Assignment 3

Team number: 49
Team members

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

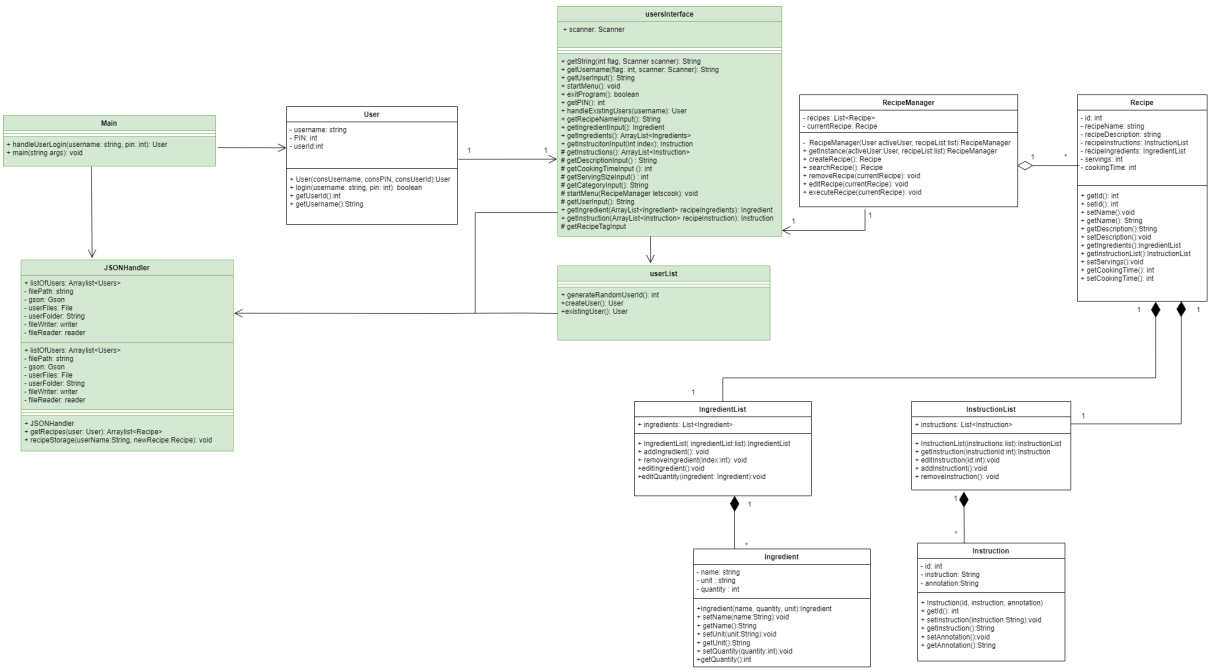
Summary of changes from Assignment 2

Author(s): Adrian

- Deleted the state machine for recipeSearch and replaced with one for the RecipeManager class
- Changed one object from the sequence diagram for searchRecipe(), Storage(doesn't exist) to JsonHandler
- Added the following classes to the class diagram: JsonHandler, UserList, Main, UserInterface
- Removed the RecipeTag and the annotation classes.
- Deleted the state machine diagram for editRecipe

Revised class diagram

Author(s): Moegiez

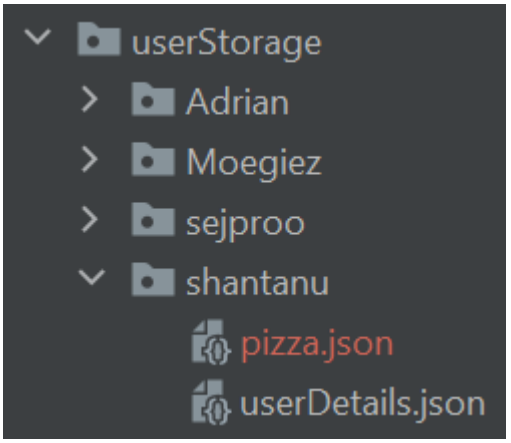


The class diagram has changed quite a bit from the last time. These have been summarised in the paragraphs below,

There have been three new classes that have been added to the diagram, The UserList class , JSONHandler class, and the UserInterface class. The rationale behind adding these classes is that the UserList class contains some auxiliary functions to process users such as generating a userID and creating a new user and checking if a user exists already with the same name.

