**CSE 1325 – Fall 2014**

**Object-Oriented and Event-driven Programming in Java**

**Lab 4**

Design Document Due Date: **Thursday, Nov. 13, 2014 at 11:59pm**

**(one minute before midnight 12am)**

Lab Due Date: **Thursday, Nov. 20, 2014 at 11:59pm**

Grade value: 10% out of 100% for all grades

**Objective:**

The theme of this project is to design classes and methods for creating a computer simulation of the scenario described below using object-oriented design (OOD) principles. This project will build on the basis of your previous projects in the semester. Please make sure you adhere to OO principles throughout the design and implementation. The projects are designed to give you an opportunity to exercise the principles taught in the course. Remember, Java is only a vehicle for supporting OOD. Make sure you fully leverage the features available in Java for your benefit as long as we’ve touched on them in class. Ask before jumping into totally new concepts for your lab.

In this project, you will be designing the data abstractions and required operations on those abstractions for the given scenario.

Be sure to check the DEDUCTIONS section at the end of this assignment to avoid penalties

You are also required to design your program in advance before you begin writing code. You will document your design and turn in the design document at least a week before the lab assignment is due. The goal of the design document is to assist you in developing the actual program.

**Description:**

You must design classes, data fields (attributes) and methods for the problem description below. Make sure you understand and differentiate between public and private ( and optionally protected) fields and methods. Use static fields/methods where appropriate. Make sure you have at least one constructor (more as needed) with arguments. Each class should have appropriate get (accessor) and set (mutator) methods. Each class should have a toString method (and other formatting methods as needed).

Use a “test” class to run the simulation scenario (also called the application) (to test/call each method defined in classes). The test class is the class with the main routine in it. For Lab #4, the test class will still offer the menu of simulation actions but it must also accept a file of simulation actions to execute. The test class should ask the user if they want a menu which includes all the possible simulation actions listed or if they have a file of the actions they are inputting. Based on the user’s request, the test class should accept user input and should execute the menu options to provide understandable output. If the user requests the menu, then the action options should be selected from a GUI as specified below. If the user is entering a file of instructions, the user will be able to specify the file name for the instructions file. Check for input errors. Test your source code—be sure every method works correctly.

The test class main routine should have scenario menu choices for all of the tasks listed below. This means that each element (choice) below must be executable (in a complete implementation). Each choice should be run with multiple sets of input data. Each set of input data for one menu choice is considered a test case for your classes. See further requirements for a test plan document which includes the test data that you have used.

**Scenario:**

Your program will be a simulation of a restaurant where the patrons (the diners) arrive, are seated at a table by a maître d’ (the restaurant host or hosts), are waited on by a server (waiter or waitress), and then have continuing interactions with the server and with the busser (the person that cleans tables, replaces dropped silverware, etc.) until the party of patrons receives a bill, pays the bill, and departs the restaurant.

The initial phase the “set up” phase, of your simulation system must set up the restaurant. The user of your simulation must define the number of tables in the restaurant, the number of patrons at a given table including a maximum number for that table, the number and names (and other info as needed) of servers that are working, the tables that “belong” to a certain server, the number and names of bussers working, the tables that “belong” to certain bussers, the number and name(s) of maître d’ that are working. This data will be entered as initial data for your simulation. This will be input for creating your restaurant object through the constructor method. Information about the constructors is given with the class info below.

The classes that are created for the restaurant will have additional data fields and methods as defined later in this assignment. These include things like whether a table is set ready for new patrons, what activity is occurring at a given table, etc. Additional classes may also be needed to implement the simulation such as a cash register, a menu, or a patron.

For Lab #4, the data to read in for the restaurant set up phase MUST be read from files. The files for restaurant set up will be described and one small sample will be given. Details of the files are given below.

Once the restaurant is set up, the simulation will switch to an operation mode. The operations mode consists of the Simulation Actions and the Simulation Display Actions. The operations mode may still be controlled through the use of a menu to run the simulation but for Lab #4, the operations mode may also be controlled by reading a set of simulation steps from a file and executing those steps. This process will be described in more detail later in this document. The user of your simulation should now be able to select actions to occur within the restaurant simulation. Some of the actions may be simple and some may have multiple steps. The user of your simulation will be controlling the simulation by selecting actions such as having patrons enter, get seated at a table, displaying various data, give menu selections to servers, receive menu items to eat, make other requests, get a bill from the server, and pay a bill. The other elements of the simulation must respond appropriately to each of these actions to keep the simulation running properly.

Your simulation will be “driven” or tested by inputting a set of actions to perform in a certain order. Your system should be able to determine if actions can be applied to the current restaurant situation or if the action cannot be performed. If the action is valid, then it should be performed and should change other values in the system to indicate that the action has occurred. In Lab #4 we will also simulate a “clock” that would allow the simulation to model actions happening in time and we will use time as another aspect of the simulation. This time will be recorded using the PDate as before but PDate will now use GregorianCalendar to get real times. Additionally for this lab, we will also use correct ordering of actions to insure that patrons don’t get a bill before they have ordered, for example.

The descriptions below will contain information about valid orderings of actions that can occur based on the status of a certain table of patrons, the status of the restaurant based on the number of patrons at tables and patrons waiting, and so on. In Lab #4 there are still some simplifications made for this simulation which will eliminate certain real world problems. As an example, the simulation will assume that all menu items that are ordered by a table will all arrive together and correctly at that table. However, you also have new requirements that increase the complexity of the system.

The following information is given in sections as follows. You must read all the data in all the sections to determine all the requirements because there are interactions between objects in the system. The first section will describe the classes that are required for the assignment. You may also create additional classes as needed. Within the classes, the minimum required set of data members and behaviors will be described. Additional data fields and methods may be added to a class as needed. Remember to make sure that your class describes ONE simulation component – keep things together that go together but also pull apart information that belongs to different simulation elements. The required simulation classes are listed below and then each class is described individually.

Program input requirements: For Lab #4, the initial set up data MUST be read from files. The format for the files will be given later in this document. The input files should contain all the data needed to completely initialize or set up the restaurant, ready to open its doors for the day. Once the setup is complete, the simulation should switch to operation mode with the menu choices read from an input file as described later or selected from a menu of choices.

