CAB302 Software Development

Assignment 2: Test-Driven Development and Graphical User Interface Programming

Semester 1, 2017

Due date: Friday 2/6/2017 11:59 PM

Weighting: 35% [Marked out of 35].

Assessment type: Pair assignment

Version: 1.0

BitBucket Repository: https://bitbucket.org/cab302/asgn2

Change History

Date	Change Detail	Version
25/04/2017	Initial version	1.0
05/05/2017	Added detail about using relative directories for testing log files	1.1
07/05/2017	Remove discussion of record-id and pizza size	1.2

IMPORTANT:

Assignment 2 uses a class only found in Java 1.8 (LocalTime). Please make sure that your version of Java is up to date.

Make sure to place all log files in the "logs" directory and to reference them via a relative path for example: ".//logs/20170101.txt ".

Section 1: Learning Objectives

- To experience team-based program development using an approach similar to test driven development.
- To develop a Graphical User Interface.
- To develop your skills in object oriented design including: inheritance, encapsulation and polymorphism.
- To develop your skills implementing software patterns.
- To demonstrate your ability to write clear, understandable program code.

Section 2: Your Tasks

You must obtain the incomplete source tree from the assignment 2 repository on BitBucket. The link to obtain the repository is listed above.

- You will need to provide a solution to a problem written in the Java programming language.
- The necessary functionality of your solution is described in this specification document.
- In addition, you will be provided with a 'Javadoc' Application Programming Interface specification.
- You must develop comprehensive test suites in JUnit for the classes produced by your partner.

Section 3: Scenario

Pizza Palace is a restaurant in West End, Brisbane. Last year, they had a group of students from Kingsland University of Technology develop a logging system that recorded orders placed at their restaurant. They would now like to produce a system that interprets the logs and displays the information on a Graphical User Interface (GUI).

Section 4: Downloading Code

You need to download some pre-existing code from a repository on BitBucket. The link for the repository is provided on the first page. You should import the repository into Eclipse using the steps outlined in the GitForPracs document available on Blackboard. You can find this document using the following steps: Blackboard->Learning Resources-> Help Guides and Other Resources.

Section 5: Problem Details

Your task is to write a Java program to analyse log files and display the derived information to a GUI. Each log file contains information about a set of pizza orders split into two main sections:

1. Information about the pizza being ordered; and
2. Information about the customer ordering the pizza.

There are certain constraints regarding the pizzas and customers which your program must adhere to. Here, we describe the constraints relating to pizzas and the customers, then we outline the log file and finally, describe the functionality required for the GUI.

Section 5.1 Pizzas

Here are some details about the restaurant's pizzas:

1. The restaurant sells three types of pizzas: margherita, vegetarian and meat lovers. The toppings of each of the pizzas and prices are presented in Table 1. No other type of pizzas are available nor are any changes to the pizzas allowed.

2. Each topping has a set unit cost for being included on a pizza as shown in Table 2. No other toppings are allowed.
3. The restaurant keeps track of the time that a pizza was ordered and delivered. A pizza takes at least 10 minutes to cook and is thrown out after 1 hour (including delivery time).
4. The restaurant begins taking orders at 7:00pm (when the kitchen opens) and stops taking orders at 11:00 pm (when the kitchen closes) Australian Eastern Standard Time. Deliveries may continue after 11:00 pm.
5. At least one pizza must be ordered for the order to be valid.
6. A customer may order more than one of the same type of pizza within the same order. However, if they want to purchase another type of pizza, then that must be in a separate order.
7. The maximum number of pizzas in any one order is 10.
8. You need to track details about the pizzas being ordered. For example, you need to record how much each pizza costs to make (calculated from its toppings) and how much each pizza sells for. You also need to calculate the order cost (cost per pizza multiplied by quantity of pizzas ordered), the order price (sales price per pizza multiplied by quantity of pizzas ordered) and the profit made per order (order price minus order cost).
9. Each pizza contains a human description of their type (either "Margherita", "Vegetarian" or "Meat Lovers").

Pizza	Topping	Price ($)
Margherita	Tomato, Cheese	8
Vegetarian	Tomato, Cheese, Eggplant, Mushroom, Capsicum	10
Meat Lovers	Tomato, Cheese, Bacon, Pepperoni, Salami	12

Table 1: Pizzas, Ingredients and Prices

Topping	Cost ($)
Cheese	1
Tomato	0.5
Bacon	1.5
Salami	1
Pepperoni	1
Capsicum	1.2
Mushroom	2
Eggplant	0.8

Table 2: Ingredients and Costs

Section 5.2 Customers

Here are some details about the restaurant's customers:

1. There are three types of customers: those who come to the restaurant to pick up their pizza(s), those who have their pizza(s) delivered by a driver and those who have their pizza(s) delivered by a drone.
2. Each customer has a name, a mobile number, a location specified in x and y co-ordinates and a human understandable description of their type (either "Pick Up", "Driver Delivery" or "Drone Delivery").

3. The name of the customer is between 1 and 20 characters long and cannot be only white spaces.
4. The mobile number must be 10 digits long and begin with '0'.
5. The x and y location of the restaurant is 0, 0.
6. The customer's x location indicates the number of blocks east or west of the restaurant that the customer is located. A value of 5 means that the customer is located 5 blocks east, while a value of -5 means that the customer is located 5 blocks west.
7. The customer's y location specifies the number of blocks north or south of the restaurant that the customer is located. A value of 5 means that the customer is located 5 blocks north, while a value of -5 means that the customer is located 5 blocks south.
8. If the customer chooses to pick up their pizza then their location is always 0,0.
9. You need to calculate the distance travelled to deliver a pizza in terms of blocks. If the customer chooses to have their pizza delivered by a drone, then the distance is equal to the Euclidean Distance (shown in Formula 1) between the restaurant and the customer. If the customer chooses to have their Pizza delivered by a driver, then distance is equal to the Manhattan Distance (shown in Formula 2) between the restaurant and the customer. If the customer decides to pick up the pizza then the distance travelled is 0. A description of Euclidian and Manhattan distance is provide in Figure 1.
10. The restaurant will not deliver if the customer is at the restaurant.
11. The restaurant will not deliver is the customer is more than 10 blocks north, south, east or west from the restaurant – however, the restaurant will deliver a distance greater than 10 blocks. For example, a driver would deliver to a customer located at (5,6) despite have a Manhattan distance of 11.

$$distance_{\text{Euclidean}}(p,q) = \sqrt{(p_{x-}q_x)^2 + (p_{y-}q_y)^2}$$

Formula 1: Euclidean distance for points p= (p_x,p_y) and q = (q_x,q_y)

$$distance_{\text{Manhatten}}(p,q) = |p_{x-}q_x| + |p_{y-}q_y|$$

Formula 2: Manhattan distance for points p= (p_x,p_y) and q = (q_x,q)$_y$

Section 5.3 The Log Files

The log files are text documents formatted as comma separated value (CSV) files. A log is created daily, with its filename specifying its date. Each line holds a record of a single order placed by a customer. The format of each line, as well as some of the constraints, are listed follows.

```
order-time, delivery-time, customer-name, customer-mobile, customer-code,
customer-x-location, customer-y-location, pizza-code, pizza-quantity
```

- order-time – The time that the order was placed formatted as "hh:mm:ss" (where *hh* is hours, *mm* is minutes and *ss* is seconds) specified in 24 hour format.
- delivery-time – The time that the delivery was made formatted as "hh:mm:ss" (where *hh* is hours, *mm* is minutes and *ss* is seconds) specified in 24 hour format.
- customer-name – A string that specifies the name of the customer.
- customer-mobile – A string that specifies the customer's mobile number.
- customer-code – A string that specifies if the customer will travel to the restaurant to pick up the pizza ("PUC") or have it delivered to their house either it by a drone ("DNC") or by a driver ("DVC").

- customer-x-location – An integer that specifies the number of blocks east or west of the restaurant that the customer is located
- customer-y-location – An integer that specifies the number of blocks north or south of the restaurant that the customer is located.
- pizza-code – A string specifying the type of pizza that the customer is ordering, either margherita(“PZM”), vegetarian (“PZV”) or meat lovers (“PZL”).
- pizza-quantity – An integer that specifies the number of pizzas ordered.

