Assignment 2

Team number: 49
Team members

Name	**Student Nr.**	**Email**
Taher Jarjanazi	2707386	t.jarjanazi@student.vu.nl
Moegiez Bhatti	2686748	m.a.bhatti@student.vu.nl
Adrian Andronache	2742960	a.andronache@student.vu.nl
Shantanu Jare	2738504	s.jare@student.vu.nl

Format: we establish the following formatting conventions when describing our models in this document: The name of each **class** is in bold, whereas the attributes, operations, and associations as underlined text, and *objects* are in italic.

Summary of changes from Assignment 1

Author(s): Taher & Adrian

- Slide 4 with header “Basic Functional Features” has been changed to “Mandatory Functional features” based on the received feedback as to give more clarity about the functional features will definitely be implemented.
- Slide 5 with header “Advanced Functional Features” has been swapped with slide 6 and changed the header to “Optional Functional Features” to better indicate the functional features that are optional and we are not obligated to implement but may be able to implement given we already finished with the mandatory functional features.
- Slide 6 with header “Advanced Functional Features continued” has been swapped with slide 5 and changed the header to “User Interface Functional Features”, also indicating that the command-line interface is mandatory to implement and the GUI will be extra UI that may be implemented after having finished the CLI.
- The features ID’s have been adjusted according to the new order of slides so the functional features overview flows better.

Class diagram

Author(s): Moegiez

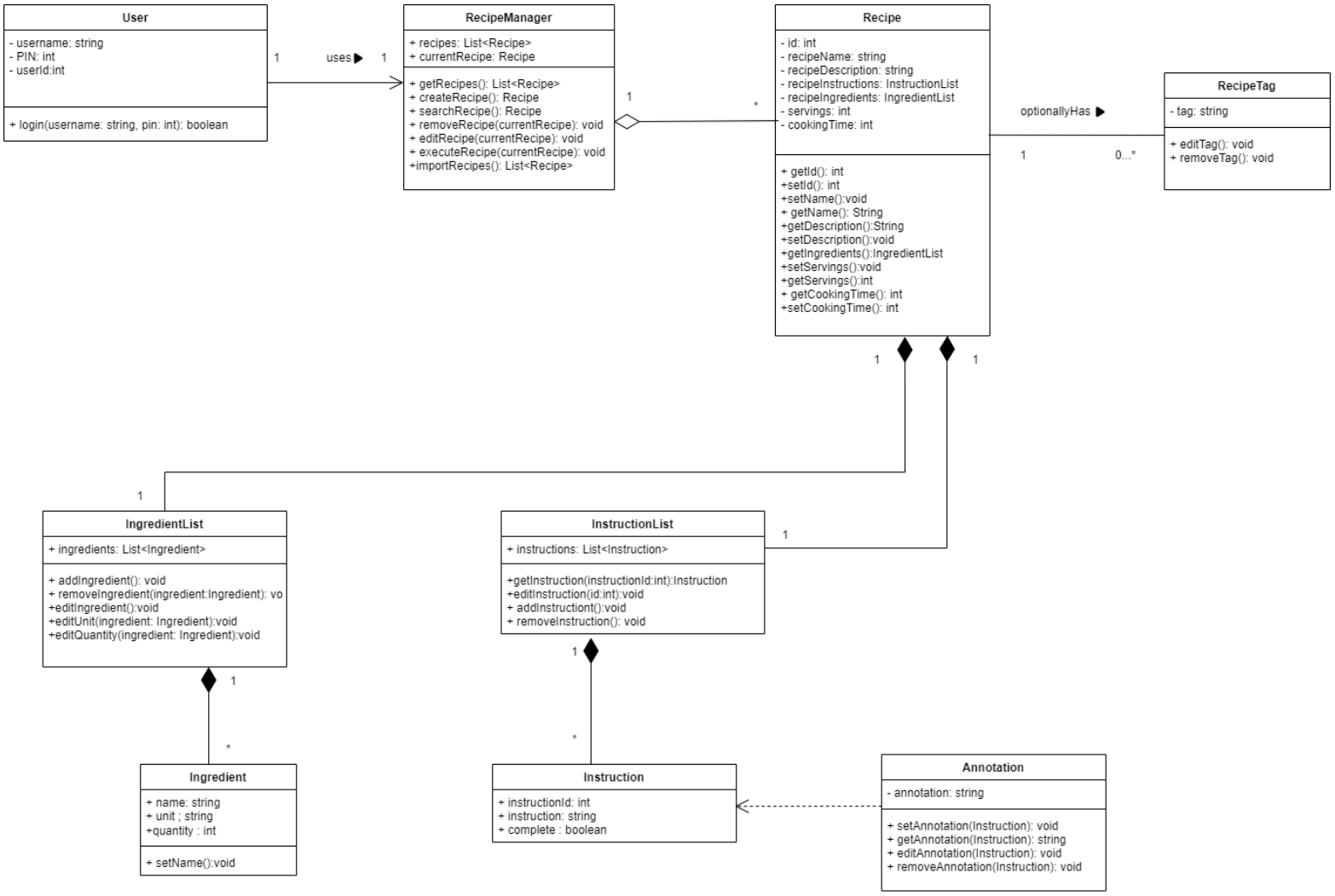

Figure 1: UML Class Diagram for the recipe management system

User Class

The **user class** represents the account information and login functionality of a user. The **user class** represents the user or entity that interacts with the system being modelled, which in this case is the recipe manager. The user class has several attributes; userName and pin. The userName attribute has a type string and represents the username. The pin attribute has a type int and represents the pin of a user, which is needed in order to login.