Classes to define: (Some additions are made to all the classes for Lab #4)

Restaurant

Table

Kitchen

Person

Employee – subclass of Person

Maitre d’ – subclass of Employee

Server – subclass of Employee

Busser – subclass of Employee

Musician – subclass of Employee

Manager - subclass of Employee

KitchenStaff - subclass of Employee

Chef - subclass of Employee

Patron – subclass of Person

TableOrder

MenuOrder

MenuItem

PDate – class is modified to use GregorianCalendar

Address

BankAccount

Supplier

Enumerations to define:

TablePatronStatus

MenuItemCategory

Job Title

Pay Type

Shift

Employee status

Pay method

Order Status

BankAccountType

BankaccountOwner

Interface to define:

Payable

Restaurant class should contain the following data and methods:

Data members:

1. Has a name

2. Has a maximum number of tables

3. Has maximum number of patrons

4. Has a current list of tables

5. List of available menu items

6. Has current list of employees

7. Has current list of maitre d’ that are present and working

8. Has current list of servers that are present and working

9. Has current list of bussers that are present and working

10. Has total number of seated patrons – should be calculated not entered directly

11. Has total number of waiting patrons – should be calculated not entered directly. This number should stay low.

12. Has a current shift – see enumerated type below

13. Has an address – use the Address class below

14. Has current list of managers that are present and working

15. Has a bank account – use the Bank class below

16. Has a current list of musicians that are present and working

Methods: (in addition to ALL of the standard methods that should be included)

a. Add a waiting patron to the total (when patrons arrive at restaurant)

b. Move a patron (or group of patrons) from waiting total to seated total (should be linked to actions e and f)

c. Remove a table from a server after the table is empty

d. Assign a table to a server when patrons are seated

e. Update number of patrons at a table when patrons are seated (should be linked to actions b and f)

f. Update the table patron status when patrons are seated from EMPTY to SEATED total (should be linked to actions b and e)

h. Accept payment for a bill for a table - Update amount in restaurant bank account

i. Update number of patrons at a table when patrons have paid and left

j. Print list of all employees including at least first name, last name, job title and hire date

k. Print list of all currently working employees giving first name, last name, job, and list of tables.

l. Pay all accounts using Payable interface (employees and suppliers) - Update amount in bank account

z. Constructor – must have one constructor that accepts input information to define initial values for name, max tables, and max patrons, then the constructor must accept input to create a current list (array or arrayList) of Tables, a current list of MenuItems that are available, the current shift, and a current list of employees with enough information to indicate their job title and whether they are currently present and working or not. The bank account should be initialized with $100,000.

Table class should contain the following data and methods:

Data members:

1. Table number

2. Maximum number of patrons at this table

3. Current number of patrons at this table

4. Assigned server

5. Assigned busser

6. Table patron status – this is an enumerated value indicating what action is occurring at the table. The status action values must occur in order for a table. The enumerated status values are {SEATED, DRINKS, ORDERS, SERVED, EATING, WAITING, BILLED, PAID, LEFT, EMPTY}. These values will be described in detail in the enumeration description below.

7. A TableOrder – an object of class TableOrder below

8. Number of patrons that have received their menu items ordered

9. A timestamp using PDate of hh:mm:ss when the table had the current party SEATED

Methods: (in addition to ALL of the standard methods that should be included)

a. Update current list of orders (set) – after ORDERS and after SERVED

b. Cleaned table is reset to initial conditions – set values of current patrons = 0, table patron status = EMPTY, list (array or arrayList) of orders has no contents, and number of patrons with items = 0

z. Constructor – must have one constructor that accepts input information to define table number, maximum number of patrons, then the assigned server and the assigned busser, then the initial table set up information of current patrons = 0, table patron status = EMPTY, list (array or arrayList) of orders has no contents, and number of patrons with items = 0

TableOrder class should contain the following data and methods:

Data members:

1. A list of MenuOrders

2. Table number for the MenuOrder

3. TableOrderUP? Is every menu order at status UP? Boolean: true means entire TableOrder is ready to serve (use a private method to determine this value)

4. Total cost of TableOrder (This is the bill for the restaurant visit. Only one bill will be given per table.)

5. A timestamp using PDate of hh:mm:ss when the TableOrder was recorded

6. A timestamp using PDate of hh:mm:ss when the TableOrder was served

Methods: (in addition to ALL of the standard methods that should be included)

a. Print a TableOrder giving names of each item ordered and the number ordered of that item

b. Print a TableOrder giving names of each item ordered and the status of that item

c. Check a TableOrder to determine if every MenuOrder status is equal to UP. If yes, set TableOrderUp? to true.

d. Calculate the total cost of the TableOrder from the MenuOrders

z. Constructor – must have one constructor that accepts input information to define table number, an empty list of MenuOrders, and TableOrderUp set to false

y. Constructor – must have one constructor that accepts input information to define table number, a set of MenuOrders, and TableOrderUp set to false

MenuOrder class should contain the following data and methods:

Data members:

1. Menu item number

2. Number of servings ordered

3. Special requests as a list of Strings

4. OrderStatus – an enumeration value { START, PREP, COOKING, PLATED, UP }

Methods: (in addition to ALL of the standard methods that should be included)

a. Display a MenuOrder with all category, name, description, cost, and number of servings and list the tablenumber

b. Calculate cost of the current menu order

z. Constructor – must have one constructor that accepts input information to define item number, number of servings, OrderStatus set to START, and any special requests

Menu item should contain the following data and methods:

Data members:

1. Menu item name

2. Menu item number

3. Menu item description

4. Menu item cost

5. Menu item category – an enumeration value {APPETIZER, SALAD, MAIN, SIDE, DESSERT, SPECIAL, FEATURED}

6. Menu item allergy issues

Methods: (in addition to ALL of the standard methods that should be included)

a. Print the MenuItem giving category, name, description and cost

b. Print the MenuItem allergy info giving name and allergy issues

Person class has the following attributes

Data members:

1. First name

2. Last name

3. Email address

Methods: (in addition to ALL of the standard methods that should be included)

z. Constructor – must have one constructor that accepts input information to define any first and last name, and e-mail address

y. Constructor – must have one constructor that accepts a string input as first name only.

Employee class should inherit from the Person class and contain the following additional data and methods:

Data members:

1. Employee ID number

2. Address – use the Address class below

3. Job Title enumerated as MAITRE\_D, SERVER, BUSSER, MANAGER, MUSICIAN

3. Pay Type enumerated as SALARY, HOURLY, CONTRACTED, OTHER

4. Total hours worked for the just completed week (last week)

5. Hours worked so far for the current week

6. Pay value (a money amount) which will be one of the following:

If employee is HOURLY then the pay value is a Pay rate per hour

If employee is SALARY then the pay value is a Weekly salary

If employee is CONTRACTED then the pay value is a Weekly contract amount

If employee is OTHER then the pay value is a One time payment

7. Total tips (a money amount) collected for the just completed week (last week)

8. Tips collected so far for the current week

9. Date of hire - use the PDate class below

10. Currently at work? Boolean status variable

11. Bank account – use Bank object below

12. Shift start time using PDate - If Currently\_at\_work is true, then this time relates to the current shift. If not currently at work, this time relates to MM DD hh:mm of next scheduled shift.