JSON Handler class:

The JSONHandler class contains functions to access and interact with the JSON files where userDetails such as PIN, username are persisted.The JSON files of the users and their recipes is stored using the following structure.



The locally stored userStorage folder contains a list of users and their folders. Each of the folders contains the json files which contains important userDetails and also the recipes unique to each user.

userInterface class:

The userInterface class contains the auxiliary functions that we used for the user to interact with the recipeManager using the command line to create,execute,edit and remove a recipe. The class contains functions that take input from the user when they want for e.g. create a recipe, such as taking input of the list of instructions, ingredients, the name, etc. These functions have been used by the RecipeManager class and other classes such as the IngredientList and InstructionList class to enable the above mentioned functions.

userList class:

The userList class contains some auxiliary functions that enable the login process of the user. It contains the following methods,

- createUser()
  This function enables the user to create his/her own userProfile in the program.
- existingUser()
  This function checks whether there exists a user with the same username that the new user has entered.
- genrateRandomUserID()
  This generated a unique randomUserId between 0-100 to assign to a new user that in the process of creating his/her user profile

Apart from the changes mentioned above the pre-existing classes have been modified as well such as the RecipeManager class where we removed the importRecipes() and getRecipes functions and added the RecipeManager() constructor and the getInstance function. The getInstance() function ensures there is a single instance of recipeManager throughout the program obeying the singleton design pattern.In the Recipe class the only thing that was changed was that the recipeTag variable was changed from type recipeTag to String and the Recipe Tag class deleted. The only method that was added was the getInstructionList() method.

The next change that was made was the annotation class was deleted and the type of the annotation method was changed from annotation to String. Constructors were added to the InstructionList, Instruction, IngredientList and Ingredient classes and apart from that a few missing getter and setter methods have been added.

Application of design patterns

Author(s): Taher & Moegiez

	DP1
Design pattern	Singleton Design pattern
Problem	there is a risk of creating multiple instances of a class, which can

	lead to several issues in a software system. For example: - Inconsistent state: When multiple instances of a class exist, there is a risk of inconsistency in the state of the objects. This can cause problems in the system, such as incorrect output, unexpected behavior, and crashes. - Wasted resources: When multiple instances of a class are created, they can consume unnecessary system resources such as memory and CPU time. This can cause performance issues and impact the system's overall efficiency. - Difficulty in maintaining the code: When multiple instances of a class exist, it can be challenging to maintain the code, especially when multiple developers are working on the same codebase. This can lead to bugs, regressions, and difficulties in debugging. - Lack of global access: Without a Singleton design pattern, it can be challenging to ensure global access to a particular object in the system. This can make it difficult to share resources between different parts of the system and can hinder collaboration between different teams working on the same project.
Solution	The Singleton design pattern offers a solution for ensuring that a class has only one instance in a program, and that this instance is easily accessible to all clients. The solution involves defining a static instance variable in the class, which holds the single instance of the class, and making the constructor of the class private, which prevents external code from creating new instances. Instead, a public static method is provided to access the single instance of the class, which creates the instance if it does not exist and returns it to the caller. This allows all clients of the class to use the same instance, ensuring consistency and avoiding unnecessary duplication of resources. This is implemented in the RecipeManager class in this exact manner.
Intended use	Implementing the Singleton design pattern for a RecipeManager class in a recipe management system ensures that there is only one instance of the class created for each user, which helps to avoid inconsistencies and duplication of data. It ensures that the RecipeManager instance is accessible globally, making it easier to manage the user's recipes and ensuring that they are available whenever needed. It also ensures that the RecipeManager class is easy to maintain and modify, making it a valuable design pattern for

	recipe management systems.
Constraints	No constraints needed
Additional remarks	

	DP2
Design pattern	Iterator Design Pattern
Problem	The Iterator design pattern solves the problem of iterating through a collection of items, in our case the collection of ingredients of a recipe, without exposing the underlying representation of that collection.
Solution	The IngredientList class implements the Iterable interface for the Ingredient object. The Iterator interface then will be responsible for iterating through the Ingredient objects within the IngredientList.
Intended use	At runtime, the Iterator design pattern will be implemented when iterating through the ingredients in the IngredientsList object. Specifically, during the recipe execution the ingredients will be shown to the user as a first view before going through the instructions. Also, when going through the instructions, the user can view the ingredients and then the ingredients will be iterated through and printed to the console.
Constraints	No constraints
Additional remarks	