The **user class** has one method, which is the login operation. This method takes the userName and the pin and returns a boolean. If the boolean is true, the user can access the system

The **user class** is linked to the **recipeManager** class with an association relationship, represented by a solid line. This relationship portrays how the user interacts with the recipeManager class, which is also indicated by the name “uses”. The direction shows which class initiates the interaction, which in this case is the **user**. The relationship has a multiplicity of 1 for **user** and 1 for **recipeManager,** which means that at a specific instance, one user interacts with 1 recipe management system.

RecipeManager Class

A **recipeManager** class in a UML class diagram for a recipe management system would be a class responsible for managing recipes. It acts as a collection of recipes which the **user** can interact with.

The **recipeManager** class has two attributes: recipes and currentRecipe. The recipes attribute is a list consisting of recipe objects. The attribute currentRecipe is a reference to a recipe object that is currently selected.

The **recipeManger** has several methods. The most important ones are listed below:

- getRecipes(): List<Recipe>
  This method gets all the recipes in the system and returns a list of all the recipes.
- createRecipe(): Recipe
  This method allows the user to create a new recipe and add it to the system. It takes no arguments and returns a recipe object.
- editRecipe(currentRecipe): void
  This method takes a recipe object as an argument and has access to all its attributes. In this manner the recipe can be updated.
- searchRecipe(keyword): Recipe
  This method takes a string as an argument and returns a list of recipe object that match the keyword in name, ingredient or tag.
- executeRecipe(currentRecipe): void
  This method takes a recipe object as an argument and uses its attributes and methods to execute the recipe in a step by step manner.
- importRecipes(): recipes
  method that can import json recipes. It returns a list of recipes.
- removeRecipe(currentRecipe): void
  This method removes the currentRecipe.

The relationship between the **recipeManager** class and the **recipe** class is an aggregation relationship, which means that the **recipeManager** contains one or more **recipe** classes, but they can exist independently. For example, a recipeManager can have many recipes, but each recipe can exist without being part of any recipeManager. The multiplicity of this

relationship is 1 recipe for many recipes, which means that 1 recipe manager can have many recipes.

Recipe Class

In this system a **recipe** class represents a single recipe and contains all kinds of information about this recipe. There are several attributes present which describe the recipe: id,name, description, list of ingredients, list of instructions, servings and cookingTime. One notable attribute is description, which is of the type string and contains a short description of the recipe. Other important attributes are the recipeInstructions and the recipeIngredients, which are both lists consisting of **ingredient** class objects and **instruction** class objects respectively. This class also contains several methods, these are setters and getters for the attributes.

The **recipe** class is related to the **recipetag** with an association relationship. It has a multiplicity of 1, in relation to the multiplicity of the RecipeTag which has a multiplicity of 0 to many. This means that one recipe can have zero to many tags.

The **recipe** class is also related to the ingredientList and the instructionList class with a composition relationship. A composition relationship in a UML class diagram indicates that an object is composed of other objects and is responsible for their existence. Deleting a recipe would result in these two classes being deleted as well. The Recipe class has a multiplicity of 1 and both the lists also have a multiplicity of 1. This indicates that every Recipe has 1 ingredientList and 1 instructionList.

RecipeTag Class

A **recipeTag** class is a class that represents a label that can be added to a **recipe**. It has an attribute called tag, which is of the type string and contains a short description of the label. A few examples are; vegetarian, vegan, gluten-free. This tag can be used to search for specific recipes. This class also has two methods: editTag() and removeTag(), which both return void.

Ingredients List Class

The **IngredientList** class represents a list of ingredients required to make a particular recipe. The **IngredientList** class has two attributes: ingredients and recipeId. The ingredients attribute is a list of Ingredient objects. The **IngredientList** class has five methods: addIngredient, removeIngredient, editIngredient, editUnit. The addIngredient method adds an ingredient to the list. The removeIngredient method removes an ingredient from the list. The editIngredient method modifies an existing ingredient in the list. The editUnit and editQuantity methods take an ingredient class as a parameter and change the unit and quantity respectively. The **IngredientList** class is related to the **ingredient** class with a composition relationship. It has a multiplicity of 1 and the **ingredient** class has a multiplicity of many. This means that one **ingredientList** can have many ingredients.

Ingredient Class

An **ingredient** class represents a single ingredient that can be used in a recipe. It has attributes such as name, unit and quantity. The unit and quantity attributes can be accessed and changed in the ingredientList class. It has one method, the setName function, in order to set the name of the ingredient.

InstructionList Class

The **InstructionList** class represents a list of instructions that can be followed to prepare a recipe. The **InstructionList** class has several attributes, which is instructions which is a list of Instruction objects. The **InstructionList** class has several methods, such as getInstruction, editInstruction, editInstruction, addInstruction, and removeInstruction. The getInstruction method returns an instruction from the list according to the instructionId. The editInstruction method modifies an existing instruction in the list. The addInstruction method adds a new instruction to the list and returns the updated list The removeInstruction method deletes an instruction from the list and returns the updated list. The **instructionList** Class is related to the instruction class with a composition relationship. The **instructionList** class has a multiplicity of 1, the instruction class of many. This means that 1 **instructionList** can consist of many instructions.

