1.2 Gigabyte Café

—

Group II

—

Dawid Zawiślak
Gabriel Mendoza
Scott Jette
Enrique Vaca
Zheng Quan

—

Version: 2.1

—

April 20^{th}, 2015

Contents

List of Figures

1 Introduction

1.1 Introductory Notes

References to `DIRECTORY`, `in`, `factory`, and `menu` will be seen and mentioned throughout the document. These always mean the same thing. `in` is simply an initialized Scanner that reads from `System.in`, `factory` is a ConsumableFactory which is tasked with the creation of Consumable files. `menu` is the only instance of Menu (any further calls to `getInstance()` will return a reference to the same Singleton Instance.)

Likewise, references to `addItem()` and `removeItem()` will be made, particularly inside of Menu. These are not real methods and they are also not shown in the class diagrams— these methods are intended to be placeholders. `addItem()` refers to the methods `addAppetizer()`, `addMainDish()`, `addDessert()`, `addSoda()`, and `addLongDrink()`. `removeItem()` refers to the family of remove methods of the same foods and drinks.

This report has been written in LaTeX. A copy of the source code may be provided upon request.

1.2 Scope and Goals of Project

The project is intended to allow a shop owner to maintain a menu for the benefit of its customers. This menu is capable of adding and removing items— that will be saved to a collection of text files for later viewing. This will benefit shop owners because of ease of use and ability to put onto multiple devices without a lot of maintenance. This will also benefit the customers because they will have access to an interactive menu that is easy and fun to use.

1.3 Hardware and Software Platforms for Development and Deployment

The use of this menu requires the shop owner to install the latest version of Java onto the device. Since this device is not resource intensive, it should not require a fast device. The interface will be through `System.in` and `System.out` so it should be able to run on GNU-Linux, Windows, and Mac OS X-Darwin.

It is assumed that the shop owner created a directory that will be used for the reading and writing of `*.summary` files. If this folder does not exist, the menu will be empty on every run.

2 Requirements Specifications

This systems functionality is to provide an interactive interface for the restaurants menu. The menu items are managed using the 'Consumable' interface, to provide the commonalities between different kinds of menu items that branch out from the 'Food' and 'Drink' superclasses.

There are three types of expected inputs that the system provides, depending on the user. If customers are using the system, then they will only be allowed access to the 'MenuViewer' instance. In contrast, if the shop owner is using the system, then said user can access through the 'MenuEditor' to add or remove elements from the menu. The third type of input is from a database file, that will be implemented using a collection of text files. These text files will be used to load and save previously created menu items. If any changes are made during the execution of the system, changes will be reflected back to the file so that they can be reloaded the next time the program is ran.

In more detail, when the program executable is ran, the system will open up database text files, one for every type of menu item— appetizers, main dishes, desserts, sodas, and long drinks— and read from them. The program will read from the file and create instances of different menu items through use of the 'ConsumableFactory' and passed as arguments to the `addItem(...)` method of the Menu. All of these will be stored internally by the program using an `ArrayList<Consumable>`, one for every menu subsection. Similarly, menu items can also be added manually, except that the user will be prompted for input during each step of the creation of each menu items.

3 Software Architecture

3.1 Main Components

The main components of the project is the 'Restaurant', which will get an instance of 'Menu' that is generated through the use of the Singleton Design Pattern. 'MenuItems' is a directory that stores files that will be written to and loaded into the program at run time. From there, the customer will be able to view the menu, as shown by the instance of 'MenuViewer' in diagram 1. The instance of 'MenuEditor,' as shown in diagram 1, is simply the administrative tools that the shop owner will have in order to add items, remove items, and change prices.

3.2 Architecture Diagram

The architecture diagram can be found in the back of this document. It is diagram 1.

4 Detailed Design

The heart of the restaurant menu is found in the class 'Menu' that enables the store owner to add/remove files to this list and also allows the customer to look through the goods. This menu employs design patterns to make it easier to maintain. The interface will contain the ability to browse through the menu and add items to a list of favorites.