Revised object diagram

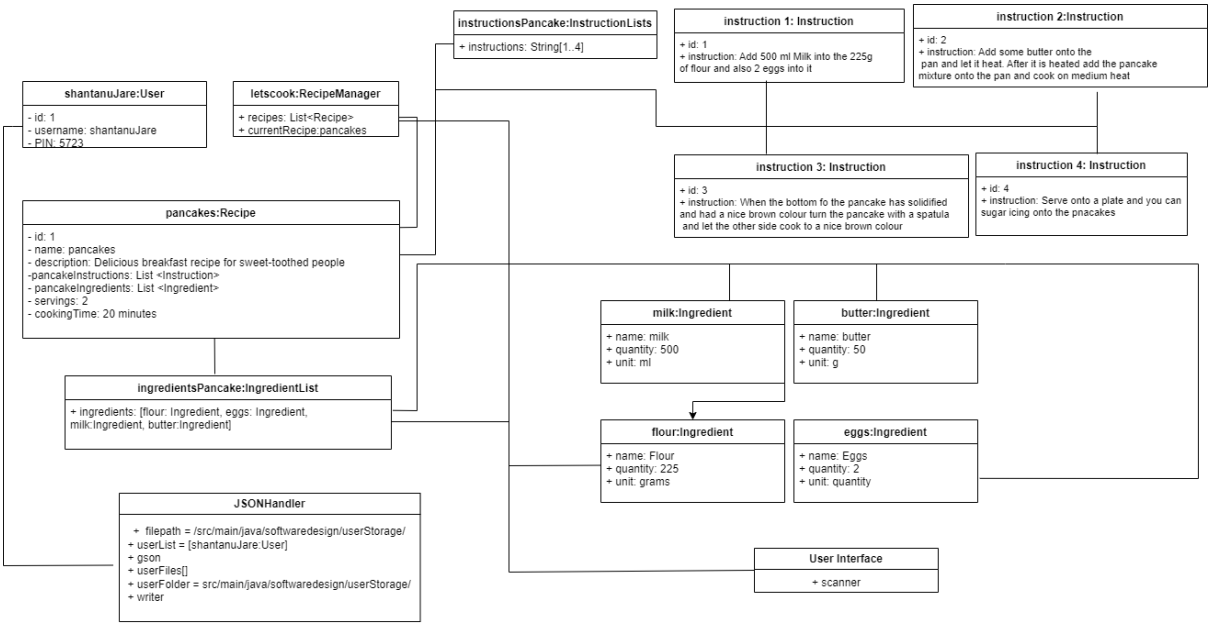
Author(s): Shantanu

The Object Diagram has been revised and the instances of the new classes added to the class diagram have been added to the object diagram with their appropriate methods.There have been certain instances of classes such as the recipeTag and the annotation that have been removed as these were not included in the class diagram. The instances of the userList class, JSONHandler class, and userInterface class The userList class doesnt contain any methods and therefore it isnt shown in the object diagram.

JSON Handler instance:

It contains the gson instance of the google.gson class that has member function to enable the conversion of the java instance into a gson string for it to be written to the JSON files. It also contains the file instance of the json files to interact with them.It also contains the list of the users that are saved in the json files as well to enable the login process.

userInterface instance:
It only contains the scanner attribute that is used for taking user input from the user.