Instruction Class

The **Instruction** class represents a single instruction that can be part of an **InstructionList**. The Instruction class has attributes such as id, instruction, and complete. The id attribute is an integer that identifies the instruction. The instruction attribute is a string that describes the instruction. The complete attribute is a boolean that indicates whether the instruction has been completed or not. This attribute is checked in order to move on to the next instruction in the list. The **instruction** Class is related to the **annotation** class with a dependency relationship. A dependency relationship is a directed, supplier-client relationship that shows that some element requires or depends on another element for specification or implementation. In this case, a class that uses another class as a parameter. The **annotation class** uses instruction as a parameter in its methods, and therefore is dependent.

Annotation Class

The **annotation** class represents a note or comment that can be attached to an instruction. It has an annotation attribute, which is a string. It also has three methods, in order to set, get, remove and edit an annotation.

Object diagram

Author(s): Shantanu & Moegiez

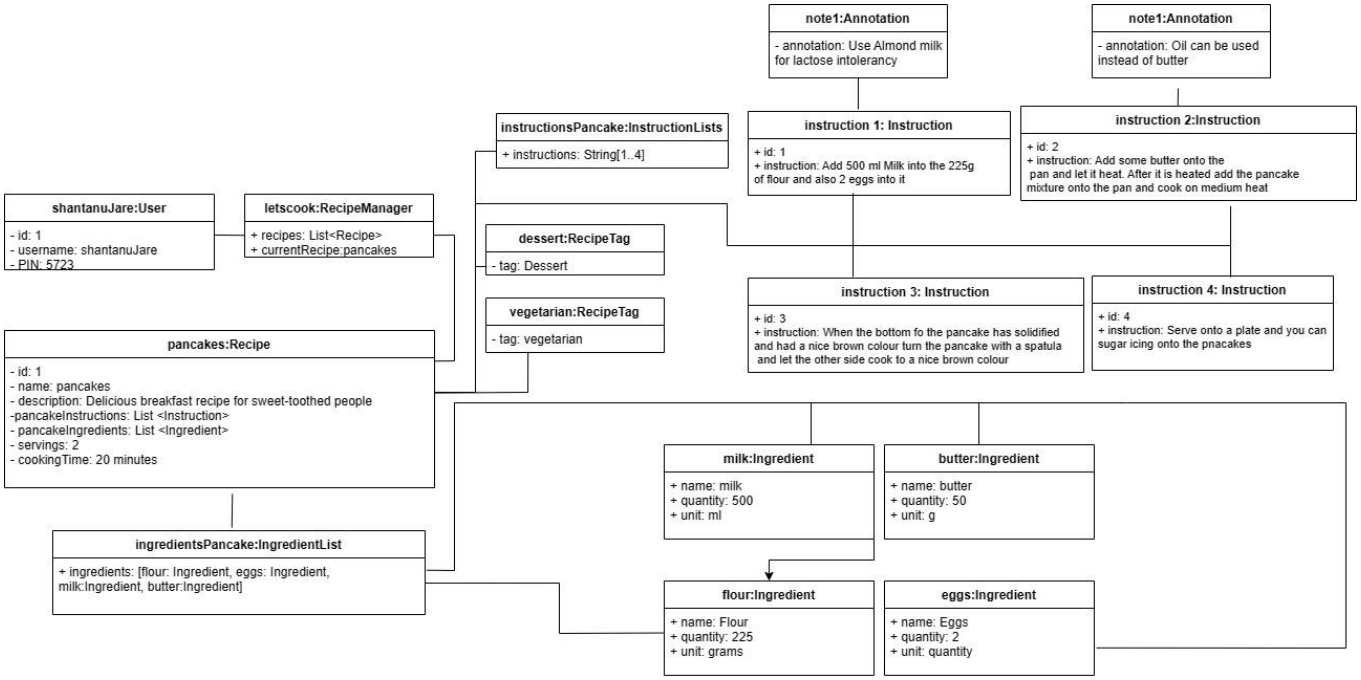

Figure 2: UML object diagram for the recipe management system

User Object(shantanuJare:User):
The instance of the user class in the object diagram contains the id, userName and the pin of the user shantanuJare, it contains the following fields id-1, username-shantanuJare and PIN-5732

RecipeManager Object(letscook:Recipe Manager):
The instance of the RecipeManager class in this instance is called ‘letscook’. It has a list of recipes, only one in this case is the pancake. It contains the currentRecipe member which is the pancake recipe instance.

Recipe Object(pancake:Recipe):

The pancake object is the instance of the recipe class. It has the following members, id-1, name-pancakes, description, IngredientList, InstructionList, serving size which is the amount of people the recipe can feed and the cooking time.

InstructionList Object(pancakeInstructions:InstructionLists):

The pancakeInstructions instance contains a list of the instructions of the process of cooking the pancake

I**ngredientList Object(pancakeIngredients:IngredientLists):**

The pancakeIngredients instance contains a list of the ingredients used to cook the pancake.

Ingredient Object(butter:Ingredient):

The butter instance of the ingredient class contains the following members, name:butter, quantity:2 and the unit of the ingredient such as grams, millilitres etc. In this case it is grams.