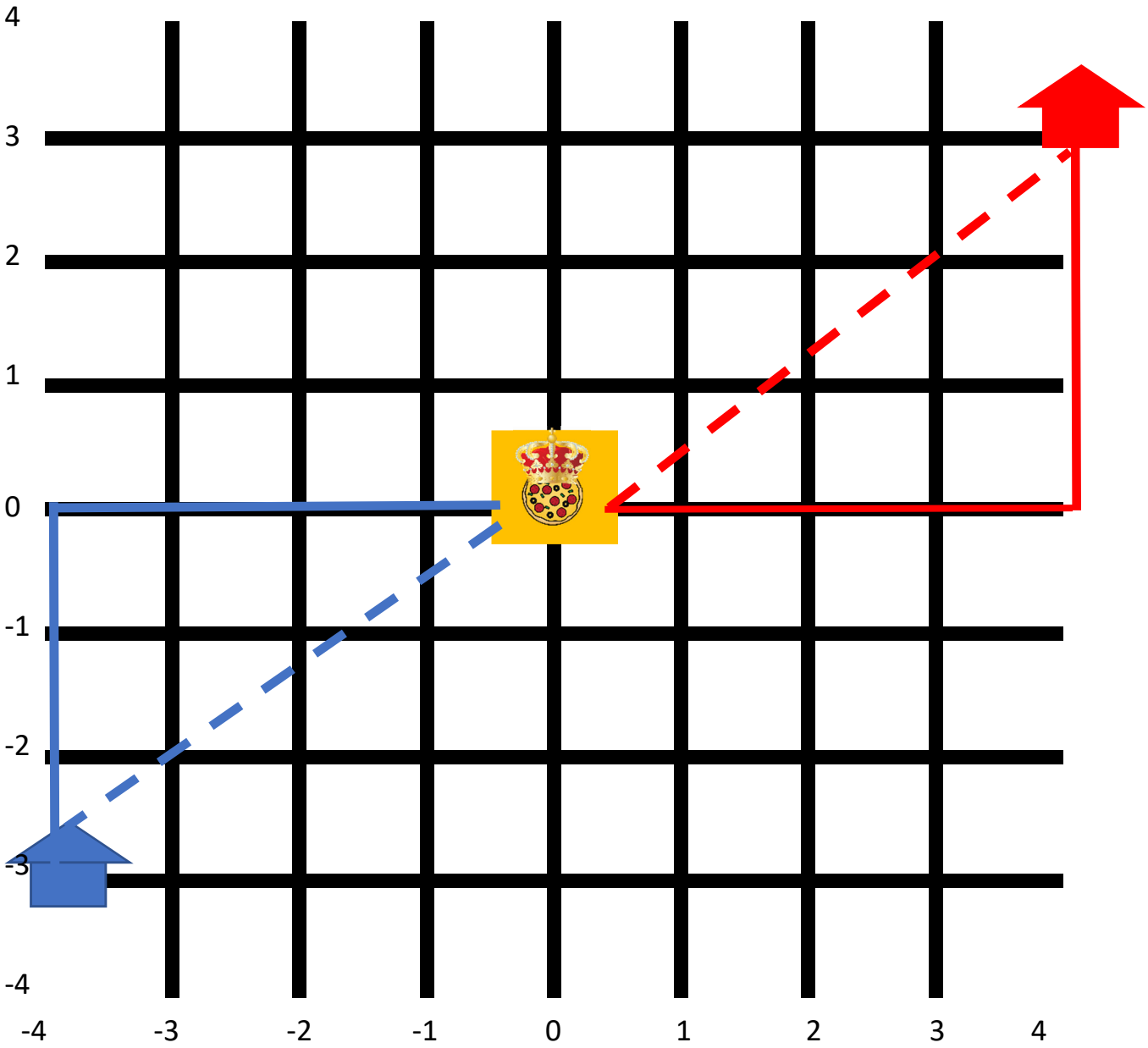

Figure 1: An example of the difference between Manhattan Distance (the solid lines) and Euclidian Distance (the dashed lines). In this example the Pizza Palace is at 0,0 with the customers houses at -4,-3 (blue) and 4,3 (red). Both of these customers have a Manhattan Distance of 7 and a Euclidian Distance of 5 from the Pizza Palace.

Section 5.4 The GUI

You need to design and implement a graphical user interface (GUI). The GUI does not need to be elaborate; however, it must provide the functionality outlined below. Throughout this section, we will use the term "PAG" which means Perform a Gesture. The GUI implementation is left to your own reasonable discretion, so this term allows us to abstract otherwise specific widget actions such as "press a button".

1. **PAG to load a Log file.** The user must be able to load a log file that is saved on disk. The functionality to request opening a file can be done via JButton or similar and actual files can be chosen via a JFileChooser. Once the log file is chosen it should be processed and analysed by the rest of the system.
2. **PAG to display information.** Once the information is processed the user can PAG to display the information contained in the log files. Separate components should be used to display information about the customers and the pizzas. A component such as a JTextField or JTable is suitable to display this information. The information needs to be user friendly, so the codes used to describe pizzas and customers should be translated to into pizza and customer 'types' using descriptive language (Margherita, Meat Lovers, Vegetarian/Pick Up, Driver Delivery, Drone Delivery). The user must not be allowed to perform this gesture unless the log file has successfully been loaded. Specifically, the following information needs to be presented for each order and customer.

   Customer:
   - Customer Name
   - Mobile Number
   - Customer Type
   - X and Y Location
   - Distance from Restaurant

   Order:
   - Pizza Type
   - Quantity
   - Order Price
   - Order Cost
   - Order Profit
3. **PAG to perform and display calculations.** Once the information is processed the user can PAG to calculate the total profit made and total distance travelled for all orders made that day. Again, separate components should be used to display each total. A component such as a JTextField is suitable to display each total. The user must not be allowed to perform this gesture unless the log file has successfully been loaded.
4. **PAG to reset.** Finally, the user should be able to reset and clear all the information from the screen. All components should be also reset to their initial states. The user must not be allowed to perform this gesture unless the log file has successfully been loaded.

The GUI should also report to the user when each functionality has been successfully completed or when any error has occurred.

Section 6: Example Log File

Below are some example log files (all with valid imput)

20170101.txt

```
19:00:00,19:20:00,Casey Jones,0123456789,DVC,5,5,PZV,2
20:00:00,20:25:00,April O'Neal,0987654321,DNC,3,4,PZM,1
21:00:00,21:35:00,Oroku Saki,0111222333,PUC,0,0,PZL,3
```

20170102.txt

```
21:17:00,21:27:00,Emma Brown,0602547760,DVC,-1,0,PZV,5
21:52:00,22:07:00,Lucas Anderson,0755201141,DNC,-4,5,PZL,9
20:43:00,20:53:00,Sophia Singh,0193102468,DNC,1,8,PZV,2
21:05:00,21:34:00,Bella Chen,0265045495,PUC,0,0,PZM,1
19:46:00,20:01:00,Sophia Brown,0101102333,DNC,-2,4,PZV,7
21:45:00,21:56:00,Eli Wang,0858312357,PUC,0,0,PZL,7
20:23:00,20:44:00,Riley Brown,0708426008,DNC,-2,0,PZV,2
21:08:00,21:30:00,Emma Chen,0678585543,DNC,-4,2,PZL,4
22:46:00,22:58:00,Jackson Taylor,0698979160,DVC,-5,-10,PZL,5
20:47:00,21:11:00,Caden Kumar,0862001010,PUC,0,0,PZL,9
```

20170103.txt

```
20:05:00,20:26:00,Aiden Zhang,0161429209,DVC,-3,9,PZV,2
22:35:00,22:55:00,Aria Thompson,0695536923,DVC,6,0,PZV,6
19:39:00,20:06:00,Eli Walker,0106952291,DNC,8,1,PZM,2
19:17:00,19:32:00,Jackson Williams,0738659032,DVC,-7,-6,PZM,9
19:07:00,19:30:00,Aria Jones,0490411652,DVC,6,1,PZV,6
21:00:00,21:11:00,Caden Zhang,0369972735,PUC,0,0,PZM,8
21:54:00,22:14:00,Jackson Smith,0700483384,DVC,-1,-1,PZL,8
22:16:00,22:31:00,Olivia Williams,0771390439,PUC,0,0,PZM,3
19:42:00,19:53:00,Eli Thompson,0971562239,DNC,-2,1,PZM,9
22:14:00,22:31:00,Aria Martin,0366014592,DVC,-1,8,PZV,5
22:16:00,22:44:00,Riley Kumar,0802976919,PUC,0,0,PZL,6
```

<More lines follow>

Section 7: Detailed Tasks

The source tree for the assignment includes the packages shown in Figure 2. The key subdirectories are:

- src: This directory contains all the source code. As a team you will need to complete code for system functionality in the `asgn2Customers`, `asgn2Pizzas`, `asgn2Resurant`, `asgn2GUis`, `asgn2Wizards` packages as well as test code in the `asgn2Tests` package.
- `doc`: This directory contains the API. It has been generated using the Javadoc tool on a complete system.
- `logs`: This directory contains some test logs.
- `libs`: This directory contains the libraries for JUnit and Hamcrest.

