Assignment 3

TAs: Majeed and Jumana

Publication Date: 5/12/2013
Unit Tests Deadline: 12/12/2013 23:59
Assignment Submission Deadline: 26/12/2013 23:59

1 General Description

In this assignment you will simulate a restaurant which employes multiple chefs. The restaurant receives a list of orders, and its task is to simulate cooking these orders, as well as delivering them to their customers once they are finished. Each order cooked and successfully delivered will provide the company an income. Each order requires different tools, and ingredients. In order to successfully cook an order, your task is to make sure that the chefs has the needed kitchen tools as well as the ingredients needed.

You will manage the simulation process starting from receiving new orders, purchasing required kitchen tools and ingredients, until the order is delivered. Each chef will cook an order, and once cooked, the order needs to be delivered to an appropriate delivery person. Successfully delivering an order in reasonable time will net the restaurant an income.

At the end of the process, you will print out statistics which will show how well your restaurant has done. Money gained as well as how much time each order took to complete and to deliver.

Please note that we explain in detail the vast majority of the system; however, we kept some sections vague to allow you more freedom in deciding how to implement them. If there is no precise explanation of a section, it means you are free to implement it as you see fit, as long as it provides the desired result.

Note: "Found in:" denotes that it may be found in noted class, but not limited to it. You may add the same item anywhere as you see fit.

2 Passives

In this section, we will detail the different parts of the system, the contents of each part, and their tasks. Fields mentioned for any object are mandatory, however, it is allowed to add more, as you see fit.

2.1 Dish

Found in: Menu

This object will hold information of a single dish. This object can be stored in the restaurant's menu and can be used in different dish and dish order-related operations.

A Dish object will hold the following information: (1) Dish Name (2) Dish Cook Time (3) Collection of Dish Ingredients (4) Collection of Required Kitchen Tools (5) Difficulty Rating (6) Reward.

Each dish has different difficulty rating which will affect the distribution of the different OrderOfDish to the different chefs. Detailed explanation below. A dish also requires a collection of exhaustable ingredients and different kitchen tools, in order to successfully simulate the cooking procedure.

Dish Difficulty Rating is an integer value.

2.2 OrderOfDish

Found in: Order

This object will hold information of a dish order.

This object will hold: (1) Dish (See 2.1) (2) Quantity

This object will be sent to RunnableCookOneDish objects which will simulate the dish cooking, as explained below.

2.3 KitchenTool

This object will hold information of a single kitchen tool type. Each kitchen tool has: (1) Name (2) Quantity. This item is not consumed in the process, but returned once its use is done.

2.4 Ingredient

This object will hold information of a single ingredient type. Each ingredient contains the following fields: (1) Name (2) Quantity. This item is consumed in the process. Ingredients used in cooking are not returned once the cooking is complete.

2.5 Warehouse

Found in: Management

This object contains: (1) a collection of available kitchen tools (2) a collection of available ingredients. This warehouse is the shared storage component where the different chefs acquire their tools and needed ingredients from.

You may assume the warehouse contains enough ingredients and kitchen tools in order to successfully complete the simulation process.

2.6 Order

This object will hold the information of an order. It will hold (1) Order ID (2) Difficulty Rating (3) Order Status ((i) INCOMPLETE(ii) INPROGRESS (iii) COMPLETE (iv) DELIVERED) Magic Numbers and Solution (4) Collecion of OrderOfDish (5) Customer Address.

Difficulty rating is calculated as a combination of all the difficulties of all the dish types found in the order. Each dish *type* increases the difficulty of the order. Example: Your order consists of 3 Pizzas and 2 Hamburgers, the order difficulty rating equals the difficulty rating of pizza plus the difficulty rating of hamburger.

This object will be sent to CallableCookWholeOrder which will simulate the cooking procedure of the whole order.

Found in: Management

2.7 Management

This object contains the following fields: (1) Collection of Orders (2) Collection of Chefs (3) Collection of Delivery Persons (4) Warehouse.

This department handles two different tasks:

1. Handles adding new orders to the collection of *orders to cook*.
2. Handles taking new orders from *orders to cook* and sending them to the appropriate chef in order to cook them. Once all the chefs are busy, it needs to wait until notified by a chef, that they are free to receive new orders to cook.

Be sure not to affect parallelism. If a task is waiting, that doesn't mean the whole application needs to wait, it means only that task needs to wait, the rest of the application can continue working normally.

2.8 Statistics

Found in: Management

This object contains the following fields:

1. Money Gained: Money gained from delivering orders.
2. Delivered Orders: Collection of delivered orders, including their information.
3. Collection of Ingredients consumed and their quantity.

Updating this object may be done from anywhere in the program, however it is advised to do so in minimal amount of places.