Methods: (in addition to ALL of the standard methods that should be included)

a. Add to current week tips

b. Add to current week hours

z. Constructor – must have one constructor that accepts input information to define all the Person class fields using super to call the constructor for Person to define any first and last name and e-mail address. For the remaining Employee fields you may define employee ID, and address. Job title should be left blank and Pay type should be left blank. The remaining values should be filled in with the total hours worked last week, the hours that have been worked so far this week, the pay amount ($), the total tips collected last week ($), the tips collected so far this week ($), a date of hire, and a Boolean indicating if the employee is currently at work. Put $100 in the Employees bank account initially.

Maitre\_D class should inherit from the Employee class and contain the following additional data and methods:

Data members:

1. Last date worked - use the PDate class below

2. Shift preference – enumeration of LUNCH, AFTERNOON, EARLY\_DINNER, LATE\_DINNER

Methods: (in addition to ALL of the standard methods that should be included)

a. Accept new patrons to restaurant - Update total number of waiting patrons

b. Seat waiting patrons - Update table patron status to SEATED for table when patrons are seated, update other table info and update total number of waiting and seated patrons

c. Cash out a bill for a table - Update table patron status from BILLED to PAID for a table

d. Use Payable interface to calculate the Maitre\_D's weekly paycheck amount by:

If hours per week >= 40

Pay = weekly salary

Else

Pay = weekly salary \* (hours per week/40)

z. Constructor – must have one constructor that accepts input information to define all the Employee and Person class fields using super to call the constructor for Employee. For the Maitre\_D, Job title should be MAITRE\_D and Pay type should be SALARY. The remaining Maitre\_D field values should give the date of last day this person worked and a single shift choice.

Server class should inherit from the Employee class and contain the following additional data and methods:

Data members:

1. Maximum hours allowed per week

2. List of 4 shift preferences in order – use enumeration of LUNCH, AFTERNOON, EARLY\_DINNER, LATE\_DINNER

3. List of assigned tables

Methods: (in addition to ALL of the standard methods that should be included)

a. Calculate a bill for a table using TableOrder and deliver bill - Update table patron status from EATING or from WAITING to BILLED for tables in list of assigned tables

b. Get drink orders from seated patrons - Update table patron status from SEATED to DRINKS for tables in list of assigned tables

c. Get patrons’ orders of items from the menu (food orders) and deliver to kitchen - Update table patron status from DRINKS to ORDERS and create TableOrder for the specific table

c1. Check on table order status – If TableOrderUp is true, retrieve food for the table

d. Bring food from the kitchen to the table - Update table order status from ORDERS to SERVED for tables in list of assigned tables

e. Check on the table - Update table patron status from SERVED to EATING for tables in list of assigned tables

e1. Check on the table - Update table patron status from EATING to WAITING for tables in list of assigned tables

f. Recognize that patrons have left the table - Update table patron status from PAID to LEFT for tables in list of assigned tables

g. Using the Payable interface, calculate the Server's weekly paycheck amount by:

If hours per week <= 40

Pay = pay per hour \* hours worked in the week

Else

Pay = pay per hour \* 40 +

pay per hour \* 1.5 \* (hours worked in the week – 40)

z. Constructor – must have one constructor that accepts input information to define all the Employee and Person class fields using super to call the constructor for Employee. For the Server, Job title should be SERVER and Pay type should be HOURLY. The remaining Server field values should give the maximum hours per week this server may work, an ordered list of shifts sorted from most preferable to least preferable, and a list (array or arrayList) of table numbers that are assigned to this server.

Busser class should inherit from the Employee class and contain the following additional data and methods:

Data members:

1. Age

2. Employee status – PROBATIONARY or PERMANENT

3. List of assigned tables

Methods: (in addition to ALL of the standard methods that should be included)

a. Clean and reset a table - Update table patron status from LEFT to EMPTY for tables in list of assigned tables when table has been reset

b. Using the Payable interface, calculate the Busser's weekly paycheck amount by:

pay per hour \* hours worked in the week

z. Constructor – must have one constructor that accepts input information to define all the Employee and Person class fields using super to call the constructor for Employee. For the BUSSER, Job title should be BUSSER and Pay type should be HOURLY. The remaining Busser field values should give the employee’s age, the status, and a list (array or arrayList) of table numbers that are assigned to this busser.

KitchenStaff class should inherit from the Employee class and contain the following additional data and methods:

Data members:

1. Age

2. Kitchen role (from enumerated type PREP, SOUS, DESSERT)

Methods: (in addition to ALL of the standard methods that should be included)

a. Using the Payable interface, calculate the Kitchen staff person's weekly pay amount by:

pay per hour \* hours worked in the week

z. Constructor – must have one constructor that accepts input information to define all the Employee and Person class fields using super to call the constructor for Employee. For the KITCHEN staff, Job title should be KITCHEN and Pay type should be HOURLY. The remaining Kitchen staff field values should give the employee’s age and a their role in the kitchen.

Chef class should inherit from the Employee class and contain the following additional data and methods:

Data members:

1. Last date worked - use the PDate class below

2. Shift preference – enumeration of LUNCH, AFTERNOON, EARLY\_DINNER, LATE\_DINNER

Methods: (in addition to ALL of the standard methods that should be included)

a. Accept new MenuOrders from TableOrders – update status to START for all MenuOrders

b. Move TableOrders through the kitchen – update START to PREP

c. Update status of a MenuOrder from PREP to COOKING

d. Update status of a MenuOrder from COOKING to PLATED

e. Update status of a MenuOrder from PLATED to UP

f. Use Payable interface to calculate the Chef's weekly paycheck amount by:

percentage of net profit \* restaurant bank account amount

z. Constructor – must have one constructor that accepts input information to define all the Employee and Person class fields using super to call the constructor for Employee. For the Chef, Job title should be CHEF and Pay type should be OTHER. The remaining Chef field values should give the date of last day this person worked and a single shift choice.