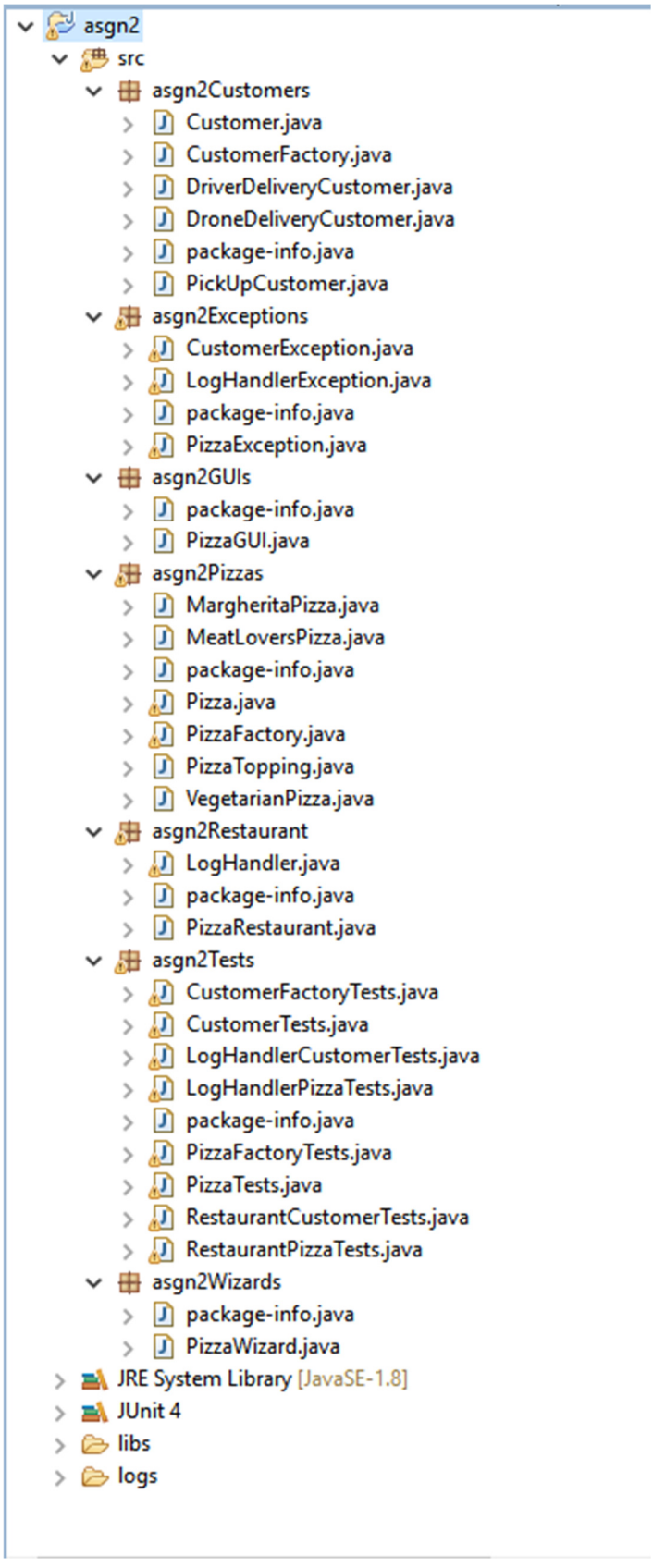

Figure 2: The Assignment Source Tree

Section 7.1 Tasks by Package

Please Note: For all packages and classes carefully check that you have spelt the file's name correctly, otherwise it will not work with our automated marking software.

PACKAGE: `asgn2Pizzas`:

This package contains classes that represent pizzas at the restaurant as well as a class that creates those classes using the factory method pattern and an enumeration that represents pizza toppings. There are three types of pizzas (as described in Section 5.1) represented by three concrete classes which inherit from a single abstract `Pizza` class. The hierarchy of the classes is represented in Figure 4.

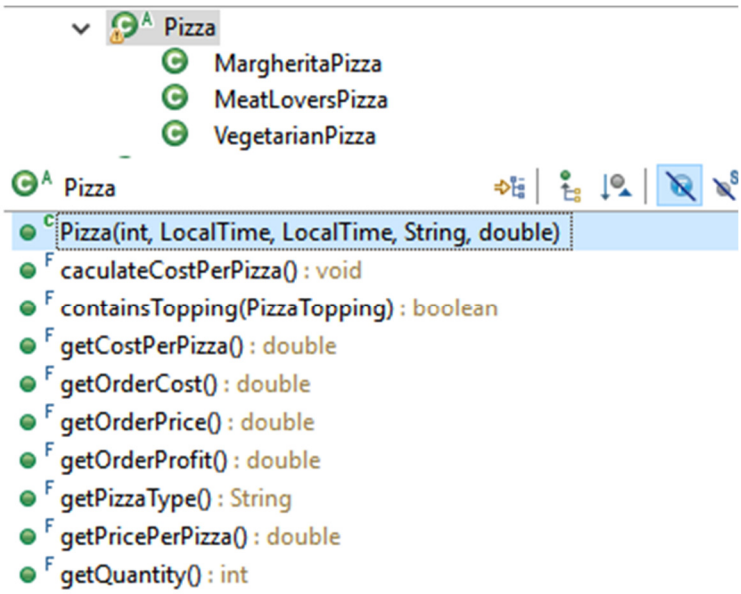

Figure 4: Pizza Hierarchy.

The classes in this package are as follows:

- `Pizza`: An abstract class representing a pizza made at the restaurant. It contains a method to calculate the cost of the pizza derived from its toppings as well as methods to retrieve the quantity of pizza ordered, the cost and price per pizza and per order, the profit made on the order and if the pizza contains a specific topping or not. It also contains a method to retrieve the type of pizza which corresponds to the user-friendly description outlined in Section 5.4. The class also contains a method that can be used to test equivalence between `Pizza` Objects (which is implemented for you and you do not need to test). The class keeps track of when the order was made and when the pizza was delivered (either in the restaurant or to the customer's house).
- `MargheritaPizza`: A concrete class that represents a margherita pizza.
- `MeatLoversPizza`: A concrete class that represents a represents a meat lovers pizza
- `VegetarianPizza`: A concrete class that represents a vegetarian pizza

The specific toppings, costs and prices for each of the concrete `Pizza` subclasses are outlined in Section 5.1.

In addition, the package also contains the following class:

- `PizzaFactory`: A class that contains a single method `getPizza`, which uses the factory method pattern to construct one of the three concrete `Pizza` subclasses. The class choice is specified by the `pizzaCode` parameter which has one of three valid values as outlined in Section 5.3.
- `PizzaTopping`: An enumeration that contains a set of pizza toppings and their associated cost. It also contains a method to retrieve a cost for a specific topping. You do not need to make any changes to this enumeration nor do you need to provide any tests for it.

PACKAGE: `asgn2Customers:`

This package contains classes that represent customers at the restaurant as well as a class that creates those classes using the factory method pattern. There are three types of customers (described in Section 5.2) represented by three concrete classes which inherit from a single abstract `Customer` class. The hierarchy of the classes is represented in Figure 3.

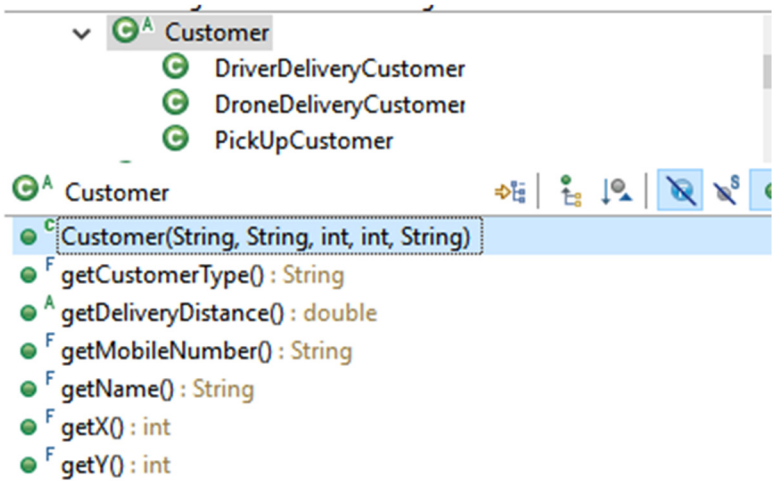

Figure 3: Customer Hierarchy.

