**Stage B : Design**

Hong Joon Choi

**Stage B : Design**

B1. Data structures

The file that holds reservation will be a sequential file, with all the sorted reservations saved. Each reservation will have five fields: customer name, reserved time, number of customers, table number and additional request.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Variable type** | **Sample data** | **justification** |
| CustomerName | String | HongJoon | Customer name is a string because name is made of letters |
| ReservedTime | Calendar | 1352300886780 | Reserved time is represented in milliseconds, Calendar data type has built in function that does conversion for us. Millisecond form will be changed into specific dates and times whenever user has to see this data. |
| numberOfCustomers | byte | 8 | Number of customers will be in integer form, and it won’t require large variable size since it is not possible for groups of 100 people to reserve a restaurant. |
| TableNumber | Int[] | 74, 75 | Each tables is assigned with its ID number, and it has four seats on it.  Variable type is in array because more than two tables may be assigned for group of customers who have more than 5 people. |
| request | String | I want to seat outdoor | Because request from customer is in sentence form, String variable is used. |

An array of reservations at Restaurant class will read all the bookings for specific day and it will list on screen. When the user wants to get specific reservation for restaurant, program will search for names. When a name is found, program will allow user to edit or delete the booking when it is processed.

B2. Algorithms

|  |  |  |
| --- | --- | --- |
| **Name** | Search | |
| **Description** | Returns index of array that holds the requested String | |
| **Preconditions** | There is an array that holds data. | |
| **Parameters** | **Local Variable**  **Name Type Vaulue** | **Return values** |
| String ss  String[] array |  | Int |
| **Code** | for int i=0, i<array.length  if ss = array[i]  return i  end if  next i  return -1 | |
| **PostConditions** | Index of an array will be found and will allow user to do operations with it, if nothing is found, -1 is returned | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | | smartSearch | |
| **Description** | | Returns multiple indexes of array that has part of the String. For example, “james” and “amy” will be found if user inputs “a” | |
| **Preconditions** | | There is an array that holds data of String | |
| **Parameters** | **Local Variable**  **Name Type Vaulue** | | **Return values** |
| String ss  String[] array  Int arraySize | list int[] -1  numberFound int 0  disassemble char[] ‘’  reassembled String “” | | Int[] |
| **Code** | for int i=0, i<arraySize  disAssemble = staff[i] to array of characters  for k=0, k<staff[i].length  String assembled =""  for int j=k, j<staff[i].length  assembled+=disAssemble[j]  next j  if assembled startsWith ss  list[numberFound]=i  numberFound++  break  end if  next k  next i  return list | | |
| **PostConditions** | Array of integers which indicate index of the array will be returned. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | | Sort | |
| **Description** | | Sorts array in ascending numerical order. | |
| **Preconditions** | | Array that holds time values (in milliseconds), is out of order | |
| **Parameters** | **Local Variable**  **Name Type Vaulue** | | **Return values** |
| int[] time | Swap int 0 | | int[] |
| **Code** | for k=0,k<length  for int i=0,i<length  if time[k]<time[i]  swap = time[k]  time[k] = time[i]  time[i] = swap  end if  next i  next k  return data | | |
| **PostConditions** | Array will be sorted in ascending order. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | Delete | | |
| **Description** | Deletes an index from array, and decrease size of the array by 1 | | |
| **Preconditions** |  | | |
| **Parameters** | | **Local Variable**  **Name Type Value** | **Return values** |
| int index  Reservation[] original  int arraySize | | overwrited Reservation[] null | Reservation[] |
| **Code** | | for int i=0,i<index  overwrited[i] = original[i]  next i  for int j=index, j<arraySize-1  overwrited[j] = original[j+1]  next j  return staffOverwrite | |
| **PostConditions** | | Data in array at particular index will be deleted and size of the array will be decreased by 1. | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | allignToMiddle | | |
| **Description** | Aligns String to the middle of given amount of spaces | | |
| **Preconditions** |  | | |
| **Parameters** | | **Local Variable**  **Name Type Value** | **Return values** |
| String message  Int space | | ss String “” | String |
| **Code** | | int blank=space-a.length;  if blank%2!=0  ss+=" "  end if  for int i=0,i<(blank/2)  ss+=" "  next i  ss+= message  for int i=0;i<(blank/2)  ss+=" "  next i  return ss | |
| **PostConditions** | | Message is aligned at the center of given space. | |