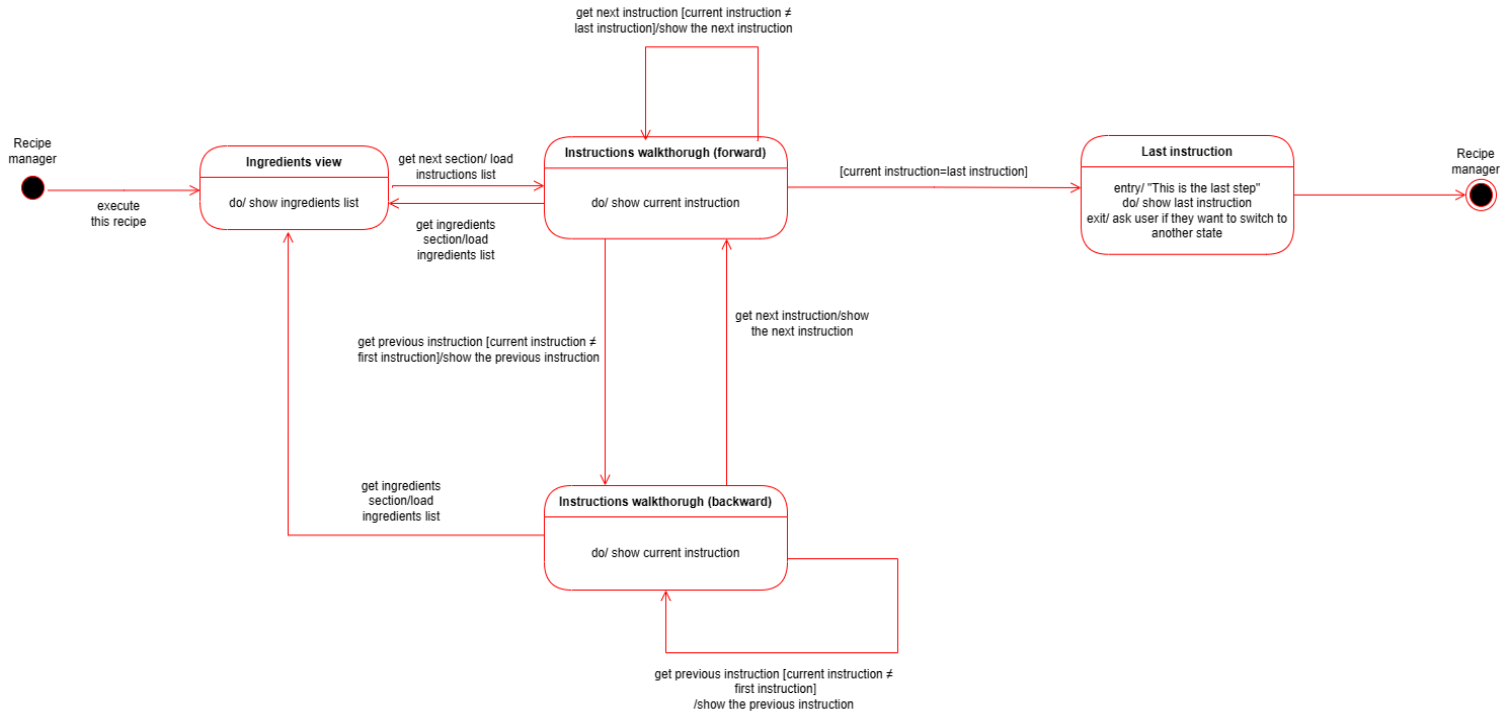


This chapter contains a revised version of the same UML object diagram you modelled in Assignment 2 (with all changes highlighted graphically), together with a textual description of all main improvements **and** the reasoning behind them.