The classes in this package are as follows:

- `Customer`: An abstract class representing a customer at the restaurant. It contains methods to retrieve the customer's name, mobile number, x and y locations, all of which is derived from the information in the log file. It also contains methods to retrieve the distance travelled from the restaurant, a method that can be used to test equivalence between `Customer` Objects (which is implemented for you and do not need to test), and retrieve the customer type which corresponds to the user-friendly description outlined in Section 5.4.
- `PickupCustomer`: A concrete class that represents a customer who has chosen to pick up their pizza.
- `DriverDeliveryCustomer`: A concrete class that represents a customer who has chosen to have their pizza delivered by a driver.
- `DroneDeliveryCustomer`: A concrete class that represents a customer who has chosen to have their pizza delivered by a drone.

Each of the `Customer` subclasses has to overwrite the `getDeliveryDistance` method using the formulas outlined in Section 5.2.

In addition, the package also contains the following class:

- `CustomerFactory`: A class that contains a single method `getCustomer`, which uses the factory method pattern to construct one of the three concrete `Customer` subclasses. The class choice is specified by the `customerCode` parameter which has one of three valid values as outlined in Section 5.3.

PACKAGE: `asgn2Restaurant`:

This package contains classes that act as a bridge between the GUI, the log files and the classes in the `asgn2Customers` and the `asgn2Pizzas` packages. It contains the following classes:

- `LogHandler`. A class that contains methods that use the information in the log file to return `Pizza` and `Customer` objects. The class contains methods that return either an individual `Pizza` object (`createPizza`) and `Customer` object (`createCustomer`) as well as methods that return an `ArrayList` of `Pizza` objects (`populatePizzaDataset`) and `ArrayList` of `Customer` objects (`populateCustomerDataset`). Note that the objects in both `ArrayLists` should in the same order that they appear in the log file.
- `PizzaRestaurant`. This class acts as a 'model' of a pizza restaurant and contains an `ArrayList` of Pizza objects and an ArrayList of `Customer` objects. It contains a method that can populate the `ArrayLists` (`processLog`) (via other methods in the system) and several methods to retrieve information about the `ArrayLists` (`geCustomerByIndex`, `getPizzaByIndex`, `getNumPizzaOrders`, `getNumCustomerOrders`, `getTotalDeliveryDistance`, `getTotalProfit`) and reset the ArrayLists (`resetDetails`).

PACKAGE: `asgn2Exceptions`:

This package contains classes to represent different types of exceptions. You do not need to modify or test any of the classes in this package.

- `PizzaExceptions`. A class that represents exceptions related to the construction and handling of `Pizza` objects.
- `CustomerExceptions`. A class that represents exceptions related to the construction and handling of `Customer` objects.
- `LogHandlerExceptions`. A class that represents exceptions related to reading in the log such as incorrect format or exceptions related to input/output.

PACKAGE: `asgn2GUIs`:

This package contains a single class:

- `PizzaGUI`. This class is the graphical user interface for the rest of the system. Currently it is a 'dummy' class which extends `JFrame` and implements `Runnable` and `ActionLister`. It should contain an instance of an `asgn2Restaurant.PizzaRestaurant` object which you can use to interact with the rest of the system. You may choose to implement this class as you like, including changing its class signature – as long as it maintains its core

responsibility of acting as a GUI for the rest of the system. You can also use this class and `asgn2Wizards.PizzaWizard` to test your system as a whole.

PACKAGE: `asgn2Wizards`:

This package contains a single class:

- `PizzaWizard`. This class is the 'entry point' to the rest of the system and provides a `public static void main` method. At the moment, this just calls the `asgn2GUIs.PizzaGUI` class. You can probably leave the class as it is, however, you must make sure that it is the one and only entry point to the rest of the system.

PACKAGE: `asgn2Tests`:

This package contains the JUnit test classes. The classes contained in this package are:

- `PizzaTests` that tests the `asgn2Pizzas.MargheritaPizza`, `asgn2Pizzas.VegetarianPizza`, `asgn2Pizzas.MeatLoversPizza` classes. Note that an instance of `asgn2Pizzas.MeatLoversPizza` should be used to test the functionality of the `asgn2Pizzas.Pizza` abstract class.
- `CustomerTests` that tests the `asgn2Customers.PickUpCustomer`, `asgn2Customers.DriverDeliveryCustomer`, `asgn2Customers.DroneDeliveryCustomer` classes. Note that an instance of `asgn2Customers.DriverDeliveryCustomer` should be used to test the functionality of the `asgn2Customers.Customer` abstract class.
- `PizzaFactoryTests` that tests the `asgn2Pizzas.PizzaFactory` class.
- `CustomerFactoryTests` that tests the `asgn2Customers.CustomerFactory` class.
- `LogHandlerPizzaTests` that tests the methods relating to the creation of `Pizza` objects in the `asgn2Restaurant.LogHander` class.
- `LogHandlerCustomerTests` that tests the methods relating to the creation of `Customer` objects in the `asgn2Restaurant.LogHander` class.
- `RestaurantPizzaTests` that tests the methods relating to the handling of `Pizza` objects in the `asgn2Restaurant.PizzaRestaurant` class as well as `processLog` and `resetDetails`.
- `RestaurantCustomerTests` that tests the methods relating to the handling of `Customer` objects in the `asgn2Restaurant.PizzaRestaurant` class as well as `processLog` and `resetDetails`.

Note the naming convention is to use the plural form (i.e. `MyClassTests.java`) rather than the singular form (i.e. `MyClassTest.java`).

Section 8: Testing and Exceptions

The following principles need to be followed when dealing with testing and exceptions in Assignment 2. Please note that this these principles only refer to the classes in following packages: `asgn2Pizzas`, `asgn2Customers`, `asgn2Restaurant`, `asgn2GUIs`, `asgn2Wizards`. Exceptions in your JUnit tests should be handled as per the standard covered in lectures and tutorials.

Section 8.1 – What types of errors are there and where should errors be handled?

Some parts of the API deliberately lack detail regarding when exceptions should be thrown, for example the constructors of the `asgn2Pizzas.Pizza` and `asgn2Customers.Customer` subclasses say something similar to "*throws exception if supplied parameters are invalid*". However, the exceptions (and where they should be handled) can be broken down into the following cases:

1. Log File Type Errors

Errors associated with reading the log file should be dealt with in `asgn2Restuant.LogFileHandler`. These errors include input/output errors (for example: file not found) and errors associated with the wrong type which will cause parse errors. The required type for each field are listed in Table 4. Most of these types are straightforward, except for the mobile number field that is a String rather than a numeric type, since it contains constraints atypical to numbers (e.g. must start with a '0') and is not used to perform traditional numeric operations (e.g. you don't add together two mobile numbers).

Field	Type
Order Time	LocalTime
Delivery Time	LocalTime
Customer Name	String
Customer Mobile	String
Customer Type	String
Customer X Location	int
Customer Y Location	int
Pizza Type	String
Pizza Quantity	int

Table 4: Required Format.

Note that if there is an error in the log file then neither the customers nor the pizzas dataset should be populated.

2. Constraint Violation Errors

Sections 5.1 and 5.2 list a detailed set of constraints. These constraints should be handled by subclasses of the `asgn2Pizzas.Pizza` and `asgn2Customers.Customer` classes. Most of these are logic based (e.g. a delivery cannot be made to certain locations or a pizza cannot take longer to cook than a specified duration). However, some are based on format or other things (e.g. a mobile number needs to be in the correct format).

3. Incorrect Pizza and Customer Codes

There are only three valid pizza and customer codes. Any invalid codes should be handled by the `asgn2Pizzas.PizzaFactory` or `asgn2Customers.CustomerFactory` classes.

4. System Errors

Broader system errors (e.g. those involving accessing multiple instances of the `asgn2Pizzas.Pizza` or `asgn2Customers.Customer` classes) should be handled by the `asgn2Restuant.PizzaRestaurant` class.

5. Other Errors Not Specified

Section 5 has covered specific errors for this system, however, there are some general errors (for example: accessing an invalid index) which have not been discussed. These errors should also be accounted for in your solution, particularly, if we have discussed these errors during the lecture.