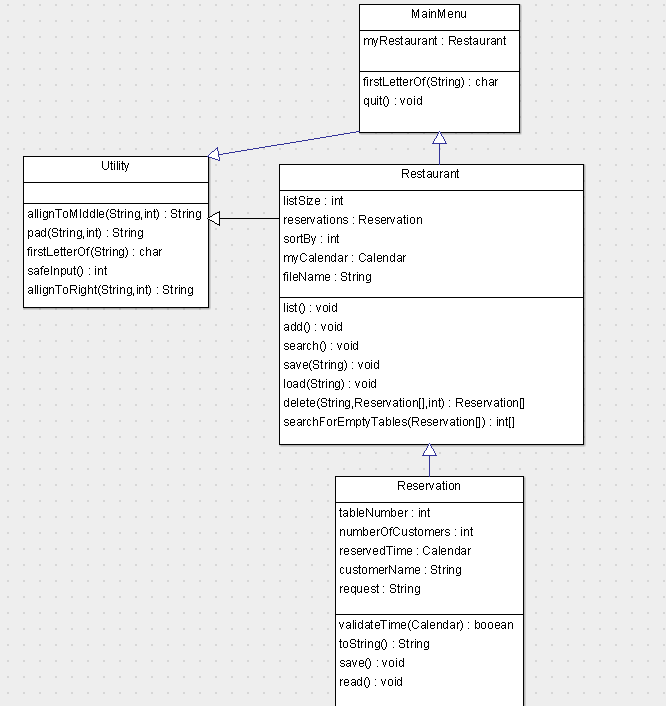
|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | Pad | | |
| **Description** | Allocates String and leaves space until the given points | | |
| **Preconditions** |  | | |
| **Parameters** | | **Local Variable**  **Name Type Value** | **Return values** |
| String message  Int space | | ss String “” | String |
| **Code** | | do  ss+=" ";  while ss.length<space  return ss | |
| **PostConditions** | |  | |

|  |  |  |
| --- | --- | --- |
| **Name** | validateIfTableIsEmpty | |
| **Description** | Validates if chosen table or tables are empty at specific time. The method returns true if the table or tables are available to be used at specific time. Customer who reserved a table is given 2 hous to finish their meal | |
| **Preconditions** |  | |
| **Parameters** | **Local Variable**  **Name Type Vaulue** | **Return values** |
| Long time  Reservation[] app  Int[] tables |  | Boolean |
| **Code** | for i=0, i<app.length  if app[i].reservationTime-time<3600000  for j=0, j<tables.length  for k=0, k<app[i].tableNumber.length  if tables[j]=app[i].tableNumber[k]  return false  end if  next k  next j  end if  next i  return true | |
| **PostConditions** | Program will indicate user if chosen tables are available at specific time | |

B3. Modular organization

**B3.1 class diagrams and explanations**

The program will have four classes, MainMenu, Restaurant, Utility and Reservation.



**Class 1 :MainMenu**

The MainMenu will be responsible for creating the restaurant class. It will also have to save the file, having an array to keep reservations saved. Main menu does not have any function except for quit().

**Class 2 : Utility**

This class is a collection of utility functions. Functions include aligning methods, input method that handles error and method that returns first character of the String. Class MainMenu and Resstaurant extends this class.

**Class 3 : Restaurant**

Restaurant class is collection class of reservations. This class is responsible for reading all reservations made and saved to the file. This class has two fields: reservations and size of reservation list. It has five functions that manages reservations: list, add, search, delete and function that validates if chosen table is empty at specific time. Other functions are save() and load().

**Class 4 : Reservation**

Reservation class is base class of the program and it holds data of one reservation. It has five fields to it, which is described at part B1. It has function of validating the range of time.

**B3.2 linking to solutions to problems.**

**Legibility :** List() function will print out list of reservations in clear and organized way. Utility functions at class Utility, such as AllignToMiddle() and pad() will make the list much clearer.

**Sort :** sorting algorithm will solve the problem of putting list out of order whenever modification is done. The program will automatically sort array whenever any action is done by user.

**Search :** Even if customer doesn’t spell out his/her name clearly, or if spelling of customer’s name is unclear, smartSort() method will allow user to find for names just by entering part of their names.