Maximum number of pages for this section: 1

Revised state machine diagrams

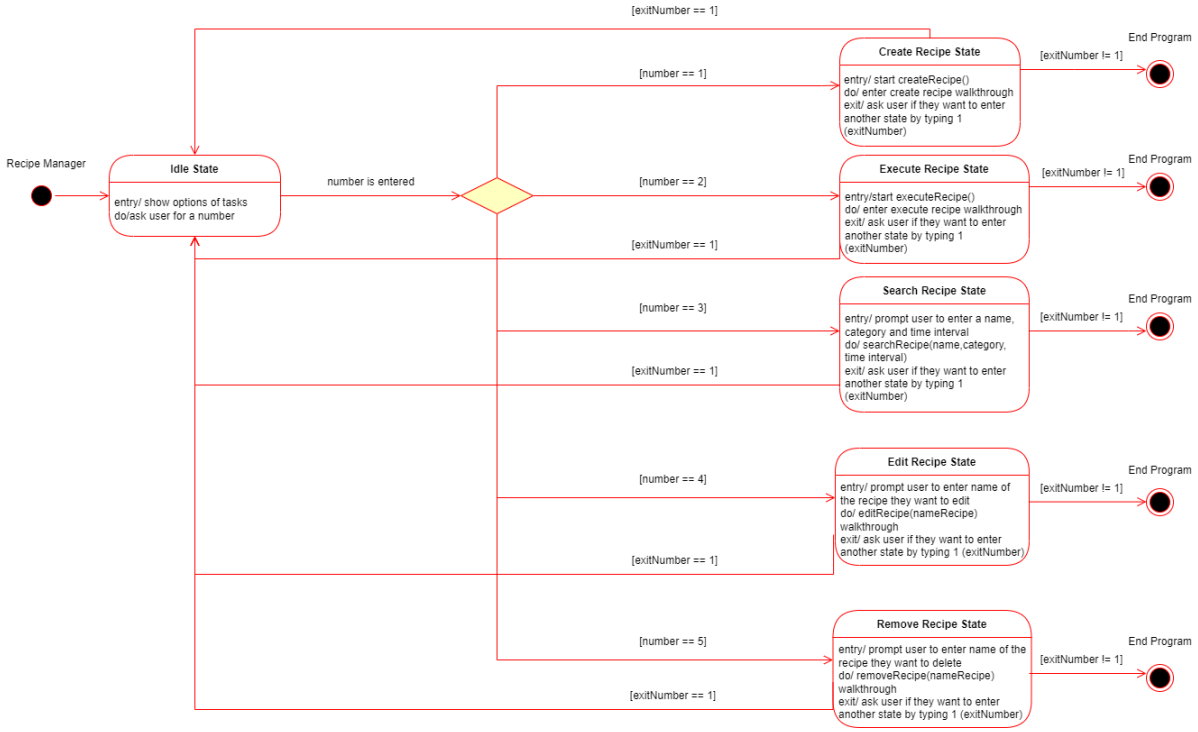
Author(s): Adrian & Taher



1. Recipe State Machine

We have changed the state machine diagram to describe the *Recipe* object and its states when it is being executed by the RecipeManager method executeRecipe(). Now there are 4 distinct states for the *Recipe* object: Ingredients view, Instructions walkthrough (forward), Instructions walkthrough (backwards), and Last instruction. The ingredients view is the first state and it can be accessed from two other walkthrough states. We have changed the naming of the states to be nouns so it is more obvious that they are describing states. The arrangement of the states has also been changed to show that the states are traversed from left to right until the terminal state.

2. Recipe Manager State Machine



Our previous diagrams weren't state diagrams because we didn't describe the states of an object, but of the state of methods. In order to fix that we built new state machines.