4.1 Patterns Used

The patterns used for this project consist of the Singleton Pattern, Factory Pattern, and Iterator Pattern. The Singleton Pattern enforces that each restaurant will only have one menu. The Factory Pattern allows the store owner to create Consumable class files along with the batch processing of `*.consumable` files that contain items for the menu. The Iterator Pattern is used so the customer of the café is able to look at each item in a sub-menu one-by-one.

4.2 Classes

For reference, class diagrams may be found in the Figures section of this document. Classes 'Restaurant', 'Menu', 'ConsumableFactory', 'MenuSubsection', and 'MenuIterator' may be found on diagram 4. Diagram 5 contains information pertaining to the application of 'Consumables' and all classes that implement it.

4.2.1 Restaurant

The 'Restaurant' class is simply where the menu begins execution. Nothing further is done here. This is where the interaction between both, the customer and employee will take place. This interaction is largely text based.

A constant string titled `DIRECTORY` is inside of Restaurant. `DIRECTORY` is set to the name of the folder that will house the `*.consumable` files. If this directory does not exist, it will simple be created for you. Please note that it **MUST** end with a '/' at the end. It is currently set to 'Consumables/'.

Aside from `DIRECTORY`, `in`, `factory`, `menu`, and `favorites` can be found. `DIRECTORY`, `in`, `factory`, and `menu` are explained in the Introductory Notes. `favorites` is an ArrayList of Consumables that hold any items the user marks as a favorite.

Lastly, in order to divide the interface into several parts, the methods `admin_tools()`, `favorites()`, and `browse_menu()` are the interfaces of these parts of the Menu. `add_files()` works in conjunction with the ConsumableFactory pattern in order to create all the Consumables stored inside of the directory– `DIRECTORY` prior to any interaction with the user.

4.2.2 Menu

'Menu' is brought forward by calling the `getInstance()` method. As shown in diagram 2, it begins life by initializing itself, and then returns that single instance to Restaurant. 'Menu' will hold five 'MenuSubsection' instances that contain the sub-menus for appetizers, main dishes, desserts, sodas, and long drinks.

The method `addItem()` (...do note that no `addItem()` method exists, but this is rather a place holder for `addAppetizer()`, `addDessert()`, `addMainDish()`, and so forth) requires a cast to the appropriate class object (Appetizer, MainDish, Soda, etc...) Upon doing so, it will add the Consumable item to the appropriate 'MenuSubsection.' `removeItem()`, as the name implies, will remove the particular 'Consumable' from its 'MenuSubsection.'

`getAppetizers()`, `getMainDishes()`, `getDesserts()`, `getSodas()`, and `getLongDrinks()` will return the 'MenuSubsection' that holds that particular collection of food items, and `resetMenu()` will reset all of these 'MenuSubsections' so that they will contain no goods.

4.2.3 ConsumableFactory

The ConsumableFactory class is one of the design patterns inside of the Gigabyte Café. ConsumableFactory will be what designs a 'Consumable' object after being given certain input. It will be responsible for processing any `**.consumable` files while also reading in any requests the store owner makes when adding items to the Menu. 'ConsumableFactory' shall consist of two methods: `getConsumable(String)` and `createConsumableFromFile(File)`. `getConsumable(String)` will generate a 'Consumable' based on whatever string was provided to it. It will ask the user for input, and in this case, the user shall be the shop owner. `createConsumableFromFile(File)` will generate a 'Consumable' based on the input of a file named `*.consumable`. A code snippet that shows roughly how the factory may read files may be found in the 'Code Snippets' section.

4.2.4 MenuSubsection and MenuIterator

'MenuSubsection' and 'MenuIterator' implement the interfaces 'Container' and 'Iterator' respectively. It is important to note that 'MenuSubsection' contains the class 'MenuIterator.' Instances of this 'MenuIterator' are limited to existing only through instances of 'MenuSubsection.'

'MenuSubsection' utilizes a 'ArrayList¡T¿' to store every 'Consumable.' Accessing this 'ArrayList' is done through the 'MenuIterator' which can be returned to the caller through the `getIterator()` method. The methods `add(Consumable)`, `remove(Consumable)`, and `contains(Consumable)` are extensions that will call the appropriate methods inside of 'ArrayList.' These methods are intended to restrict access to this list.

