. When either button is clicked, the annotation is saved to the corresponding instruction, and the TextArea is cleared for the next or previous instruction. When the user reaches the last instruction, clicking the next button will redirect the user back to the View Recipe screen. This also applies to the first instruction and the prev button. Before this redirection, the recipe's instructions are updated to contain the new annotations.

Location of the main Java class: softwaredesign.Launcher
Location of the Jar file for directly executing the system: out/artifacts/software_design_vu_2020.jar

Time logs

https://docs.google.com/spreadsheets/d/1cG_WAq0Jk-Kv1ze1cUtDE-HXqOdwg1_pY8omEYkDVVk/edit?usp=sharing

Team number	99		
Member	**Activity**	**Week number**	**Hours**
Thanh	Implementation	6	2
Shivangi	Implementation	6	2
Kyle	Implementation	6	2
Tu	Implementation	6	2
Shivangi	Modify class diagram	7	2
Shivangi	Modify object diagram	7	2
Thanh	Modify object diagram	7	2
Thanh	Modify state machine dia	7	2
Shivangi	Modify state machine dia	7	1
Tu	Modify sequence diagrar	7	2
Kyle	Modify sequence diagrar	7	2
Kyle	Finish document	8	2
Shivangi	Finish document	8	2
Tu	Finish document	8	2
Thanh	Finish document	8	2
		TOTAL	25