Instruction Object(instruction1:Instruction):

The instruction1 instance contains the first instruction of the recipe to cook the pancake. It is to add 500 ml milk into 225 g of flour and 2 eggs into the mixture.

State machine diagrams

Author(s): Taher, Adrian & Shantanu

Diagram 1: Recipe execution

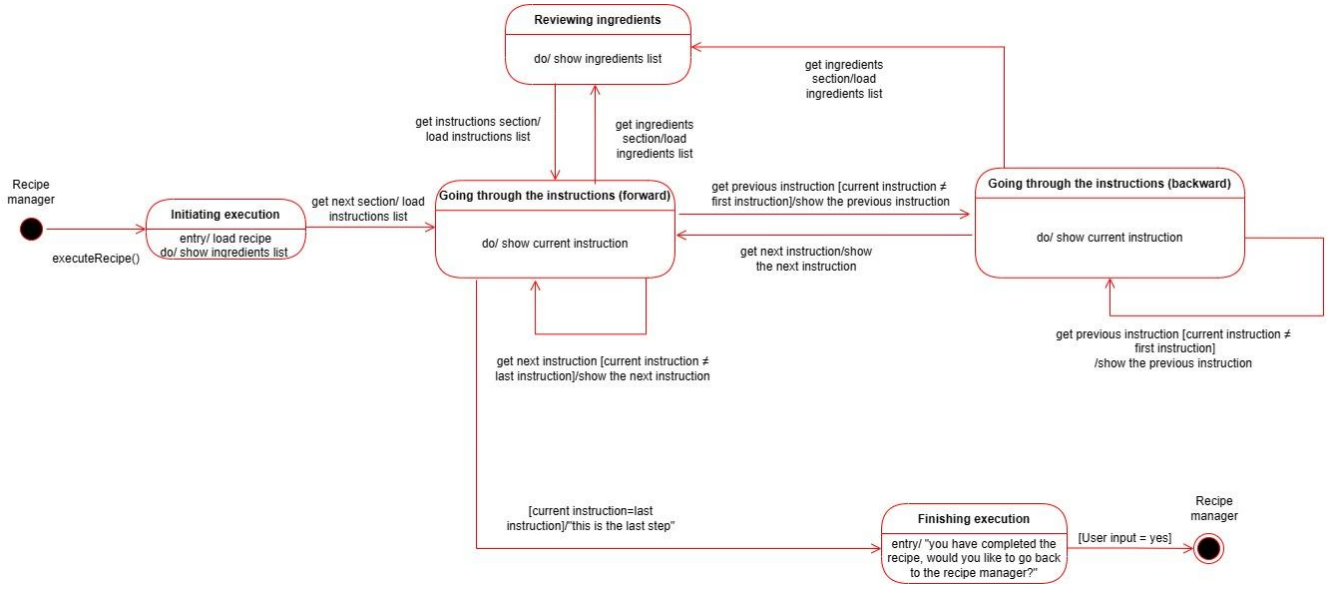

Figure 3: UML state machine diagram for internal behaviour of the RecipeManager class when executing a recipe

This state machine diagram shows the execution of a recipe, which is a central part of the recipe management system. Specifically, the state machine diagram above describes the executeRecipe() method in the **RecipeManager** class.

The state machine starts and ends on the **RecipeManager** as start and final states, respectively. The process starts with the execution of the executeRecipe() method and we get to the “Initiating execution” state where, upon entry, the recipe to be executed is loaded and the ingredients list is shown. When the event for going to the next section in the execution arrives, then the state instructions list is loaded and the state switches to “Going through the instructions (forward)” state. Here the state machine shows the current instruction, which is the first instruction in the instructions list after coming from the “Initiating execution” state. At this state there are multiple events that can take place. The user can go to the next instruction so an internal state transition will happen, unless the current instruction is the last instruction, where the next instruction will be shown. A user may also want to check the ingredients list, so an event hereof will load the ingredients list before

transitioning to the “Reviewing Ingredients” state where the ingredients list will be shown. A user may want to go back to the previous instruction so, unless the current instruction is the first instruction, the previous instruction will be shown and the state will be switched to the state “Going through the instructions (backward)”. There, the user can still go to previous instructions so an internal state transition will happen, unless the current instruction is the first instruction, where the previous instruction will be shown. Also from the “Going through the instructions (backward)” state, a user may also want to check the ingredients list, so an event hereof will load the ingredients list before transitioning to the “Reviewing Ingredients” state where the ingredients list will be shown. From the “Going through the instructions (backward)” state, an event to get the next instruction can take place if the user now wants to go forward with the instructions, so the next instruction will be shown before transitioning to the “Going through the instructions (forward)” state. In the “Reviewing Ingredients” state, there is a single event, namely to get the instructions section, where the instructions list will be loaded and the state will transition to the “Going through the instructions (forward)” state, where the current instruction will be shown. The reason why that event switches the state to the “Going through the instructions (forward)” state and not the “Going through the instructions (backward)” state is because most likely when a user has doubts about the ingredients and they are cleared once after seeing the ingredients list, then it is only natural for them go forward with the recipe execution and thus the instructions. The advantage of this design is that there are less state transitions and the forward flow through the instructions is encouraged. On the other hand, the system cannot switch from the “Reviewing Ingredients” state to the “Going through the instructions (backward)” state immediately.