Manager class should inherit from the Employee class and contain the following additional data and methods:

Data members:

1. Percentage of net profit as a floating point number

2. Boolean value indicating if this manager has authority to write checks for payables

Methods: (in addition to ALL of the standard methods that should be included)

a. Cash out a bill for a table and add amount to restaurant bank account - Update table patron status from BILLED to PAID for a table

b. Write checks for payables

c. Using the Payable interface, calculate the Manager's weekly paycheck amount by:

percentage of net profit \* restaurant bank account amount

z. Constructor – must have one constructor that accepts input information to define all the Employee and Person class fields using super to call the constructor for Employee. For the Manager, Job title should be MANAGER and Pay type should be OTHER.

Musician class should inherit from the Employee class and contain the following additional data and methods:

Data members:

1. Number of musicians

2. Name of group

Methods: (in addition to ALL of the standard methods that should be included)

b. Use the Payable interface to calculate the Musician's weekly paycheck amount by:

contracted weekly pay amount

z. Constructor – must have one constructor that accepts input information to define all the Employee and Person class fields using super to call the constructor for Employee. For the MUSICIAN, the First and Last names will be for the musician who handles the contracts, Job title should be MUSICIAN and Pay type should be CONTRACTED. The remaining Musician field values should give the name of the music group and the total number of group members.

Patron class should inherit from the Person class and contain the following additional data and methods:

Data members:

1. First time patron?

2. Pay method enumerated as CREDIT, CASH, DEBIT, GIFT, COMP, NONE (patron may be paying or may be a non-paying guest of the paying patron – method is NONE) The value of this enumerated type will indicate if this person is the paying patron or a non-paying patron

3. Table number

4. Cost of restaurant visit (assumes one bill per table) Will be either $0.00 for a non-paying patron or the cost of TableOrder for the paying patron

Methods: (in addition to ALL of the standard methods that should be included)

a. Calculate a tip to add to the bill

b. Make a payment for the bill – can be an amount greater than the total of the bill

Kitchen class should contain the following additional data and methods:

Data members:

1. List of TableOrders

Methods: (in addition to ALL of the standard methods that should be included)

//REMOVE and move to Chef class

//a. Update status of a MenuOrder from START to PREP

//b. Update status of a MenuOrder from PREP to COOKING

//c. Update status of a MenuOrder from COOKING to PLATED

//d. Update status of a MenuOrder from PLATED to UP

e. Check status of a TableOrder and set TableOrderUp to true if all MenuOrders are true

z. Constructor – must have one constructor that creates an empty list of TableOrders if no list exists.

y. Constructor – must have one constructor that adds a new TableOrder to the existing list of TableOrders if list exists.

Bank class should contain the following data and methods:

Data members:

1. BankAccountNumber – an integer greater than 10,000 of at least 6 digits

2. BankAccountAmount – a floating point value that can be positive or negative

3. BankAccountType {enumerated value of CHECKING, SAVINGS, OTHER}

4. BankaccountOwner {enumerated value of PERSONAL, CORPORATE, NONPROFIT, ACADEMIC, OTHER}

5. BankDebitCard? – Boolean true if user has a debit card on this account

Methods: (in addition to ALL of the standard methods that should be included)

a. Add funds to the account

b. Subtract funds from the account

c. Print bank account information of account number, account type, account owner, and account amount

z. Constructor – must have one constructor that takes an input of 0 and creates a new account with a random number greater than 10,000; an amount of $100; a CHECKING type; a PERSONAL owner; and true for the DebitCard

y. Constructor – must have one constructor that takes an input of 1 and creates a new account with a random number greater than 10,000; an amount of $100,000; a CHECKING type; a CORPORATE owner; and false for the DebitCard

x. Constructor – must have one constructor that takes an input of 2 and creates a new account with a random number greater than 10,000; an amount of $1000; a CHECKING type; a NONPROFIT owner; and false for the DebitCard

w. Constructor – must have one constructor that takes an input of 3 and creates a new account with a random number greater than 10,000; an amount of $10,000; a CHECKING type; a ACADEMIC owner; and false for the DebitCard

v. Constructor – must have one constructor that takes an input of 4 and creates a new account with a random number greater than 10,000; an amount of $10; a SAVINGS type; a OTHER owner; and false for the DebitCard

PDate class should contain the following data and methods:

Data members:

1. GregorianCalendar (or Calendar) object that will record Date and Time information

// REMOVE previous Date and Time info to replace with GregorianCalendar object

//1. Date including - default date is January 1, 2015, 1-1-2015, 1/1/2015

// a. Month as a string fully spelled out

// b. Month as a number from 1 to 12

// c. Day as a number

// d. Year as a four digit number

// e. Day of week as a string ("Thursday", "Sunday", etc.)

//2. Time including – default time is 00:00

// a. Hour using a 12 hour clock (1 to 12)

// b. Minute as a number between 0 and 59

// c. Morning or evening designator string "AM" or "PM"

// d. Hour using a 24 hour clock (0 to 23)

Methods:

a. Print date as named month string: January 16, 2015

b. Print date as integers only in month, day, year format: 1-16-2015 or 1/16/2015

c. Print time as 12 hour clock with am or pm – 5:43PM

d. Print time as 24 hour time – 17:43

e. Print date and time (you may select formats)

f. Print date and time as mm-d-yyyy hh:mm:ss

Address class should contain the following data and methods:

Data members:

1. Street number as an integer

2. Street name as a string

3. Street designator as a single word string (“Ave.”, “St.”, “Blvd”, “Road”, etc. )

4. String (on a second line) with additional address information if needed. Can be blank.

5. City name as a string

6. State/Province abbreviation as a two letter code (see US Postal abbreviations)

7. Postal code/Zip code as an integer (no error checking is required for Lab #3)

8. Country name as a string

Methods: (in addition to ALL of the standard methods that should be included)

a. Print entire address in standard US Mail format (ask me if you need more info)

b. Print only the city and state in the format “City, StateCode”

Supplier class should contain the following data and methods:

Data members:

1. Company name

2. Supply category (MEAT, VEGETABLES, BAKERY, LINEN)

3. Supplier ID – 5 or more digits starting with 444

4. Address – use the Address class above

5. Contact person

6. Has a bank account – use the Bank class below

Methods: (in addition to ALL of the standard methods that should be included)

a. Calculate an invoice amount for supplies using the Payable interface. This is the amount of money that the restaurant owes the supplier. (For purposes of Lab #4, this can be amount given by the user or a defined amount.)

b. Accept payment for an invoice - Update amount in supplier bank account

z. Constructor – must have one constructor that accepts input information to define initial values for name, category, address, contact person and bank account. The bank account should be initialized with $10,000.

Enumerations:

TablePatronStatus has the following enumerated values with the following meanings

{SEATED, DRINKS, ORDERS, SERVED, EATING, WAITING, BILLED, PAID, LEFT, EMPTY} . These values must occur in the order they are listed. For Lab #3, if a status is changed that is out of order, then the status change is invalid and should not be allowed. This assumption will change in later labs.