'MenuIterator' holds the size of the 'ArrayList', while also keeping track the current index. `next()` returns the next 'Consumable' to the caller and is used to scroll through the 'Menu.' `hasNext()` is used to determine if the end of the underlying 'ArrayList¡T¿' has been reached by returning a boolean. It will return a `boolean` value of `true` or `false`. Like `hasNext()`, `hasPrev()` checks if there is an item before the user's current position.

`current()`, `previous()`, `next()` return the items that are in the current index, next index, and previous index respectively. `previous()` and `next()` will manipulate the current position index.

`deleteCurrent()` deletes the Object contained at the current index. This will be used by owners and other employees with sufficient privileges.

4.2.5 Consumable

'Consumable' will be the interface that will be used to refer to all foods and beverages on the Gigabyte Café menu. `getPrice()`, `getCalories()`, and `getName()` are abstract methods that will return whatever the price, calories, and names of objects are. `setPrice(double)` will allow the store owner to change the price of an object on the menu. Please note that there are no setters for calories and name simply because a goods ingredients and name shouldn't be changed after its creation. Price changes are feasible, but alteration of calories or names should warrant the creation of a new item.

`printContents()` is implemented to facilitate the displaying of the 'Consumable' details. Superclasses that contain unique information (such as Ingredients or a Description) will be able to print their contents along with base information.

4.2.6 Food and Drink

'Food' and 'Drink' are abstract classes that implement 'Consumable', dividing 'Consumable' into its two types. Base methods such as `getPrice()`, `getName()`, and `getCalories()` are defined here.

4.2.7 MainDish, Appetizer, Dessert, Soda, LongDrink

'MainDish', 'Appetizer', and 'Dessert' will hold their unique bits of information (such as side dish and ingredients for 'MainDish', ingredients for 'Appetizer', and description for 'Dessert.') Each of these will have their respective getters and, if applicable, setters. `printContents()` here will print any unique values held by these classes and then call the `printContents()` of the superclass which will print the values of all common fields (... such as price, calories, and name.)

'MainDish,' 'Appetizer,' 'Dessert' will be subclasses of 'Food'— 'Soda' and 'LongDrink' will be subclasses of 'Drink.' These two classes should be quite similar in practice, but this is merely a way of having a way of separating the foods from the drinks.

5 Implementation Notes

5.1 Code Snippets

createConsumable The `createConsumable` method will ask the user for information and, from there, write a `.consumable` file in the directory that the code ran in. This needs slight modification but is essentially the 'MenuEditor' portion of diagram 1.

```
/**
 * Create a consumable file from the user's input.
 *
 * [ TODO ] This will write to where ever this
 *          was executed. This should be changed
 *
 */
public static void writeConsumable() {
    Scanner in    = new Scanner( System.in );
    String input = null;
    String item  = null;
    char   c_in  = '?';

    System.out.print( "[[F]ood or [D]rink]> " );
    input = in.nextLine();

    c_in = Character.toLowerCase( input.charAt(0) );
    if ( c_in == 'f' ) {
        System.out.print("\t[[A]ppetizer, [M]ain Dish, [D]essert]> ");
        input = in.nextLine();
        c_in = Character.toLowerCase( input.charAt(0) );
        if ( c_in != 'a' && c_in != 'm' && c_in != 'd' ) {
            System.out.println("\t\tInvalid Input");
            return;
        }
    }
    else if ( Character.toLowerCase( input.charAt(0) ) == 'd' ) {
        System.out.print("\t[[S]oda, [L]ong Drink> ");
        input = in.nextLine();
        c_in = Character.toLowerCase( input.charAt(0) );
        if ( c_in != 's' && c_in != 'l' ) {
            System.out.println("\t\tInvalid Input");
            return;
        }
    }
    else {
        System.out.println("\tInvalid Input");
        return;
    }

     String name, s_price, s_kcal;
```

```
    double price;
    int    kcal;

    System.out.println("\n--\n");

    System.out.print( "[Name]> " );
    name = in.nextLine();
    System.out.print( "[Price]> " );
    s_price = in.nextLine();
    price = Double.parseDouble( s_price );

    System.out.print( "[Calories]> " );
    s_kcal = in.nextLine();
    kcal = Integer.parseInt( s_kcal );

    PrintWriter filerw = null;
    try {
        filerw = new PrintWriter(name + ".consumable", "UTF-8");
    } catch (FileNotFoundException | UnsupportedEncodingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    filerw.println(c_in);
    filerw.println(name);
    filerw.println("" + price);
    filerw.println("" + kcal );

    switch ( c_in ) {
        case( 's' ):
            break;
        case( 'l' ):
            System.out.print("[Ingredients]> ");
            String l_ingredients = in.nextLine();
            filerw.println(l_ingredients);
            break;
        case( 'a' ):
            System.out.print("[Ingredients]> ");
            String a_ingredients = in.nextLine();
            filerw.println(a_ingredients);
            break;
            (... parts of code truncated ...)


        }


    filerw.close();
}
```