Finally, from the “Going through the instructions (forward)” state, if the current instruction is the last instruction in the instructions list, then a text to notify the user of that will be shown and the state will switch to the “Finishing execution” state. There, upon entry, another text will be shown to notify the user they have successfully completed the recipe and they will be asked if they want to go back to the recipe manager. If the user inputs ‘yes’ then the state switches to the final state of the state machine diagram, which is back to the **RecipeManager** class.

Diagram 2: searchRecipe

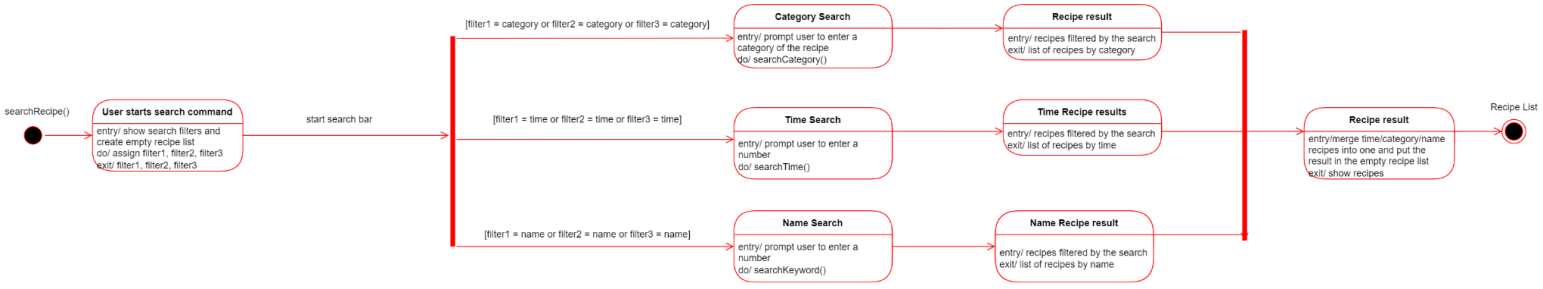

Figure 4: UML state machine diagram for internal behaviour of the RecipeManager class when searching a recipe

We built a state machine diagram for explaining the searchRecipe method of **RecipeManager** class for the fact that it is one of the most important features of our project. The state machine starts with the execution of the searchRecipe() method which lets the user search for a specific recipe or a group of specific recipes.

We arrive at the “User starts command search” state where at entry it will create an empty list for the recipes to be found after search and to show the user the search filters that they can use. The user has 3 filters to choose from which are category, name and time and this state will save the choices of the filters in the order they were selected in the variables filter1, filter2 and filter3. At exit, the state will send the values of the filters. The next state machines will start in the event of the user starting the search bar after selecting their desired filters. After the event happens there would be a decision node that would have 3 guards, one for each filter. The guards for the filters are the same and they check if the user has selected that specific filter by verifying if one of the 3 values filter1, filter2 and filter3 has the specific filter as an attribute. The next 3 states are the same aside from the fact that they filter the recipes differently with the 3 filters mentioned at the start. For each one of the three states, at entry it will prompt the user to type the category, name or time of their desired recipe and it will start searching through the recipes using its respective methods searchCategory(), searchTime() and searchKeyword().

The 3 next states Recipe Result, Time Recipe Result and Name Recipe Result at entry will take the recipes found by their respective search and put them in a list at exit. The last state will take all the lists from the 3 previous states and at entry it will merge the 3 lists into one by selecting the recipes that appear in all of the selected filters.

The merged recipes will be put in the empty list we created in the “User starts command search” state and at exit it shows the recipe list. The end state is a screen where the recipes found will be shown in a list like manner.

Diagram 3: updateRecipe:

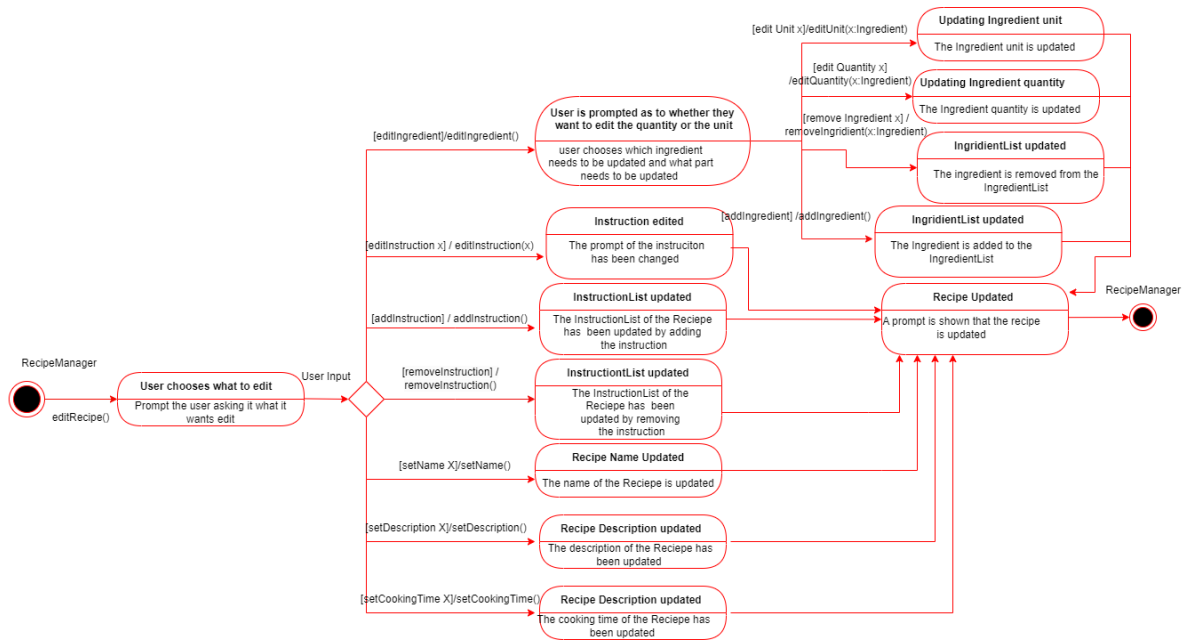

Figure 5: UML state machine diagram for internal behaviour of the RecipeManager class when editing a recipe

This state machine diagram explains the flow of the editRecipe() function of the **RecipeManager class.** The user chooses to edit a particular recipe and then the user is prompted as to what they exactly want to edit. The user can then choose to edit the recipe. This will call the editRecipe() function. The editRecipe() function will use the recipe stored in the current recipe variable of the **RecipeManager class**. The user can then choose what they want to edit, an ingredient, an instruction, or the name, cookingTime or the description. The user inputs a command that they exactly want to edit. In the state diagram this is highlighted by the first decision node which decides which branch the code will take.
For e.g if they input edit Ingredient then it will call the editIngredient() function of the **IngredientList class** The user will then have to specify which ingredient they want to edit and also what they want to edit. The user will then be prompted what they want to edit about the ingredient. The input they provide will decide which branch the code takes.
for eg: to add, remove or edit the quantity or unit of the ingredient. In the case that the user inputs the command removeIngredient x then the removeIngredient() function of the **IngredientList class** is called and the ingredient when found in the list is removed. The user can also additionally enter the command editQuantity which will call the editQuantity() function of the **IngredientListClass.** The user will then be prompted to choose what they want to edit about the Ingredient. This is highlighted by the decision node of the diagram on the top left.

The *instructionList* object or the list of instructions can be edited as well. This can be done by the addInstruction or removeInstruction command. Additionally an instruction at a particular index can be also edited. The command edit Instruction x is input by the user where x is the index of the instruction for e.g. if the user inputs 2 the 2nd instruction will be updated according to the input of the user.

Lastly, after the user has chosen what they want to change, these changes are made in the *InstructionList/IngredientList* objects of the recipe instance or the recipe instance as a whole, the recipe is updated in the recipeList in the RecipeManager object. This will also then lead to the recipeList being updated with the changes in the database.