SEATED – patrons have been seated at the table but have had no service yet

DRINKS – patrons have ordered and received drinks but not ordered menu items yet

ORDERS – patrons have ordered from the menu but have not received their menu items

SERVED – patrons have been served the items they ordered. Once all patrons are served, the status moves to EATING

EATING – patrons are eating and have not yet received a bill

WAITING – patrons are waiting for a bill (this status may be skipped at some table)

BILLED – a bill (or bills) has been delivered to the patrons at the table

PAID – the patrons have paid the bill(s)

LEFT – the patrons are no longer seated at the table but the table has not been bussed and reset

EMPTY – the table is empty, has been reset and is ready for patrons. The status that can follow EMPTY is SEATED

MenuItemCategory has the following enumerated values {APPETIZER, SALAD, MAIN, SIDE, DESSERT, SPECIAL, FEATURED}

Job Title { MAITRE\_D, SERVER, BUSSER, MANAGER, MUSICIAN, KITCHEN, CHEF }

Kitchen Role { PREP, SOUS, DESSERT }

Pay Type { SALARY, HOURLY, CONTRACTED, OTHER }

Shift { LUNCH, AFTERNOON, EARLY\_DINNER, LATE\_DINNER }

Employee status { PROBATIONARY, PERMANENT }

Pay method { CREDIT, CASH, DEBIT, GIFT, COMP }

OrderStatus { START, PREP, COOKING, PLATED, UP }

BankAccountType { CHECKING, SAVINGS, OTHER }

BankaccountOwner { PERSONAL, CORPORATE, NONPROFIT, ACADEMIC, OTHER }

Supply category {MEAT, VEGETABLES, BAKERY, LINEN}

Interface to use:

public interface Payable()

{

double CalculatePay(double Amt1, double Amt2);

}

NOTE: at this point, you should be thinking about what additional classes or additional data and methods might be needed to implement this. There is no one right way to do this. Also as you think about testing this simulation, you will notice that in this assignment there are still some “loose ends” such as tracking hours of work to get paid or food preparation. Just implement what is required but don’t try to make everything tie together or try to close all the loops in the system. You will be building on this system so more elements will be added later.

For Lab #4, your simulation must allow the user to perform the following required scenario actions from the categories of Set Up Actions, Simulation Actions and Simulation Displays). The scenario actions (the actions from all categories together) are the menu actions from Lab #3. These actions can be input from a file (format described below) or selected from a scenario menu. In Lab #4 you may choose to have separate actions menus for each category, i.e. three menus or “Set Up” and then two menus. The scenario input from the file or the scenario menu(s) should be used by your main routine to control the simulation.

Scenario Menu choices: The scenario file will use the numbers below to indicate the actions. Note that some numbering has been modified below to give each action a unique integer number. The actions have been separated into Set up, Simulations Actions, and Simulations Displays but most have retained their earlier numbering. The file format and menu for actions is described at the end of the list of actions.

Set Up Action:

-1) Set up the restaurant including tables and employees and suppliers – may only be run once. Must read data from the input files. Input file must be in the format specified below.

Simulation Actions:

0) Have a party of patrons enter the restaurant [Create one or more patrons for the restaurant] AND indicate number of patrons. IF NO BIG ENOUGH TABLE IS AVAILABLE, Keep this action on hold and recheck for an available table after every following action. Eventually, a table will be emptied that is big enough and then this group should get seated.

1) Have a maître d’ seat a party of patrons at a table [Seat patrons at a specific table #] <If this action is chosen, but no patrons are waiting or no tables are available, then skip this action. This caveat applies to all other simulation actions except number 0>

2) Have a server take drinks requests at a table [Must select a server and give a table number for all the actions by the server/busser below]

3) Have a server take menu item orders at a table

31) Have a server check on menu items ordered for a table. Table must have already ordered. Table status does not change.

4) Have a server deliver menu items to a table

5) Have a server check on a table where patrons have been served and are eating

51) Have a server check on a table where patrons have mostly finished eating and are waiting for the bill

6) Have a server calculate a bill and deliver it to a table where patrons are finished eating

7) Have a maître d’ accept payment for a bill and then indicate that the bill is paid for a table

8) Have a party of patrons leave a table and the restaurant

9) Have a busser clean a table and reset it for the next party of patrons

16) Have manager pay the employees and suppliers using the Payable interface

161) Have manager pay one employee or supplier using the Payable interface

17) Have a manager accept payment for a bill and then indicate that the bill is paid for a table

23) Have the chef in the kitchen start a menu item ordered

24) Have the chef in the kitchen prep a menu item ordered

25) Have the chef in the kitchen cook a menu item ordered

26) Have the chef in the kitchen plate a menu item ordered

27) Have the kitchen signal that a menu item is ready to serve, i.e. the order is up

32) Have a supplier submit an invoice to be paid by the restaurant

//REMOVE

// 22) Have the kitchen make a menu item ordered – this moves the menu order status

// from start to ready to serve , i.e. the order is up [This jumps through the entire sequence

// of the status for the menu order.]

Simulation Display Actions:

10) Display information about the restaurant including number of tables and current number of patrons

11) Display information about one table in the restaurant including current status, number of patrons, list of orders, assigned server and busser.

12) Display information about a server including name, preferred shifts in order, hourly pay rate (the pay value), current list of assigned tables, and an indication if the server is currently working or not. (Can display more data if desired.)

13) Display information about a busser including name, employee status, hourly pay rate (the pay value), current list of assigned tables, and an indication if the busser is currently working or not. (Can display more data if desired.)

14) Display information about a maître d’ including name, preferred shift, weekly pay rate (the pay value), and an indication if the maître d’ is currently working or not. (Can display more data if desired.)

15) Display a current snapshot of restaurant activity giving the shift name, the number of waiting patrons, and all the table information about all tables in the restaurant. For Lab #4 you must implement display action 15 with graphical elements and text, i.e you must use some sort of diagram with text in a GUI window. See an example graphic at the end of the assignment.

18) Display information about a manager including name and weekly pay rate (the pay value). (Can display more data if desired.)

181) Display information about a musician including name, group name, pay rate (the pay value), and an indication if the musician is currently working or not. (Can display more data if desired.)

19) Display information about the kitchen giving number of TableOrders in the kitchen and which tables have orders in the kitchen

20) Display information about the kitchen listing one complete TableOrder by listing all the MenuOrders giving the item(s) ordered, quantities, and the status of the MenuOrders.