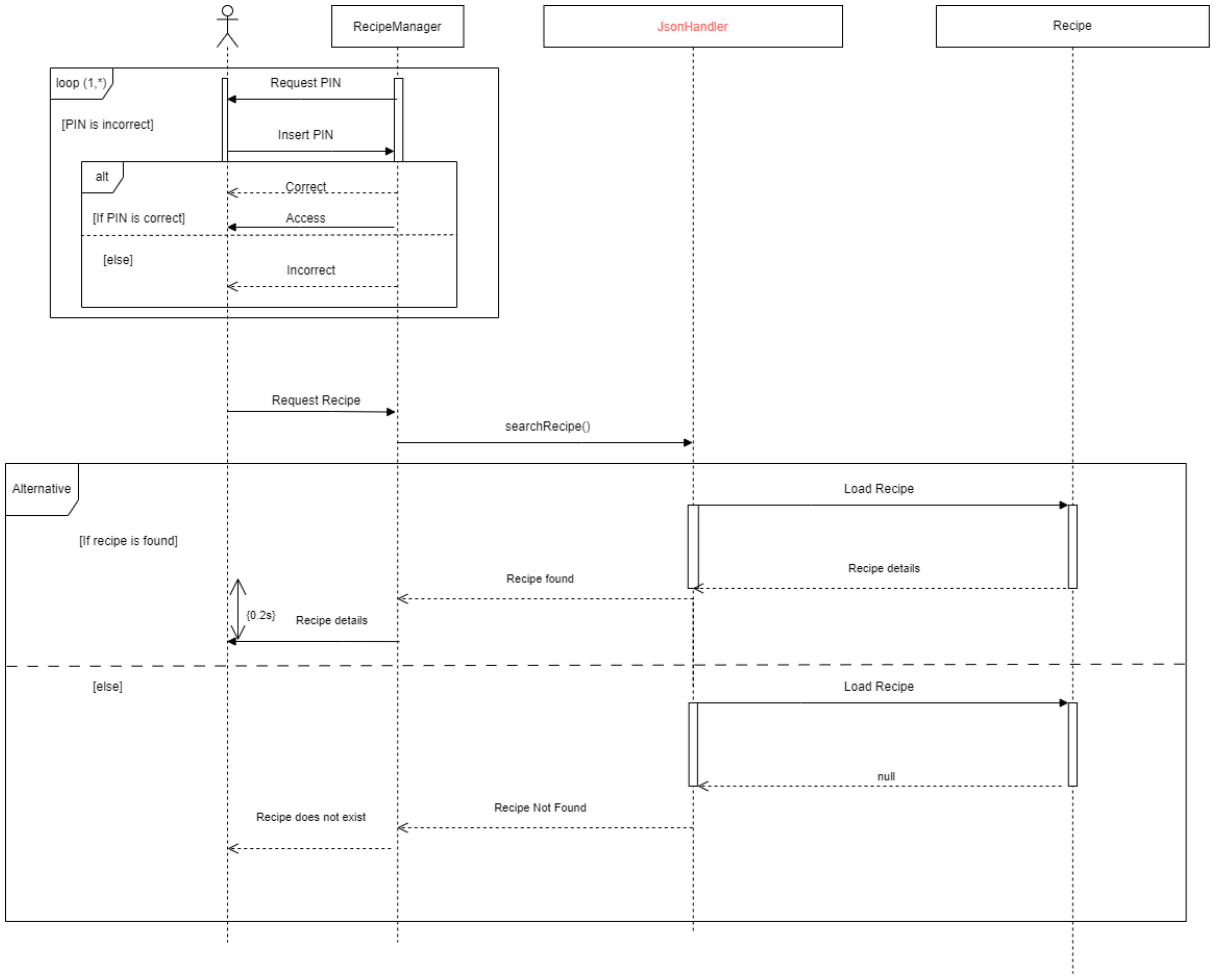
The *Recipe Manager* is the most important class of our code because it connects all of the other classes. Our *recipe manager* starts in an idle state where it will show the users the options he can choose from our program. The *recipe manager* has 5 options/states and each one is activated with typing its corresponding number and we have the event "number is entered" with a guard for each option.

Each option changes the *Recipe Manager* for the fact that each one has its own walkthrough that changes the whole state of the *recipe manager.* At the end of each state the user is asked if they want to go back to the idle state or end the program. If the number of exit is 1 the *Recipe Manager* will go back to idle state, if not the program reaches its end state.