Sequence diagrams

Author(s): Taher & Adrian

Sequence Diagram 1: Recipe Creation

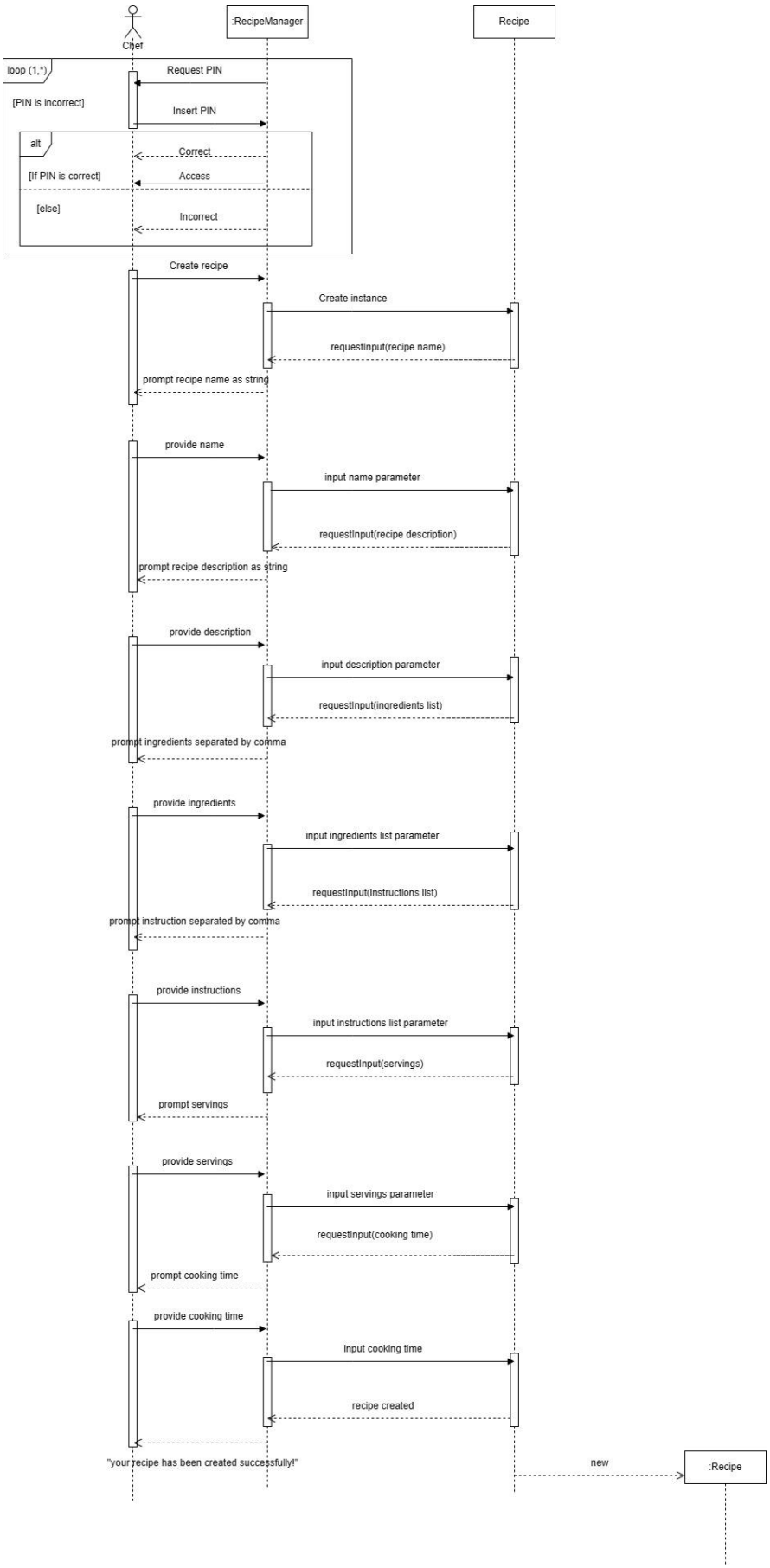

Figure 6: UML sequence diagram for recipe creation

We decided to create a sequence diagram for the interactions between the user and different parts of a system when the user creates a recipe. For this diagram we have an actor and 3 objects which are *RecipeManager, Recipe and new Recipe instance*.

The first interaction the actor has with the system is a PIN check with the *RecipeManager***.** The *RecipeManager* asks the user to insert the PIN and using an alternative frame we designed two scenarios, one for when the PIN is correct, the user gets access to the RecipeManager and one where the PIN is incorrect, the user gets a message that the PIN was incorrect, they don't get access to the *RecipeManager* and it enters a loop until the PIN is correct.

After getting access to the *RecipeManager* the user, here a chef, can send a synchronous message to the *RecipeManager* object, which will in turn send another synchronous message to the *Recipe* object to create an instance of *Recipe*. From there, the *Recipe* object will respond with the requestInput function where its parameter will be the information needed for that step of the recipe creation process. So first the recipe name is requested from the *RecipeManager* and a prompt will be shown to the user to input that information, which is a string in this case. Once the name is provided by the user, then the parameter is input and the name attribute is filled in. The same process is repeated for requesting the recipe description, ingredients list, instructions list, servings, and cooking time. When all the necessary information has been entered, the *Recipe* object responds to the *RecipeManager* object that the recipe was created and the *RecipeManager* will show the user a string confirming that the recipe has been created successfully. Lastly, the new *Recipe* instance is created.