listAllFilesInFolder The `listAllFilesInFolder` methods show how all files in a directory can be read. Any files that end in `.summary` will have their path printed in this example. In application, this would be replaced with the generation of a 'Consumable.'

```
/**
 * Lists the given files inside the given directory and all subdirectories.
 *
 * @param _folder path to a folder
 */
public static void listAllFilesInFolder( String _folder ) {
    File f = new File( _folder );
    listAllFilesInFolder( f );
}

/**
 * Lists the given files inside the given directory and all subdirectories.
 *
 * @param _folder a File class that is initialized to a folder
 */
public static void listAllFilesInFolder( File _folder ) {
    /* Return if whatever _folder is is not a folder. */
    if ( _folder == null || !_folder.isDirectory() ) {
        return;
    }

    /* List all the files inside the folder. If a file is a folder,
     * recursively list its contents. If it is a  file,  list  the
     * absolute path to the file.
     */
    for ( File f: _folder.listFiles() ) {
        if ( f.isDirectory() ) {
            listAllFilesInFolder( f );
        }
        else if ( f.isFile() ) {
            String path      = f.getAbsolutePath();
            String extension = null;

            int last = f.getName().lastIndexOf('.');

            if ( last > 0 ) {
                /* If last == 0, it is a hidden folder on a LINUX/Darwin
                 * environment.
                 */
                extension = f.getName().substring(last+1);
            }
```

```
            /* If a file is a consumable, say that it is so. This is where
             * the factory pattern will design the Consumable class
             */
            if ( extension != null && extension.equals("consumable") ) {
                System.out.println( path + " is a consumable file" );
            }

        }
    }
}
```

5.2 Unresolved Design Issues

At this time, there are no unresolved design issues worth addressing. Although it is certain that problems will come up, they have not yet arisen.

5.3 Group Distribution

Design of this document was a group effort, with the work divided in this manner:

- Dawid Zawiślak— Worked on the detailed design, put all the parts into one document, and contributed to diagram 2.
- Enrique Vaca— Worked on the software architecture section, contributed to diagram 2, and proofreading.
- Gabriel Mendoza— Worked on requirements specifications, contributed to diagram 2. Thoroughly proofread the document to ensure correctness.
- Scott Jette— Completed diagram 2 and all of diagram 3.
- Zheng Quan— Worked on the Implementation Notes section.
- The UML Class Diagrams (Diagrams 4 and 5) were a group effort between Dawid Zawislak, Enrique Vaca, Gabriel Mendoza, and Scott Jette by collaborating over Google Hangouts.
- Coding related contributions are found inside of the Java source code. Classes were mostly done by one person, and the comments should show who created the class with the `@author` flag. If work was split, `@author` will also be inside of the comments for the methods.

We already finished diagram part of this assignment, and the rest is just coding. For the coding part, as we know, there are five main consumables that must be addressed— Appetizers, Main Dishes, Desserts, Soda Selections and Long Drinks. The UML software already generated the skeleton code for us, so our job is to fill in the blanks.