If there is any doubt about which errors should be included, then you should ask the teaching team (in particular the unit co-ordinator). Since part of the challenge for this assignment is to identify the errors, you should ask the teaching team member privately (either face-to-face or by email) rather than on a public forum (e.g. Facebook).

Section 8.2 – Propagating Exceptions Without Violating the API

Exceptions should be propagated throughout the system. This means that if `MethodA()` calls `MethodB()` and `MethodB()` calls `MethodC()`, then exceptions thrown by `MethodC()` should be thrown to `MethodB()` and in turn to `MethodA()` (note: this is <u>different</u> to Assignment 1). This is because exceptions in the backend classes need to be passed to the `asgn2GUIs.PizzaGUI` class and displayed to the user. However, you still need to conform with the API regarding which methods can throw which exceptions (so you can't change a method signature to thrown an additional exception) and only the exceptions in the `asgn2Exceptions` package should be thrown (`PizzaExceptions`, `CustomerExceptions`, `LogHandlerExceptions`). This means that you have to 'translate' other types of exceptions into one of the `asgn2Exceptions` types and requires you to follow the following steps (In general, it is better to use case 1 rather than case 2).

`Case 1:` Program Defensively

In this case, you want to throw an exception before another exception occurs. You should check for the condition that would throw the exception and throw one of the custom `asgn2Exceptions` instead.

For example:

```
MethodC() throws MyCustomException(){
      if(condition) throw new MyCustomException("Message");
}
```

`Case 2:` Program Reactively

Sometimes you cannot perform a check before an exception is thrown. In this case, you want to catch that original exception within the method and throw one of the `asgn2Exceptions` instead. This requires the use of a try/catch block.

For example

```
MethodC() throws MyCustomException(){
      try {
            /* do something that raises an exception
             other than MyCustomException */
      } catch (Exception e){
            Throw new MyCustomException("Message");
      }
}
```

If you are able to catch more specific Exceptions, or multiple specific Exceptions within the same try/catch block, then you should do so.

Section 8.3 – Rethrowing the Same Exception

There are times when an exception occurs and the API says that that exception should be thrown, but you want to do something else *before* you throw that exception (for example, setting a variable). In this case the exception should be thrown within the catch block. For example:

```
MethodC() throws MyCustomException(){
      try {
            // do something that raises a custom exception
      } catch (Exception e){
            // do something else – like setting a variable
            Throw e;
      }
}
```

Section 9: Test Driven Development and Teams

Section 9.1 – Breakdown of Work

One of the keys aims of this assignment is to give you experience working in a team based test driven development scenario. Therefore, you should be designing and implementing the test cases for your partner and vice versa. As such, the breakdown of the work should be as listed in Table 5.

For most of the classes you will be able to work individually (individual model classes and individual test classes). However, sometimes both team members will need to work on the same class and sometimes on the same method (shared model classes).

	Person A	**Person B**
Individual Model Classes	In package: asgn2Pizzas • Pizza • MargheritaPizza • MeatLoversPizza • VegetarianPizza • PizzaFactory	In package asgn2Customers • Customer • PickUpCustomer • DriverDeliveryCustomer • DroneDeliveryCustomer • CustomerFactory
	In package: asgn2Restaurant • LogHandler.populatePizzaDataset() • LogHandler.createPizza() • PizzaRestaurant.getPizzaByIndex() • PizzaRestaurant.getNumPizzaOrders() • PizzaRestaurant.getTotalProfit()	In package: asgn2Restaurant • LogHandler.populateCustomerDataset() • LogHandler.createCustomer() • PizzaRestaurant.getCustomerByIndex() • PizzaRestaurant.getNumCustomerOrders() • PizzaRestuarant.getTotalDeliveryDistance()
Individual Test Classes	In package: asgn2Tests • CustomerFactoryTests • CustomerTests • LogHandlerCustomerTests • RestaurantCustomerTests	In package: asgn2Tests • LogHandlerPizzaTests • PizzaFactoryTests • PizzaTests • RestuarantPizzaTests
Shared Model Classes	asgn2GUIs.PizzaGUIs asgn2Restaurant.PizzaRestaurant.PizzaRestaurant asgn2Resaturant.PizzaRestaurant.processLog asgn2Restaurant.PizzaRestaurant.resetDetails asgn2Wizard.PizzaWizard	

Table 5: Breakdown of work between team mates

Section 9.2 – Forming Teams

Over the years we have run this assignment, there have always been people who have either requested that they be allowed to do the assignment alone (the right way of going about it) or simply handed in an assignment that is solely their own work and hoped that we'd be ok with it (the wrong way of going about it). Please ask if there is a good reason why you cannot make work in a pair but generally the answer will be no unless there are specific and very significant obstacles in the way.

In the past we have had a number of students who work full time contact me to express their concerns about working with other people, with some worried about having opportunities to pair up with other

students. I had have successful collaborations which involve coding with people on the other side of the world, and this is increasingly common. The key of course is to establish the pair. Once you have had a couple of face to face meetings, it will usually work. Mostly working full time is not a sufficient barrier – I have had people working for mining companies in remote locations with lousy internet connectivity, and I have allowed them to work alone. Those in the city should normally be able to connect sufficiently, with occasional face to face meetings.

Section 10: Additional APIs

We will cover the vast majority of work required to complete the assignment in class. However, there are some additional APIs which you will need to read yourself. This is not an uncommon practice in Java software development since the JDK is large and so programmers regularly use unfamiliar APIs. Note you should have already covered the desired functionality in your previous programming units, so you should know ***what*** needs to be done even if you don't know explicitly ***how*** it should be done in Java. Here, we have listed the some of the APIs and provided links to the Official Java Tutorial. There is also plenty of material online or in the online Java books in the library (http://libcat.library.qut.edu.au/search/~?searchtype=X&searcharg=java+programming&searchscope=8&SORT=D&submit=Search)

- **Reading in text files:** Typically, you want to read in a text file using a `BufferedReader` (https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html). Examples of how to read files using a `BufferedReader` can be found in the Java tutorials (https://docs.oracle.com/javase/tutorial/essential/io/buffers.html and https://docs.oracle.com/javase/tutorial/essential/io/file.html). There is also an example of reading in a file in the Assignment 1 `SoccerWizard` class.
- **Parsing Strings.** You will need to parse Strings to other formats using methods found in LocalTime API (https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) and Integer API (https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html).
- **Handling Time.** Times can be handled using the LocalTime API (https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) available in Java 8.
- **String Processing.** The String API (https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) contains methods to process String and convert them to other objects (such as Arrays).
- **Mathematical Operations.** The Math API (https://docs.oracle.com/javase/8/docs/api/java/lang/Math.htmll) contains methods to perform mathematical operations.

Section 11: Class Interaction

Figure 5 presents a UML Sequence Diagram for the user action of loading a log file (acted after the user performs the appropriate gesture). This is a good example of how the classes should interact with each other, although it does not list all of the classes involved in the action nor does it name the methods called.