21) Display all TableOrders in the kitchen

28) Display all Person and Employee information about any employee in the restaurant [Use existing methods for 12, 13, 14,18, 181 to do this.]

29) Display information about one supplier company

30) Display one invoice from a supplier

99) End the program

**Program Requirements**

For Lab #4 you should modify your previous Lab #3 to:

1. Update and define classes for all the described elements above,
   1. use inheritance to create at least two of the classes as defined,
   2. keep data members of a class private unless otherwise instructed
   3. create get (accessor) and set (mutator) methods for all private data
   4. create toString and/or display methods for all classes
   5. create constructors and initialize objects as needed

Input file formats: The input data for your initial restaurant set up must be stored in a text files. The input for one test run will be in five separate files. The first file will be “lab4InputT.txt” (notice the capital ‘I’ in the name). This file will have the same format of data as the “lab3Input.txt” file. The second file will be “lab4InputTaddr.txt” This file will have addresses for the employees in the first file. The third file will be “lab4InputTco.txt” This file will have supplier company information. The fourth file is a file of menu items named “lab4InputTmenu.txt”. This file is required for Lab #4. The fifth and final file will be the scenario file with actions listed and will be called “lab4Scenario.txt”. Information about each file is given below.

You should create a variety of input data files with names like “dataset1”, “dataset2”, etc. and then copy whichever dataset you want to use into the file named “lab4Input.txt” and the other files in order to run a test of your program. I may post a sample data file to test with also.

The format of the file is given as a description of the data that is put on each line and an example of that data:

EXAMPLE DATA Line #

String name of restaurant Dr T’s Treats 1

Integer number of tables 4 2

Max patrons per tables in order by table # 2 4 4 6 3

# of SERVERs, BUSSERs, MAITRE\_D, MUSICIANS 2 1 1 3 4

String first names of SERVERs with table #s VJ 1 4 Chris 2 3 5

String first names of BUSSERs with table #s Jo 1 2 3 4 6

String first names of MAITRE\_Ds Su 7

String first name of MANAGER Gwendolyn 8

String first names of MUSICIANS Rocky Fred Samson 9

Employee info string 10 to end

of file

[Example Employee info strings: Remember to put them in your file without a line break. Let them wrap to two or three lines if needed.]

VJ Murtha vjm@gmail.com 100842 Server 38 22 5.95 154.25 87.00 7 2 2013 true 39 LATE\_EVENING EARLY\_EVENING AFTERNOON LUNCH

Jo Smythe jsmth99@gmail.com 100443 Busser 22 18 3.95 42.95 11.10 9 1 2014 true 14 Probationary

Su Caster scast@yahoo.com 100532 Maitre\_D 44 13 450.00 0 0 2 5 2012 true 9 19 2014 EARLY\_EVENING

This final Employee info string gives an employee named Su Caster with e-mail address scast@yahoo.com, employee ID 100532, job type MAITRE\_D, 44 hours worked last week, 13 hours worked so far this week, a weekly salary amount of $450.00, no tips last week, no tips this week, date of hire as Feb. 5, 2012, Su is currently at work, the last date Su worked was Sept 19, 2014, and Su’s preferred shift is EARLY\_EVENING

Notes on the content of the first input file “lab4InputT.txt”:

Line #3

Maximum number of patrons that can sit at each table as integers listed in a row with spaces between. There should be as many integers as there are tables.

After set up all Tables will have status empty and should have a max size defined. For this input file, the following info would be true after set up and before patrons enter:

Table 1 is EMPTY with max seats of 2

Table 2 is EMPTY with max seats of 4

Table 3 is EMPTY with max seats of 4

Table 4 is EMPTY with max seats of 6

Lines #5 and #6

For server (or busser) first names and assigned tables, your program should read in the whole line, then split the line into smaller strings. The first string should always be a name, then the next string should be checked to see what table number it is with the table being assigned to that server (or busser) and the server (busser) being assigned to that table. The program should then check if the next string contains a number or not. If yes, add that table number to the current server (busser). If no, read the string as the name of the next server(busser) followed by table numbers. Continue until the string has been completely read.

Line #7

For maître d’, first names only separated by spaces. If maître d’ has a double first name such as “Billy Bob”, the name must be combined as a single string

Line #8

For manager, one first name

Line #9

For musicians, first names only separated by spaces. If musician has a double first name such as “Mary Lee”, the name must be combined as a single string

Line #10 until end of the file:

Multiple lines of Employee Info string: A list of values from the Person class, Employee class, and jobtype class. This data will NOT include the Address for the employee. The values in the Employee info string for ALL employees are:

First Last Email IDnum JobType HoursLast HoursCurrent PayValue TipsLast TipsCurrent HireMonth HireDay HireYear Working? Data1 Data2 Data3 Data4 Data5

The values of Data1..Data5 are specific to the job type.

For the Maitre\_d job type:

Data1 - Integer Month of last date worked

Data2 - Integer Day of last date worked

Data3 - Integer Year of last date worked

Data4 - String giving the shift preference –LUNCH, AFTERNOON, EARLY\_DINNER, or LATE\_DINNER (ignore the case when you read this string)

For the Server job type:

Data1 - Maximum hours allowed per week

Data2 – String giving the highest (first) shift preference

Data3 – String giving the next highest (second) shift preference

Data4 – String giving the next highest (third) shift preference

Data5 – String giving the lowest (fourth) shift preference

For the Busser job type:

Data1 - Integer age in years

Data2 - String giving employee status – PROBATIONARY or PERMANENT

For the Manager job type:

Data1 - Boolean value whether this manager can write restaurant checks for payables

For the Musician job type:

Data1 - Integer number of musicians in group

Data2 - String giving group name

The second input file “lab4InputTaddr.txt” will give the address information for the restaurant as the first set of data and then addresses for each employee in the restaurant. Each set of address information is given on three lines. Each address starts with the last name of the addressee, then first name (the very first entry is always the restaurant name as a single word so no first name is given for the first address entry) , followed by an integer for street number, a string for street name, and a single word street designator. NOTE that the string for street name must be one single word with no blanks such as Pine\_Tree or PineTree for the street named "Pine Tree". The street designator is as described in the Address class. Ex:

Last\_name First\_name Street# Street\_Name StrDes

The second line of the address is any additional info needed before the city info. This includes apartment numbers, location within a large complex, or other info. The second line is saved as one string and this line can be blank. Ex:

209 UTA Shopping Center OR

Block 18 O 9 OR

Bldg. 4 Apt. 1

The third line is in the format:

City name , two letter state/prov abbreviation postal code country name

This will be read as a string, a comma delimiter, a two letter abbrev., a numeric postal code, and a final string.