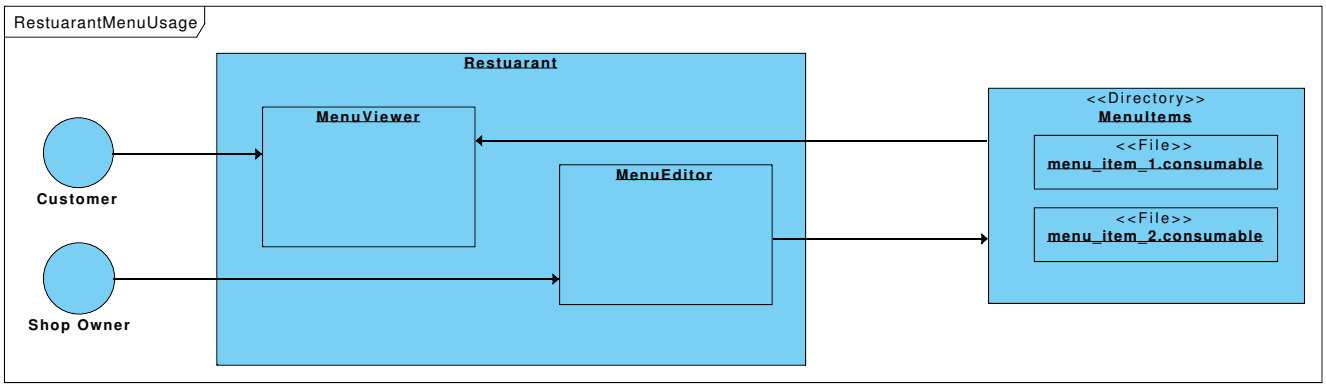
6 Diagrams

- **Class Diagrams** are diagrams 4 and 5.
- **Sequence Diagrams** are diagrams 2 and 3.
- **The Architecture Diagram** is diagram 1.