Revised sequence diagrams

Author(s): Adrian & Taher

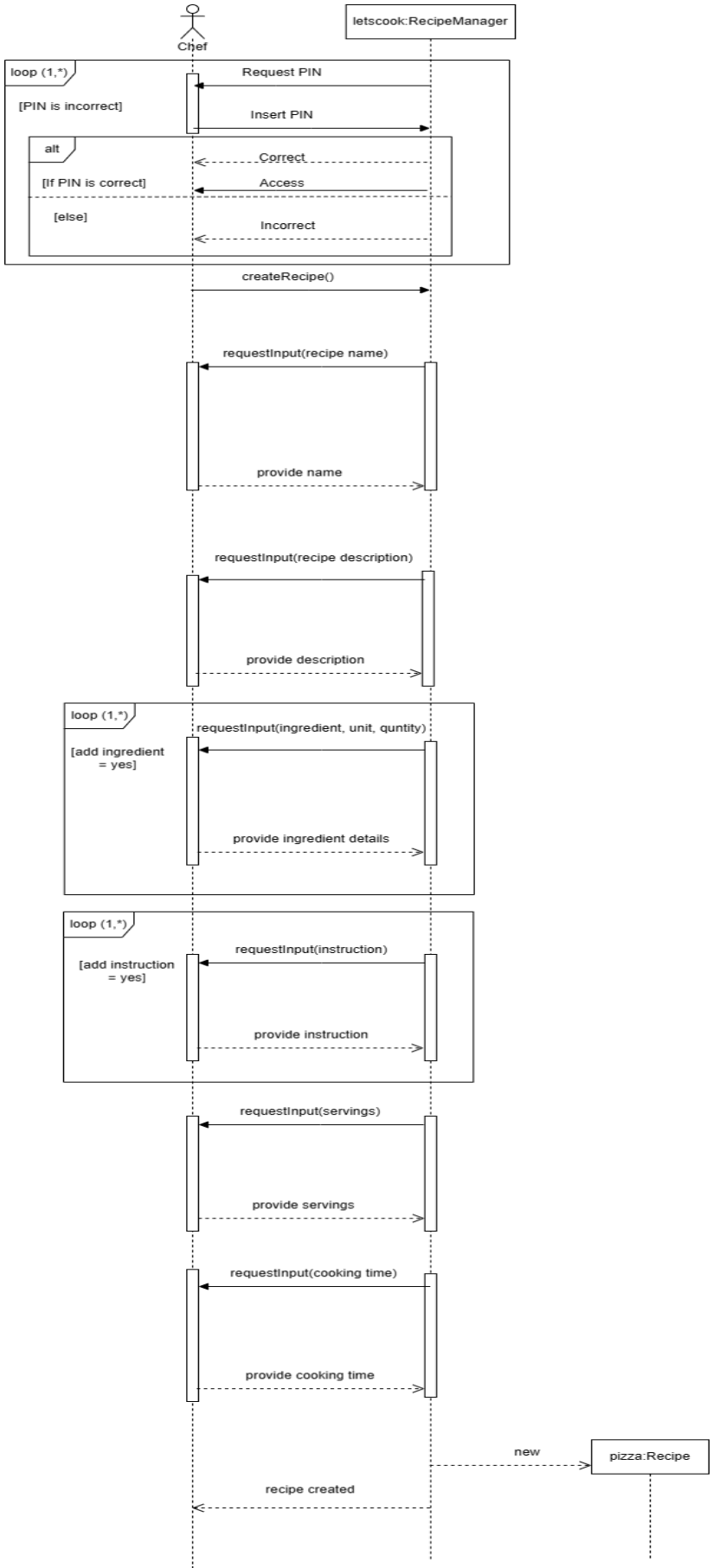
Diagram 1: Recipe Search



In our first diagram we had an object called storage, but that specific object didn't exist in our previous class diagram hence the sequence diagram was wrong.

In our code the JsonHandler class gets the data of recipes and transports them to the recipe manager. The only change we made is instead of using the storage object which doesn't exist we used the JsonHandler class which oversees the recipes json files.

Diagram 2: Recipe Creation



The sequence diagram for the recipe creation has been changed in a number of ways to accommodate for a more technical implementation. The *Recipe* object is removed because the user can only interact with the *RecipeManager* object to create a new *Recipe* object and the *letscook* object can do that without interacting with a *Recipe* object. This way the sequence of interactions between the chef and the *RecipeManager* is minimized. Second, a couple of additional loop fragments have been added for the ingredients and instructions. This is because the user can add multiple instructions and ingredients if they want to and they will be prompted for those along with their details individually. Finally, the *RecipeManager* object (*letscook*) and new *Recipe* object (*pizza*) were given names to make it obvious we are talking about instances.