The third file will be “lab4InputTco.txt” and will have information about any supplier companies working with the restaurant. The company data will be in the form:

Line 1: String giving the company name

Line 2: Street# Street\_Name StrDes SupplyCategory

Line 3: CompanyIDnumber String of any additional data

Line 4: City name , two letter state/prov abbreviation postal code country name

In addition to the input files of restaurant data, you must use a fourth file to input menu data. This file is required for Lab #4. Follow the format below. You should name this file “lab4InputTMenu.txt”

Menu input file:

The format of the file is given as a description of the data that is put on each line and an example of that data:

EXAMPLE DATA Line #

Menu item info string 1 to end

of file

[Example Menu item info strings: Remember to put them in your file without a line break. Let them wrap to two or three lines if needed.]

Turtle Cheesecake 23 A New York style cheesecake dressed with dark chocolate and walnuts and drizzled with caramel 8.95 DESSERT lactose-intolerant nut-allergy

Steamed Alaskan King Crab 8 King crab caught yesterday in Alaska, steamed, and served with drawn butter and lemon 42.50 MAIN seafood-allergy

Caprese Salad 3 Heirloom tomato and locally sourced mozzarella with balsamic vinegar reduction and basil garlic oil 13.00 SALAD

This third item would be the Caprese Salad, menu item #3, described as “Heirloom tomato and locally sourced mozzarella with balsamic vinegar reduction and basil garlic oil”. The cost is $13.00 for this SALAD item and it has no allergy issues

The fifth and last file for Lab #4 is the scenario file called “lab4Scenario.txt”. This file will contain a list of scenario instructions to perform in the given order. The user should be offered the opportunity to run the simulation scenario from this file or from the scenario menu. The file of scenario instructions will have one instruction number on each line. Your simulation should read the instruction (scenario action) on the line of the file and then should look through your list of tables to see if there is a table that is ready for that instruction. If you find a table it applies to, then execute that action for that particular table.

As an example, the instruction in the file (or selected from the menu) might be 3 which is the action of having a server go to a table to take an order. This action only applies to a table that is in the DRINKS status. Therefore, if there is no table in the DRINKS status then the action is not applied and the next action is read from the file (or from the menu). If there is more than one table in the DRINKS status, your simulation will pick one table number (using any system to pick, like first table found or a random choice among tables with the correct status) and apply action 3 to that table, changing the DRINKS status to ORDERS status.

The scenario file should be read until all instructions are read and executed if possible. Once the file instructions have all been completed, give the user a message indicating that all the instructions from the file are done and the simulation run is ending. It is NOT required to have instruction 99 as the final instruction in the file.

For the scenario file, some special formatting applies:

a) for action 0 : patrons enter – specify number of patrons as second number following 0

In addition to the scenario file, you must also offer your user a scenario menu of all the action choices. This menu MUST be in a GUI. The selections on the menu are the same selections that that your user has previously chosen from on the menu list on the screen (plus any changes required above.) For Lab #4, this menu list MUST be implemented as either a set of radio buttons to choose from or as a drop down menu (combo box) in a GUI. You may have one GUI for simulation actions and another GUI for simulation displays or you may show the simulation action commands and the display commands all in one GUI. If the user is running the simulation scenario from the GUI, then the simulation continues to run until the user selects to end the simulation (action 99).

For both file input and GUI input of simulation actions and displays, give the user some visible feedback after each action. It can be a small thing but do indicate when an action is complete in some way. The user is expected to select display statements when they wish to have data displayed so it is not required that the system give detailed info at the completion of an action although it is fine if this is done.

Javadoc Required for Lab #4:

Java provides mechanisms for comments and input for Javadoc. Please make sure your code contains both. Also, submit the files generated by the Javadoc for your program. Look up the convention for specifying parameters (@param) and user information for javadoc. Every method should have its meaningful description for use by someone else. Use the relevant Javadoc tags specified in -http://en.wikipedia.org/wiki/Javadoc or <http://java.sun.com/j2se/javadoc/writingdoccomments>

BONUS: Simulation Analysis

For extra credit, you may record some data from your simulation and then find some information using this data. In particular, you may create a “simulation data” class that records the following data about each simulation scenario action:

Record the action (instruction) given in the file or selected by the user

Record the time that action occurred (using PDate)

If the action refers to a table (the table status actions) then

a) Record the number of the table that the action was applied to (or a value indicating that the instruction was not applied)

b) Calculate and record the time between this current table status action and the previous table status action on the same table [which will be another simulation data object like this one]

Make an array or a collection of these simulation data objects and capture all of the actions executed for the simulation. After the end of the file or when the user chooses to end the simulation from the GUI, find the maximum length of time, the minimum length of time and the average length of time between any two table status steps, for example the min, max, and average length of time between ordering meals (ORDERS) and receiving the meals (SERVED). Before ending the program, print out a table of the table status steps and the min, max, and average times between each pair of table status actions.

**Standard Java conventions and coding style:**

Be sure to observe standard Java naming conventions and style. These will be observed across all projects for this course, hence it is necessary that you understand and follow them correctly. They include but are not limited to:

* Class names begin with an upper-case letter, as do any subsequent words in the class name.
* Method names begin with a lower-case letter, and any subsequent words in the method name begin with an upper-case letter. This is called "camelCase".
* Class, instance and local variables begin with a lower-case letter, and any subsequent words in the name of that variable begin with an upper-case letter.
* Each set of opening/closing curly braces must be in the same column or must be consistently placed with opening brace at the end of a line and closing brace on a line by itself.
* All user prompts must be clear and understandable
* Give meaningful names for classes, methods, and variables even if they seem to be long. The point is that the names should be easy to understand for a person looking at your code
* All static final variables will be upper case identifiers (i.e.constants)

In addition, ensure that your code is properly documented in terms of comments and other forms of documentation wherever necessary.

**What and How to Submit**

For Lab #4 WRITE A DESIGN DOCUMENT FIRST. The design must include

a) all the classes you expect to write,

b) brief (one line) descriptions of each data member (data field) of the class

c) brief (one line) descriptions of each method (member function) of the class, and

d) some indication of the relationships between the classes.

The design document may be written as lines of text or as diagrams (such as a diagram that start with the main function at the top and all others below it, a tree, UML, etc.) or as some combination of those, but it must include the information listed for a), b), c), and d) above. UML is not required for Lab #4 but will be required for Lab #5. Each class should encapsulate one thing and each piece of data at a lower level should have a more specific purpose than the class that owns it. Be sure to include all the classes that are described in this lab assignment. See the website or this assignment for the DESIGN DOCUMENT due date. It is usually ONE WEEK PRIOR to the lab due date.