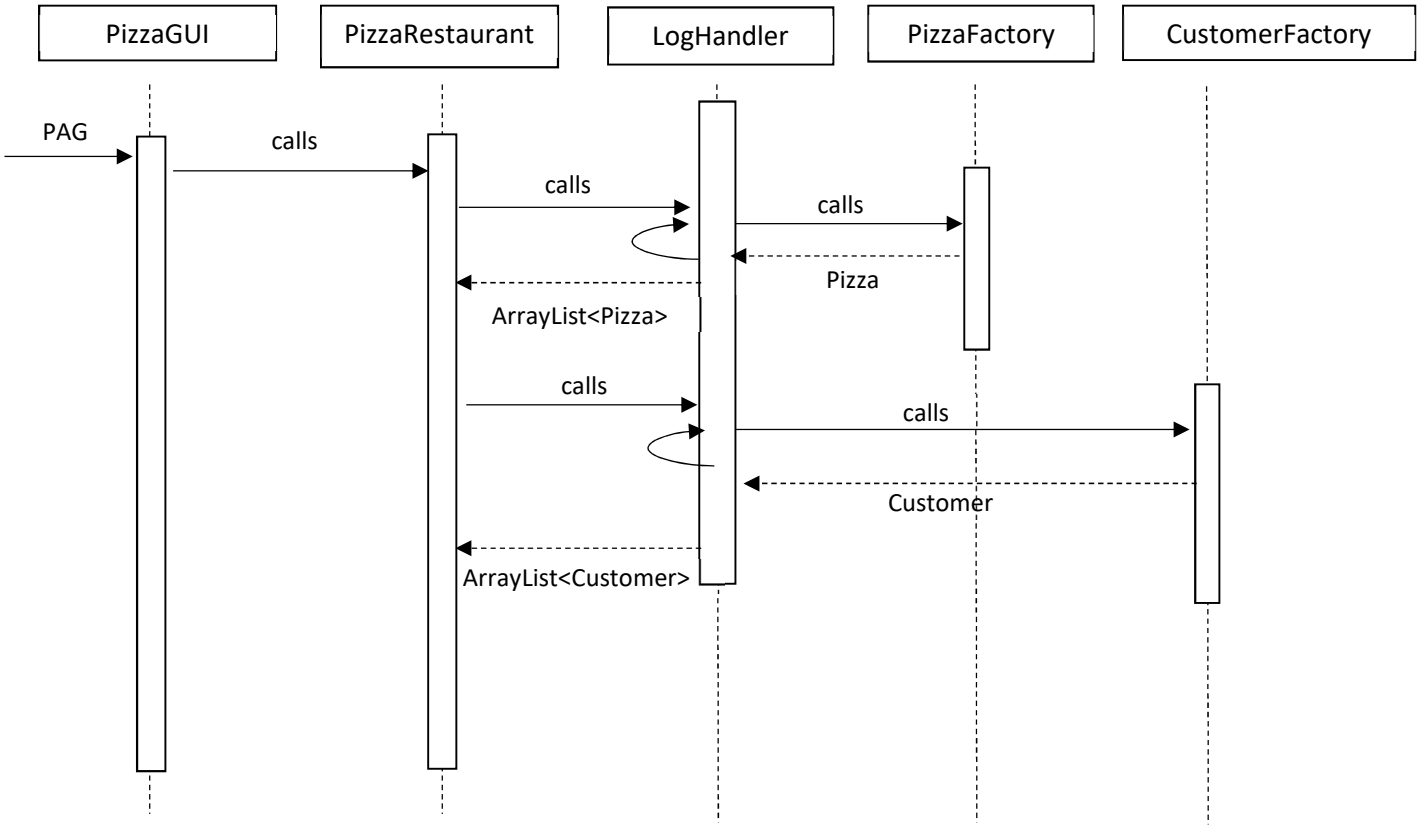

Figure 5: UML Sequence Diagram for the action of loading a log file.

Most of the other interactions will be handled by `asgnGUIs.PizzaGUI` and `asgn2Restaurant.PizzaRestaurant`.

Section 12: Academic Integrity

This is a pair assignment. Please read and follow the guidelines as specified in QUT's MOPP regarding what constitutes plagiarism (http://www.mopp.qut.edu.au/C/C_05_03.jsp) and the penalties (http://www.mopp.qut.edu.au/E/E_08_01.jsp#E_08_01.08.mdoc). Programs submitted for this assignment will be analysed by the MoSS (Measure of Software Similarity) plagiarism detection system (`http:// theory.stanford.edu/~aiken/moss/`)

Section 13: Assessment Criteria

Overall the Assignment with be worth 35 marks split across the following six categories: code quality, model code development, unit testing, GUI design, GUI functionality and documentation. Overall, the Assignment will be marked out of 35.

Section 13.1 – A note on Assessment

This is a pair project so that you can experience team based test driven development. However, we will be calculating separate marks for each team member (Person A and Person B). By default, we will be *averaging* these two marks and assigning the average to *both* team members. However, there are cases where a team member may request an *individual* mark rather than the average. For this to occur at least one team member must include the following **signed** statement in the document described in Section 13.7.

"*I (student name and number) would prefer for my partner and I to receive separate individual marks for the assignment rather than the same average mark*."

Note that if **one** team member includes this paragraph in the document then **both** team members will receive individual marks.

If both team members would prefer to receive a single average mark then there is no need to include any statement.

Section 13.2 – Code quality (5 marks)

Producing well-presented, understandable code, and being able to communicate your achievements to other programmers are essential skills for a professional programmer. To assess this, your tutor will briefly inspect and assess your code against the following check list.

- ☐ **Layout.** All program code and unit tests are neatly laid out with appropriate use of whitespace and avoidance of overly wide lines. All comments and other free-form text, such as exception messages, are clear, well-written and free of spelling or grammatical errors.
- ☐ **Readability.** Meaningful identifiers are used for all methods, parameters, fields, etc, *including loop indices*. The meaning of each identifier is made clear by avoiding cryptic abbreviations and other obfuscation.
- ☐ **Simplicity.** Data structures, algorithms and control flow constructs are all as simple as possible.
- ☐ **Maintainability.** The program code and unit tests are all made easy to maintain by use of named constants instead of 'magic numbers'. Methods are small, each serving a single purpose. In particular, each unit test serves one purpose only.
- ☐ **Documentation.** The program and unit tests are appropriately commented, in a way that complements the code, by explaining *what* the code does or *why*, but does not merely repeat *how* it works (which should be self-evident from the code itself). Commenting is sparse, serving to clarify unclear code segments only.

Section 13.3 – Model Code Development (7.5 marks)

Producing code that exactly matches your client's technical specifications is another essential skill for a professional programmer. In this case your electronically-submitted program code will be tested automatically, so you must adhere precisely to the specifications in these instructions. Submissions that do not compile correctly with our test software, or which are submitted in the wrong file format will automatically receive zero marks for model code development.

To assess the model code development for Person A, tests will be executed against the following classes/methods:

- `asgn2Pizzas.Pizza`
- `asgn2Pizzas.MargheritaPizza`
- `asgn2Pizzas.MeatLoversPizza`
- `asgn2Pizzas.VegetarianPizza`
- `asgn2Pizzas.PizzaFactory`
- `asgn2Restuarant.LogHandler.populatePizzaDataset()`
- `asgn2Restaurant.LogHandler.createPizza()`
- `asgn2Restaurant.PizzaRestaurant.getPizzaByIndex()`
- `asgn2Restaurant.PizzaRestaurant.getNumPizzaOrders()`
- `asgn2Restaurant.PizzaRestaurant.getTotalProfit()`
- `asgn2Restaurant.PizzaRestaurant.processLog()`
- `asgn2Restaurant.PizzaRestaurant.resetDetails()`

To assess the model code development for Person B, tests will be executed against the following classes/methods:

- `asgn2Customers.Customer`
- `asgn2Customers.PickUpCustomer`
- `asgn2Customers.DriverDeliveryCustomer`
- `asgn2Customers.DroneDeliveryCustomer`
- `asgn2Customers.CustomerFactory`
- `asgn2Restaurant.LogHandler.populateCustomerDataset()`
- `asgn2Restaurant.LogHandler.createCustomer()`
- `asgn2Restaurant.PizzaRestaurant.getCustomerByIndex()`
- `asgn2Restaurant.PizzaRestaurant.getNumCustomerOrders()`
- `asgn2Restaurant.PizzaRestaurant.getTotalDeliveryDistance()`
- `asgn2Restaurant.PizzaRestaurant.processLog()`
- `asgn2Restaurant.PizzaRestaurant.resetDetails()`

Section 13.4 – Unit Test Classes (7.5 marks)

Your test classes will be executed on a series of defective solutions to ensure that they adequately detect programming errors. Your unit tests are deemed to have detected a programming error in a defective program if more of the tests fail when applied to the defective program than when the tests are applied to our 'ideal' solution.

To assess the unit tests for Person A, defective solutions will be executed against the following classes:

- `asgn2.CustomerFactoryTests`
- `asgn2.CustomerTests`
- `asgn2.LogHandlerCustomerTests`
- `asgn2.RestaurantCustomerTests`

To assess the unit tests for Person B, defective solutions will be executed against the following classes:

- `asgn2.PizzaFactoryTests`
- `asgn2.PizzaTests`
- `asgn2.LogHandlerPizzaTests`
- `asgn2.RestaurantPizzaTests`

Section 13.5 – GUI Design (5 marks)

The GUI design does need not need to be world class, but some basic principles be followed. In particular, the GUI should:

- Contain widgets that enable the user to perform all the functionality listed in Section 5.4
- Group widgets appropriately using a sensible layout
- Use appropriate names for each widget.
- Use appropriate headings to describe what each customer/pizza detail represents.
- Print out floating point numbers using two decimal places and print integers with zero decimal places.
- Use appropriate units to display numerical information.
- Appropriately report to the user when each functionality is complete.
- Reporting to the user any errors that have occurred.
- Allow or not allow users to PAG under some circumstances.