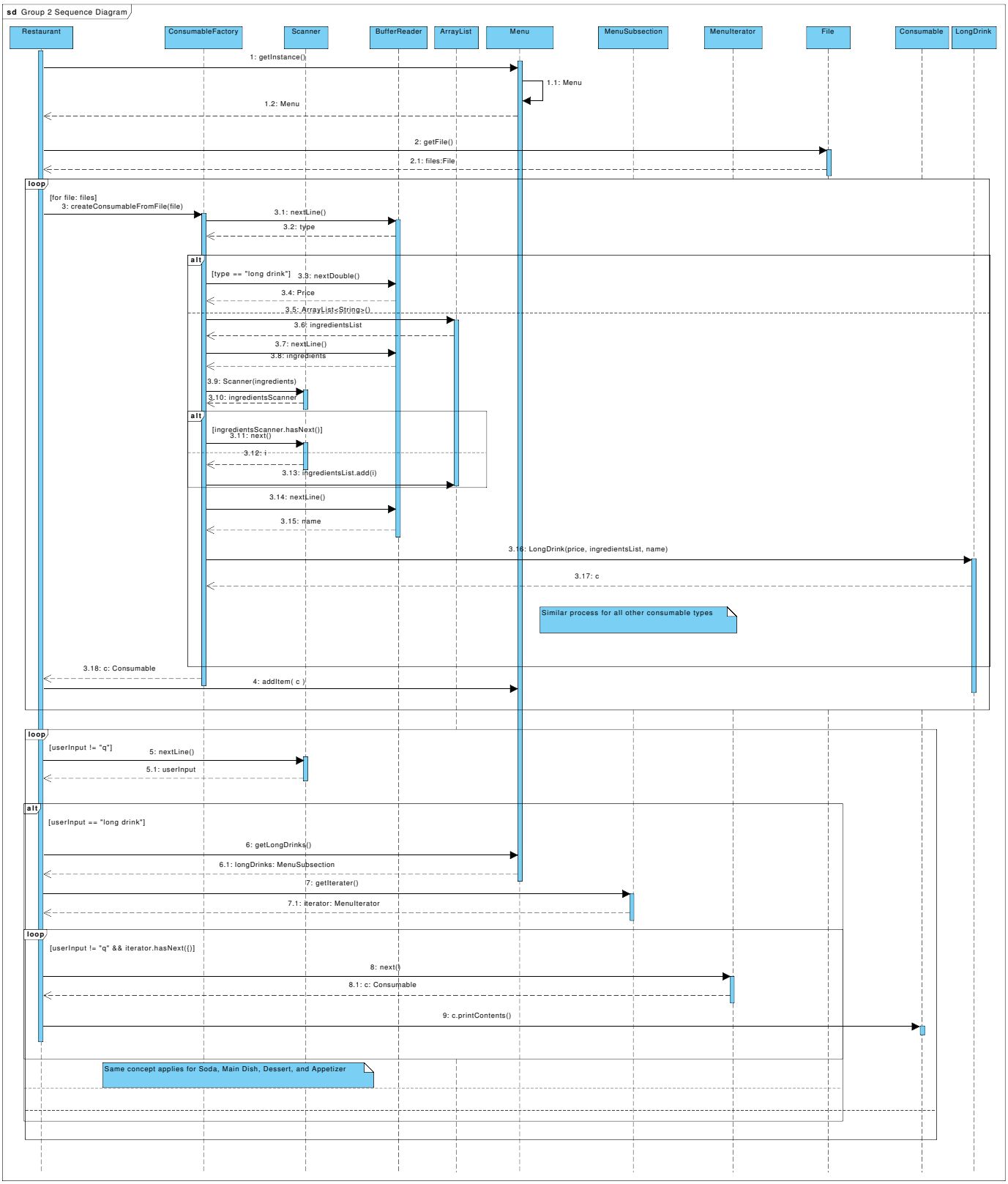
—

Sequence Diagram 2 shows how a customer reads the menu. Upon initializing, the menu will read all files and import their contents into the program. Afterwords, the user is asked what they would like to view. If this response is correct, they are given the information about the first item in the given subsection, and then will be given the next item as long as there are more available and the user does not input q (for **q**uit.)

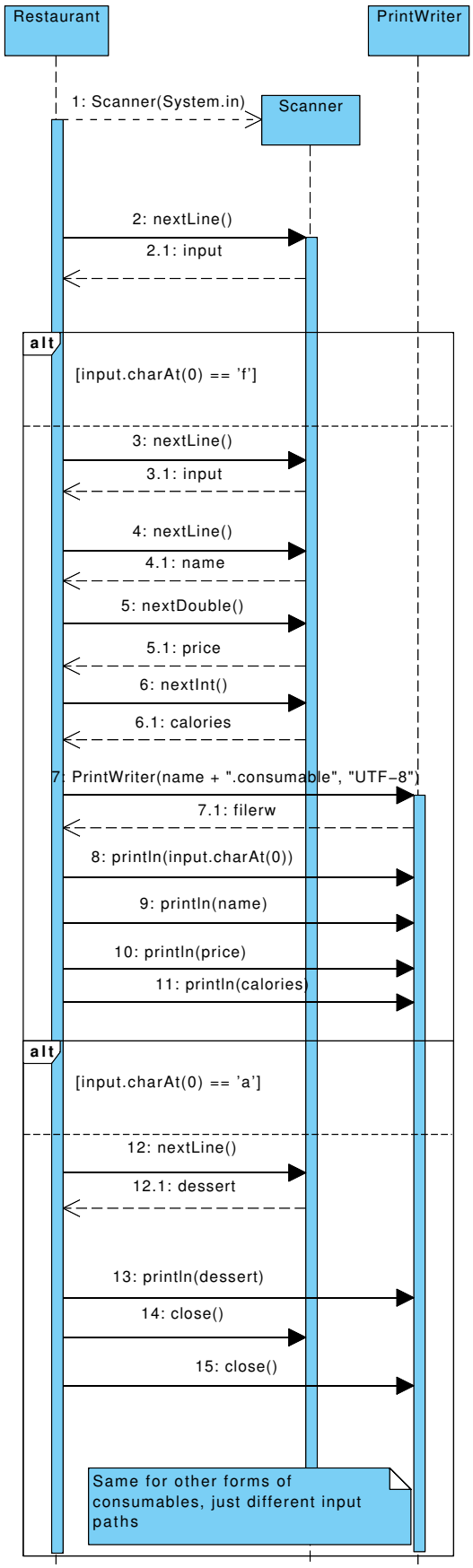
Sequence Diagram 3 shows how the shop owner would add a item to the menu. After being prompted information for a given Food or Drink, the program will save a file titled `<Food Name>.summary`, which will be loaded in the future in order for the menu to be populated automatically. This sequence diagram closely resembles the code snipped `createConsumable`.