A Lab #4 Design Document must be turned in in order for your Lab #4 assignment to be graded.

For this lab project itself, please submit the following material using the Blackboard system at elearn.uta.edu. Please submit ONE .zip file that contains:

* Each .java file used in development with the appropriate name for the included class
* One or more .txt file(s) containing each set of test data that you used with the program – this would be the initial data you create, a menu item file if, and the set of scenario steps to run.
* A document, called your "test plan", listing each test case (each set of test data from the previous document) that you used to run and test your program and what the purposes are of each test (e.g. Your first test data set might include 12 64 1999 for a date. Your test plan should say then that this data set will test whether the day of the month is a valid value. You may test more than one thing (lots of things) with a single test data set.)
* A development notes file (standard text or word file) that describes at least three ­potential problems that the current project would have given the format of the input data in the file (i.e. what, in the real world, might cause problems with your simulation right now in terms of the input data) and the total amount of time you spent on this project in terms of hours.

The .zip file containing your project must have the name format of: **lab4\_100number\_firstname\_lastname**

Upload this code to Blackboard at elearn.uta.edu

**A Few Helpful Tips:**

* Netbeans by default recompiles your class every time you save your file. If you want to force everything to be recompiled, press the ‘clean and build’ button.
* Ctrl + space in most IDE’s will show a list of available members (or variables) and methods (or functions).

**Lab #4 Grading Rubric:**

**Code: (46%)**

Documentation and coding style including: (5 points total)

Class header and method headers for all classes

Good Javadoc content

Comments (line comments and block comments)

Style (following Java conventions, indentation, consistency, meaningful identifiers, lateral separation of code from line comments, etc.)

Modularity (division of the problem into small parts)

Correct declaration and use of the Payable interface and method CalculatePay() in multiple classes (4 points)

Correct declaration of the new required classes with data and methods and any other needed classes

KitchenStaff, Chef (3 points)

Supplier (2 points)

Correct modification of the existing required classes

PDate using GregorianCalendar (3 points)

Classes implementing Payable (4 points)

Other classes (2 points)

Additional enumerations created for Kitchen Role and Supply Category and enum update of Job Title (2 points)

All previous simulation actions (menu actions) are implemented (7 pts)

Newly defined simulation actions added and modifications made to existing actions as defined (4 points)

File of simulations actions has correct format and at least 30 action steps which are different from the simulation input file provided by Dr. T (3 pts)

Four setup input data files have correct format and a set of initial test data completely different from the initial restaurant test data given in the assignment or in Dr. T’s sample data file if any is given (3 points)

Test plan includes at least two tests for each of the required scenario functions and uses the display functions to verify test results (4 points)

**Output: (51%)**

User clearly understands what is being requested for input (2 points)

File input with 4 files works correctly for initial set up (7 points)

User choice implemented for GUI-based simulation actions menu or for file simulation actions to be input (3 pts)

Required scenario tasks performs correctly

previous actions from Lab #3 (7 points)

new scenario actions for Lab #4 (7 points)

Graphical GUI ouput is given for restaurant snapshot display action (5 pts)

“set” functions and “get” functions perform appropriately (3 points)

Output gives clear information to explain the values to the user (3 points)

Output contains all the given test data (if any) and at least two additional data sets, i.e at least two other restaurants which each have at least four tables, and at least two servers (7 points)

Output demonstrates all elements of test plan (7 points)

**Other Requirements: (3%)**

Javadoc correctly implemented (3 points)

**OPTIONAL: Extra Credit Bonus: (5%)**

Simulation analysis data recorded (2 points)

Data analysis done to determine max, min, and avg (2 points)

Simulation data analysis displayed well to user (1 point)

**Grading Deductions:**

Labs which do not compile in NetBeans will receive an overall grade of 0 (zero) [-100 deduction]

Labs which have errors in them and do not terminate normally in NetBeans will receive an overall grade of 0 (zero) [-100 deduction]

Use of Java language elements not yet discussed in class by the lab due date will result in potential deduction of points - discuss with instructor before using.

Use of Java public variables (C equivalent to global variables) will result in an overall grade of 0 (zero) [-100 deduction]

Use of unstructured control elements, like the break (except in the case of switch statements), continue, goto, or exit keywords, will result in an overall grade of 0 (zero) [-100 deduction]

Additional grading notes:

Late submission of softcopy of design document, code and/or other documents to Blackboard will result in an overall grade of 0 (zero) UNLESS student has obtained prior instructor approval [-100 deduction] (No request is accepted more than 24 hours in advance)

There is a lab re-grading policy on the class website and on Blackboard. See this for information on how to REGAIN points that have been lost in a lab assignment.

Example GUI (Graphical User Interface) showing a current snapshot of restaurant activity (action number 15) is given below [May be on next page if looked at in print layout view]. The actual GUI you design can be more or less detailed or simpler than the one shown, but it should have some graphic elements, i.e. a drawing or diagram, and some text elements. It is not necessary to put labels on the drawing/diagram IF you can clearly match items in the drawing to items in the text. For example in the given GUI, if there were no labels in the drawing, then the elements would need to be defined in the text somehow such as:

Table 1 – small, square two person table

Table 2 – round four person table at bottom of diagram

Table 3 – round four person table at top of diagram

Table 4 – large rectangular table for six

Areas/Elements in the diagram would also need to be named somehow such as:

Entrance doors – a set of double doors in lower right corner of diagram represented by two angled lines facing each other and extending inward from the restaurant wall

Kitchen – area to left side of diagram separated from tables by a wall

Maitre d’ station – small black rectangle in lower right of diagram near the entrance doors

Restaurant: Dr T’s Treats

Current time: 11:43am Current shift: LUNCH

Maitre d’ on duty: Su Caster

Manager on duty: Gwendolyn Un

Servers on duty: VJ Murtha, Chris Chenow

Bussers on duty: Jo Smythe

Music group? The Big Trio

Table 1 Table 2 Table 3 Table 4 Patrons

Max Seats 2 4 4 6 waiting?

Patrons at 2 3 2 0 0

Status BILLED SEATED ORDERS EMPTY

Server VJ Chris Chris VJ

Busser Jo Jo Jo Jo

z

**4**

**3**

**2**

**1**

**Kitchen**

**Maitre d’**

Miscellaneous: If you have questions, e-mail Dr. T ([tiernan@uta.edu](mailto:tiernan@uta.edu)) and the TA.