The same GUI Design mark will be awarded to both team members.

Section 13.6 – GUI Functionality (5 marks)

The following will be used to assess the GUI Functionality for Person A.

- Successfully loading a log file.
- Successfully displaying details related to the customers as outlined in Section 5.4.
- Successfully calculating and displaying the total distance travelled as outlined in Section 5.4.
- Successfully resetting the system and clearing all information as outlined in Section 5.4.
- Informing the user when a log file contains invalid customer details.

The following will be used to assess the GUI Functionality for Person B.

- Successfully loading a log file.
- Successfully displaying details related to the order as outlined in Section 5.4.
- Successfully calculating and displaying the total profit as outlined in Section 5.4.
- Successfully resetting the system and clearing all information as outlined in Section 5.4.
- Informing the user when a log file contains invalid pizza details.

Section 13.7 – Documentation (5 marks)

You need to provide a PDF document that contains

- Both student names and numbers
- A statement indicating if one team member would like to receive an individual mark rather than an average mark. (optional)
- Evidence of testing using screen shots for each of the GUI functionality outlined in Section 13.6.

In addition, you also need to provide:

- A set of HTML files produced by Javadoc providing information about who authored each class as well as expanding the method description to include which exceptions have been thrown and why.
- A repository log file that contains evidence of the contributions that each team member has made. Instructions on how to generate this file are listed in Section 14.1.

The PDF file, document of HTML files and repository log file should be placed in the top directory of the project (copy via Windows) so that it will correctly be placed in the zip file created using the steps in Section 14.2.

Section 14: Submitting Your Assignment

The procedure to submit Assignment 2 is similar to Assignment 1 with the following changes:

1. You need to produce a document that contains the details outlined in Section 13.7.
2. You need to produce a record of your version control contributions.
3. You need to produce a set of suitable Javadoc.
4. You need to save your zip file using the name "nAAAAAAA_nBBBBBBB.zip" where nAAAAAAA is the student number of Person A and nBBBBBBB is the student number of person B.

You must submit your completed assignment as a complete source tree. In particular, you ***must*** respect the specific class name and package structures. You must submit your solution before midnight on the due date to avoid incurring a very firm late penalty and a mark of zero. You should be aware of QUT's policy on late assignments (https://www.student.qut.edu.au/studying/assessment/late-assignments-and-extensions). You should take into account the fact that the network might be slow or temporarily unavailable when you try to submit. **Network problems near the deadline will not be accepted as an excuse for late assignments.** To be sure of receiving a non-zero mark, submit your solution well before the deadline. Completing programming tasks within a given deadline is one of the requirements of a professional programmer.

Please note that a number of automated tools will be used to assist with the marking of this assignment. To this end, please follow these instructions carefully so that your files are in the appropriate folders to be detected by our tools. **In particular, check that you have spelled the names of your Java packages, classes and methods correctly and that they are capitalised exactly as specified**.

Section 14.1 Creating your Repository Log File

It is mandatory that you supply us with evidence that you have undertaken some sort of consistent version management. Most people will use git, but whatever you use, we require that you submit a log file – text *only* - to be called `repo.log`, and included with the others as shown in Figures 6 and 7. We need only a snapshot history. We don't want to see everything at once. I recommend the following commands for git, to be executed at the bash command line prompt. The first command is to set an alias to allow you to generate a quick summary again and again. The remaining commands generate the file for submission.

First we set up a global alias (at the git bash command line):

```
$ git  config --global alias.pretty "log --pretty=format:'%h : %s' --graph"
```

Then we use it to generate the summary log:

```
$ git pretty > repo.log
```

Before finally appending a complete log which we can look it as needed:

```
$ git log >> repo.log
```

Note that we will usually just look at the summary. The full log with all the dates is there for checking, and if we don't like that, we will ask to inspect your repo. The log file should be placed in the top directory of the project (copy via Windows) so that it will correctly be placed in the zip created using the steps in Section 14.2.

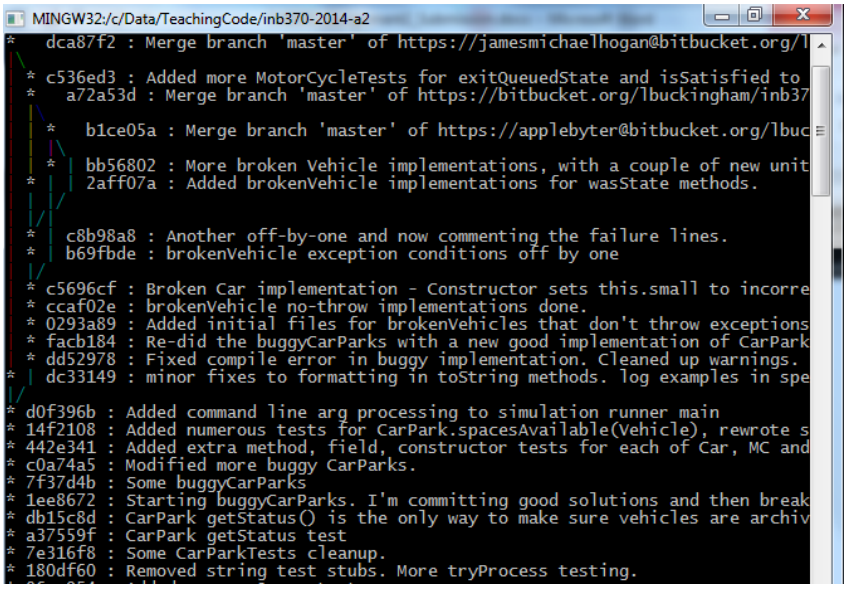

Figure 6 Example of a Git log repository.

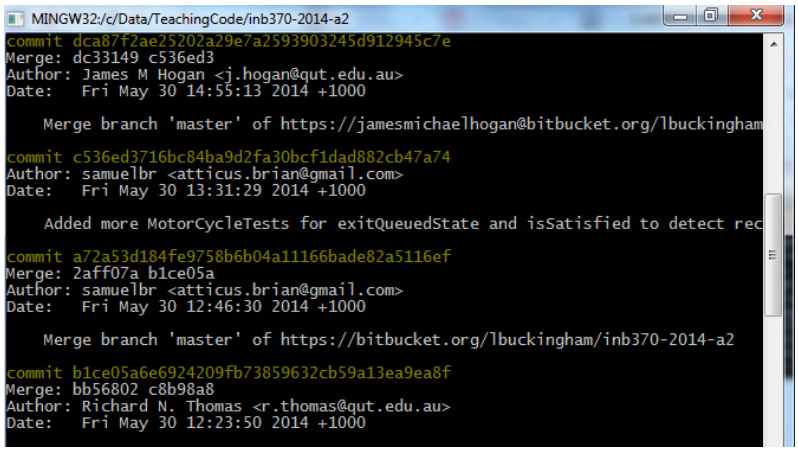

Figure 7 Further example of a Git log repository.

Section 14.2 Exporting your Solution from Eclipse

Note that the following screen shots are from the 64-Bit Windows versions of Eclipse Nano. Your screen may be different. However, we recommend that you follow the instructions using the computers in the practical rooms.

Step 1. Your entire submission will be contained in a single zip archive exported from within Eclipse. Make sure that your solution to the assignment is in a project named `asgn2`, containing the Java packages outlined in Figure 8.

Step 2. Once you are ready to submit, open the Eclipse workspace containing your assignment. Right-click on the project folder as shown in Figure 8 and select the `Export…` entry in the pop-up menu. **It is important that you select the *project* folder, as shown in the figure, otherwise not all of your files will be included in the zip archive.**

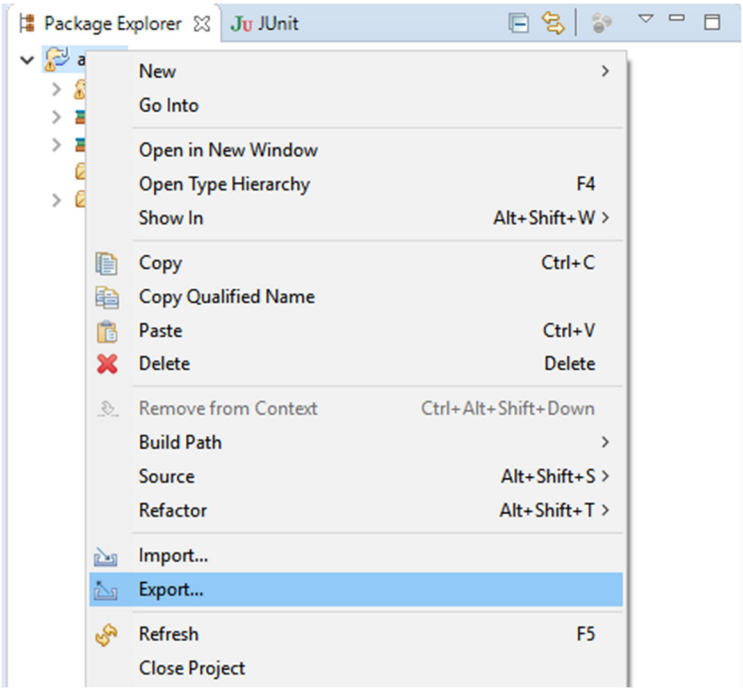

Figure 8 Right-click on the project folder as shown and choose the Export… menu entry.