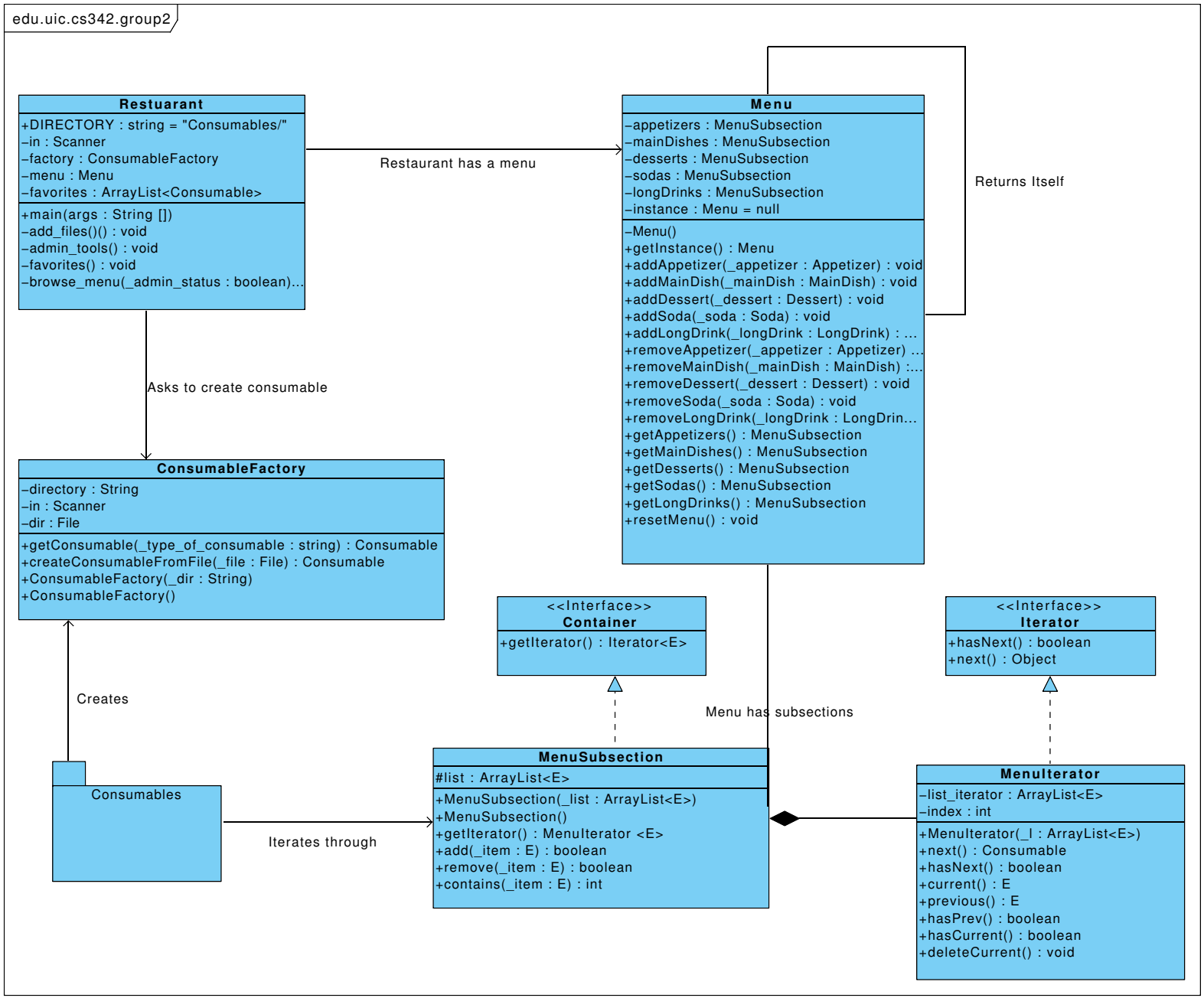
UML Componant Diagram: Component diagram for Gigabyte Café