Sequence Diagram 2: Searching a Recipe

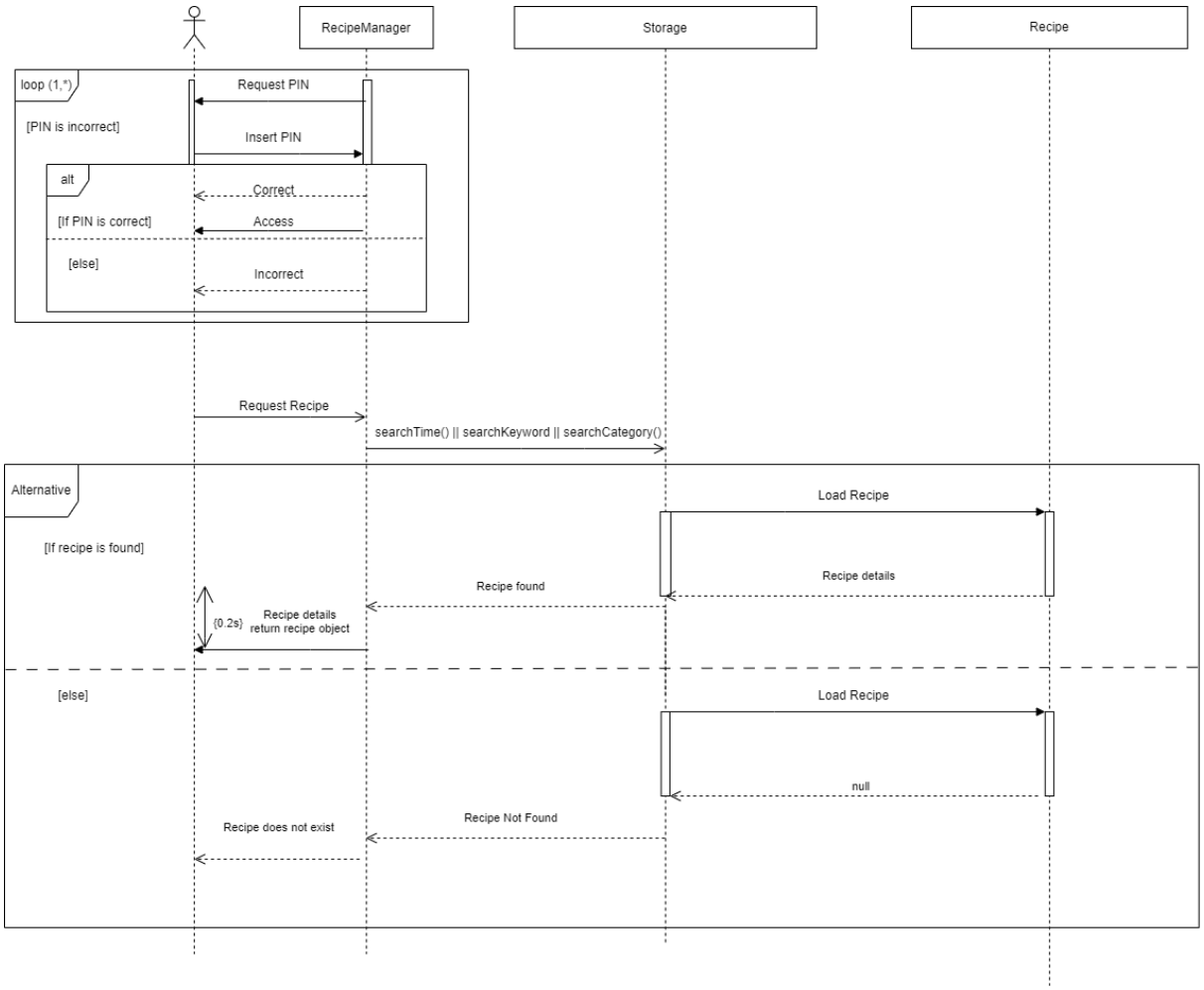

Figure 7: UML sequence diagram for searching a recipe

We decided to create a sequence diagram for the interactions between the user and different parts of a system when the user searches for a specific recipe. For this diagram we have an actor and 3 objects which are *RecipeManager, Storage and Recipe*.

The first interaction the actor has with the system is a PIN check with the *RecipeManager***.** The *RecipeManager* asks the user to insert the PIN and using an alternative frame we designed two scenarios, one for when the PIN is correct, the user gets access to the RecipeManager and one where the PIN is incorrect, the user gets a message that the PIN was incorrect, they don't get access to the *RecipeManager* and it enters a loop until the PIN is correct.

After getting access to the *RecipeManager* the user can search any type of recipe and the RecipeManager takes the information from the user and starts searching the *Storage* for recipes that fit the user's search criteria. For the search we also used an alternative frame with the condition "if the recipe is found", if the condition is true the *Storage* will communicate with that recipe through the *Recipe* object and then it will ask to transfer its details, the storage will send back a message to the *RecipeManager* to tell it that the recipe has been found and the *RecipeManager* will provide the details of the recipe to the user. If the condition is not fulfilled, the storage will not get anything from the recipe object for the fact that there isn't a recipe with those search criterias, thus the *Storage* will send message to the *RecipeManager* that the recipe wasn't found and the *RecipeManager* will tell the user that recipe with those criteria doesn't exist.

Time logs

A	B	C	D	E	F
Team number	49				
Member	**Activity**	date	**Week number**	**Hours**	**Progress**
Taher	define classes	19/02/2023	2	1	Done
Moegiez	define classes	19/02/2023	2	1	Done
Adrian	define classes	19/02/2023	2	1	Done
Shantanu	define classes	19/02/2023	2	1	Done
Taher	discuss classes	21/02/2023	3	2	Done
Moegiez	discuss classes	21/02/2023	3	1	Done
Adrian	discuss classes	21/02/2023	3	2	Done
Shantanu	discuss classes	21/02/2023	3	2	Done
Moegiez	work on class diagram	23/02/2023	3	5	Done
Adrian	work on class diagram	23/02/2023	3	6	Done
Shantanu	work on class diagram	23/02/2023	3	3,5	Done
Taher	work on class diagram	23/02/2023	3	2,5	Done
Moegiez	refining class diagram	25/02/2023	3	3	Done
Adrian	refining class diagram	25/02/2023	3	1,5	Done
Shantanu	refining class diagram	25/02/2023	3	1	Done
Taher	refining class diagram	25/02/2023	3	1,5	Done
Moegiez	working on object diagram	26/02/2023	3	2	Done
Adrian	working on statemachine diagram	26/02/2023	3	2	Done
Shantanu	working on object diagram	26/02/2023	3	1	Done
Taher	working on statemachine diagram	26/02/2023	3	2	Done
Moegiez	working on object diagram	27/02/2023	4	2	Done
Adrian	working on statemachine diagram	28/02/2023	4	2	Done
Shantanu	working on object diagram	01/03/2023	4	2	Done
Taher	working on statemachine diagram	02/03/2023	4	2	Done
Moegiez	setting up github for coding	07/03/2023	5	2	Done
Adrian	setting up github for coding	07/03/2023	5	1	Done
Shantanu	setting up github for coding	07/03/2023	5	1	Done
Taher	setting up github for coding	07/03/2023	5	3	Done
Moegiez	Finalizing assignment 2	09/03/2023	5	4	Done
Adrian	Finalizing assignment 2	09/03/2023	5	4	Done
Shantanu	Finalizing assignment 2	09/03/2023	5	4	Done
Taher	Finalizing assignment 2	09/03/2023	5	4	Done
			TOTAL	57	