Implementation

Author(s): Moegiez

In developing the recipe manager project, following the UML models created during the design phase was a wise choice as it ensured that the implementation remained consistent with the intended design. UML diagrams such as class diagrams, use case diagrams, state machine diagrams and sequence diagrams helped us visualize the system's behavior and structure, which is vital in software development.

When moving on from the design phase to the implementation of the code, the team chose to start by creating the attributes before implementing the methods. This approach is a common practice in software development, where developers define the data structures before implementing the functionality. This helps to ensure that the data structures align with the design and that the code is modular and easy to maintain.

The team also decided to use two design patterns, the Singleton design pattern, and the Iterator design pattern, in the project. The Singleton design pattern ensures that only one instance of a class is created and provides a global point of access to it. This pattern is useful when there is a need to limit the number of instances of a class in a system. The Iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This pattern is useful when there is a need to traverse through a collection of objects without exposing its internal structure. These came in handy for the RecipeManager and the executeRecipe respectively.

Overall, the team's choice to follow the UML models during the design phase, start by creating attributes before implementing methods, and use the Singleton and Iterator design patterns were all sound decisions that contributed to the success of the project. These practices ensure that the code is easy to maintain, scalable, and meets the requirements of the project.

During the development we faced certain challenges. A notable challenges were the handling of data using JSON, which we eventually solved by using the gson library. This allowed us to persist recipes as json files which save recipes created by a user so that the user can access them even after he/she has terminated the program. It also enabled us to store data about users such as their usernames, PIN and other relevant data

Another notable challenge we faced was working as a team in github. Words such as pull request, commit, push were unfamiliar for most of us. Studying git and figuring out how to use this enabled us to work as efficiently as possible.

- the location of the main Java class needed for executing your system in your source code;
  Assignment-3 branch
  src/main/java/softwaredesign/Main.java
- the location of the Jar file for directly executing your system;
  Assignment-3 branch
  out/artifacts/software_design_vu_2020_jar/software-design-vu-2020.jar
- the 30-seconds video showing the execution of your system (you are encouraged to put the video on YouTube and just link it here). https://youtu.be/0qIys2TpJXg

Time logs

<Copy-paste here a screenshot of your time logs - a template for the table is available on Canvas>

Team number	49					
Member	**Activity**			**Week number**	**Hours**	**Progress**
Taher	Learning java syntax and using intellij with github			5	10	Done
Moegiez	Learning java syntax and using intellij with github			5	5	Done
Adrian	Learning java syntax and using intellij with github			5	5	Done
Shantanu	Learning java syntax and using intellij with github			5	5	Done
Taher	Implementing classes from the class diagram			5	2	Done
Moegiez	Implementing classes from the class diagram			5	2	Done
Adrian	Implementing classes from the class diagram			5	2	Done
Taher	Coding executeRecipe()			5	6	Done
Moegiez	Coding executeRecipe()			5	6	Done
Adrian	Coding searchRecipe() and part of editRecipe()			5	7	Done
Shantanu	Coding editRecipe()			5	6	Done
Taher	Debugging the code for any errors			6	4	Done
Moegiez	Debugging the code for any errors			6	4	Done
Adrian	Adding time intervals to searchRecipe()			6	4	Done
Shantanu	Working on the JSON storage			6	5	Done
Taher	Debugging the code for any errors			6	4	Done
Moegiez	Debugging the code for any errors			6	4	Done
Adrian	Debugging the code for any errors			6	2	Done
Shantanu	Debugging the code for any errors			6	3	Done
Taher	Fixing errors userInterface			7	1	Done
Moegiez	Fixing errors userInterface			7	1	Done
Adrian	Fixing errors userInterface			7	1	Done
Shantanu	Fixing errors userInterface			7	2	Done
Taher	Working on sequence and state machine diagrams			7	3	Done
Moegiez	Working on sequence and state machine diagrams			7	3	Done
Adrian	Working on sequence and state machine diagrams			7	3	Done
Shantanu	Working on sequence and state machine diagrams			7	1	Done
			TOTAL	101		