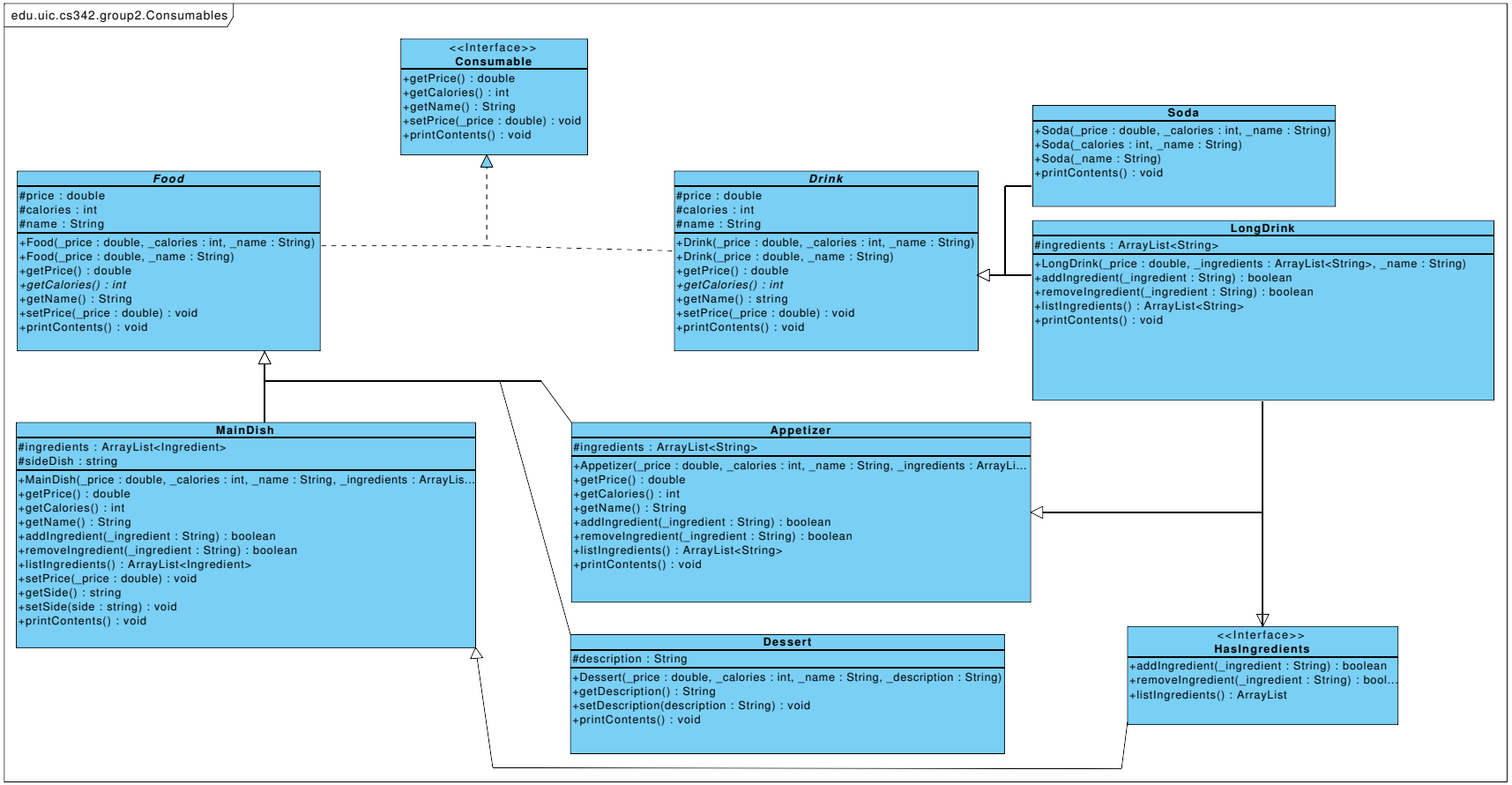
UML Sequence Diagram: Customer reading from the menu



UML Sequence Diagram: Owner adding to the menu



UML Class Diagram: Contains the Restaurant and Menu class



UML Class Diagram: Contains all food and drink items