Step 3. Once you have done this an Export Wizard dialogue box like that shown in Figure 9 will open. Select the `Archive File` option under the `General` heading.

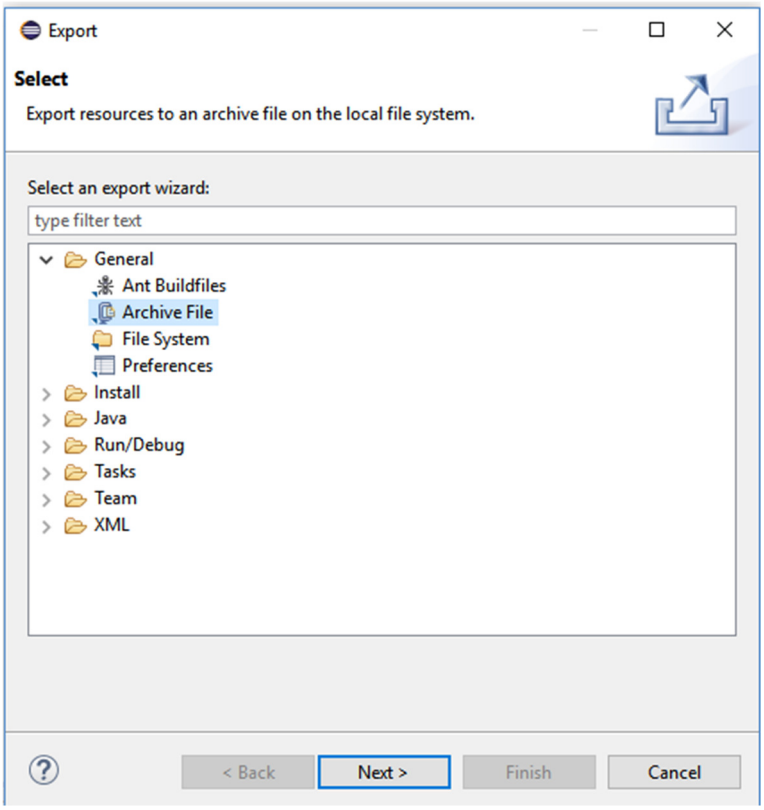

Figure 9 The first step of the Export Wizard, select the Archive File option and click Next.

Step 4. The next step of the wizard is shown in Figure 10. In the top left list of projects ensure that there is a check mark next to the name of the project containing your assignment. Do not include any other project (your assignment submission must be contained in a single project). Press the `Browse…` button next to the '`To archive file:`' text box. Browse to an appropriate folder in which to save the archive and enter the file name as "`nAAAAAAAA_nBBBBBBB.zip`" (without the quotes) where nAAAAAAAA is Person A's student number and nBBBBBBBB is Person B's student number .

Make sure in the `Options` list that the radio button next to '`Save in zip format`' is selected as are the '`Create directory structure for files`' and '`Compress the contents of the file`' options. Once this is done, press `Finish`.

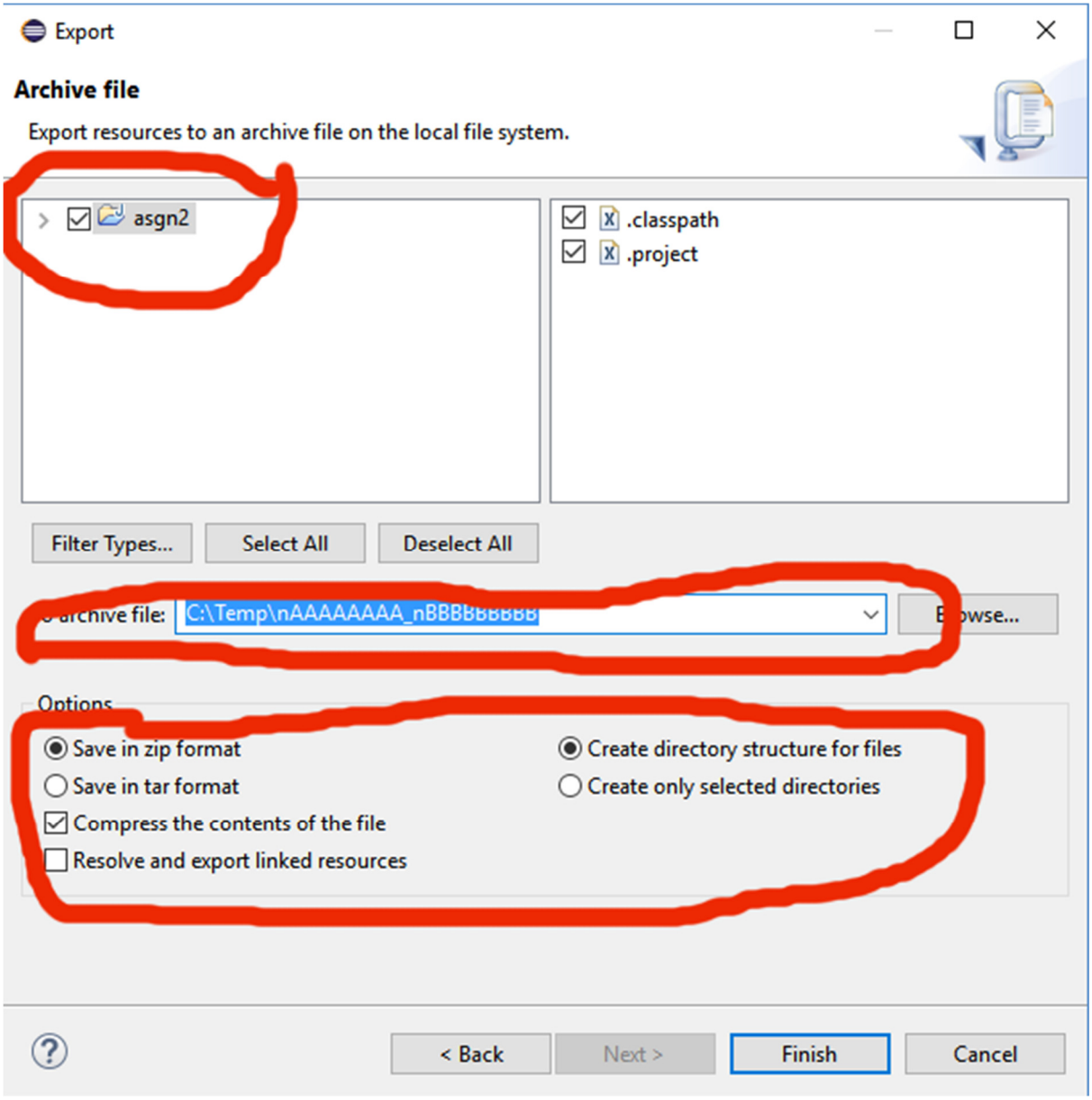

Figure 10 The final step of the Export Wizard. Ensure that the project containing your assignment has a check next to it and that the Options area is configured as shown. Press Browse… to select the save location; your zip file must be named `nAAAAAAAA_nBBBBBBB`.zip. Once you are ready select Finish to export your work.

Step 5. You can check that the zip archive is correctly structured by unzipping it and confirming that this produces (at least) a folder named `nAAAAAAAA_nBBBBBBB`, containing a folder named `src`, `doc` and your other required documentation.

Note that if you make any changes to your assignment *after* performing the above steps, you need to redo them to create a new zip archive before submitting it through the Online Assessment System.

Section 14.3 Submitting your Solution

You must submit your assignment on Blackboard. We will send out further details how to do this closer to submission.

<< END OF ASSIGNMENT SPECIFICATION >>