3 Actives

3.1 RunnableCookOneDish

Found in: CallableCookWholeOrder

Note: Must be Runnable.

This object will be our first active object. It contains the following fields: (1) Dish (2) Warehouse (3) Chef

The purpose of the runnable is to cook a single dish of a given dish order. Each dish instance has a quantity, and we're going to simulate cooking the whole quantity in parallel. Which means the number of runnables for each dish equals its quantity.

The life cycle:

- Acquire all the kitchen tools needed for the dish.
- Acquire all the ingredients of the dish.
- Sleep during the cooking time of the dish (multiplied by the chef's efficiency factor, and rounded to nearest long)
- Return acquired kitchen tools.

Of course, update all applicable variables with new data, such as the status of the Order.

3.2 CallableCookWholeOrder

Found in: RunnableChef

Note: Must be Callable.

This object will be our second active object. It contains: (1) Order object (2) Warehouse.

This callable object will cook the whole quantity of each dish for all the dishes in **parallel**, and only once all the dishes are cooked, it may exit and return the cooked Order. There is no limit on the number of threads to create. You will create one thread which wraps RunnableCookOneDish per dish. Do not forget to update Order status progress. Example: Your order contains two dishes: Pizza and Pasta, of quantity 3 of the first, and 4 of the latter. In total, you will create and start **7** different threads in parallel.

Calculating real cook time: You poll the current time before starting to execute the order, and poll current time once an order is completely cooked. Difference between the two time stamps is the real cook time.

Note: You may add as many fields as needed.

3.3 RunnableChef

Found in: Management

Note: Must be Runnable.

This object is our third active object. Fields: (1) Chef Name (2) Chef Efficiency Rating (3) Endurance Rating (4) Current Pressure (5) Collection of Futures for Orders in Progress (6) Thread pool.

This runnable receives new cook orders from the Managment, and depending on dish *difficulty*, *endurance rating* and *current pressure*, decides whether to accept or deny the request for cooking the order. The formula is as follows: if (Order *Difficulty* $\leq$ *Endurance Rating* - *Current Pressure*) then the request is accepted, and the cooking simulation procedure automatically begins. Of course, the value of *current pressure* needs to be increased by the value of order *difficulty*.

At any time, the chef may receive a shut down request. If received, the chef will finish cooking the orders he may be working on while **denying** new requests, once all currently being cooked orders are complete, the thread exits.

Once an Order is finished, the current pressure value needs to be decreased by the value of the completed Order difficulty. Then the chef sends the finished Order to the delivery department queue.

Chef Efficiency Rating can be one of three: (i) Good = 0.9 (ii) Neutral = 1.0 (iii) Bad = 1.1

Both Chef Endurance Rating and Current Pressure are of an integer type.

3.4 RunnableDeliveryPerson

Found in: Management

Note: Must be Runnable.

This object will hold the current fields: (1) Delivery Person Name (2) Restaurant Address (3) Speed of Delivery Person (4) Collection of Orders that have been delivered

It will simulate retrieving a new order to deliver as follows:

- Once an order is available for delivery, it is retrieved.
  - Calculate distance between the restaurant and customer address, divide by speed. The distance is calculated as follows: Your restaurant has a pair of coordinates (x_1,y_1) and each order has a customer address which also is a pair of coordinates (x_2,y_2). In order to calculate the distance between the two points, use Euclidean distance; then round it to the nearest integer. This number will be in milliseconds. This is the expected delivery time.
  - Actual delivery time: the time that was actually spent using Date() polling right when they receive the delivery item, and after delivery is complete:
    - Acquire item to deliver.
    - Poll current time.
    - Deliver item.
    - Poll current time.
      - Time that has passed between the first time polling and the second time polling, is the time it took for the order to get delivered.
- Sleep distance time - to destination.
- Mark order as delivered. If the (actual cook time + actual delivery time) $>$ 115% * (expected cook time + expected delivery time), then you receive 50% of the reward, otherwise, you receive 100% of the reward.
- Sleep distance time - to home.
- Repeat cycle. The delivery person will shutdown only when a shutdown request is received.

4 Program Cycle

First, you need to parse 3 files and create objects to store the information parsed from these files. These files include information regarding:

1. Management object which will include the following:
   (a) Statistics object.
   (b) Orders you will need to cook and deliver [File: OrdersList]
   (c) The Menu. Which contains the dishes the restaurant offers. [File:Menu]
   (d) Collection of runnable Chefs. [File: InitialData]
   (e) Collection of runnable Delivery Persons. [File: InitialData]
   (f) The equipment/ingredients (Warehouse) that the restaurant has [File: InitialData]

Once the parsing process is done. You will launch the simulation process, and call the Management.

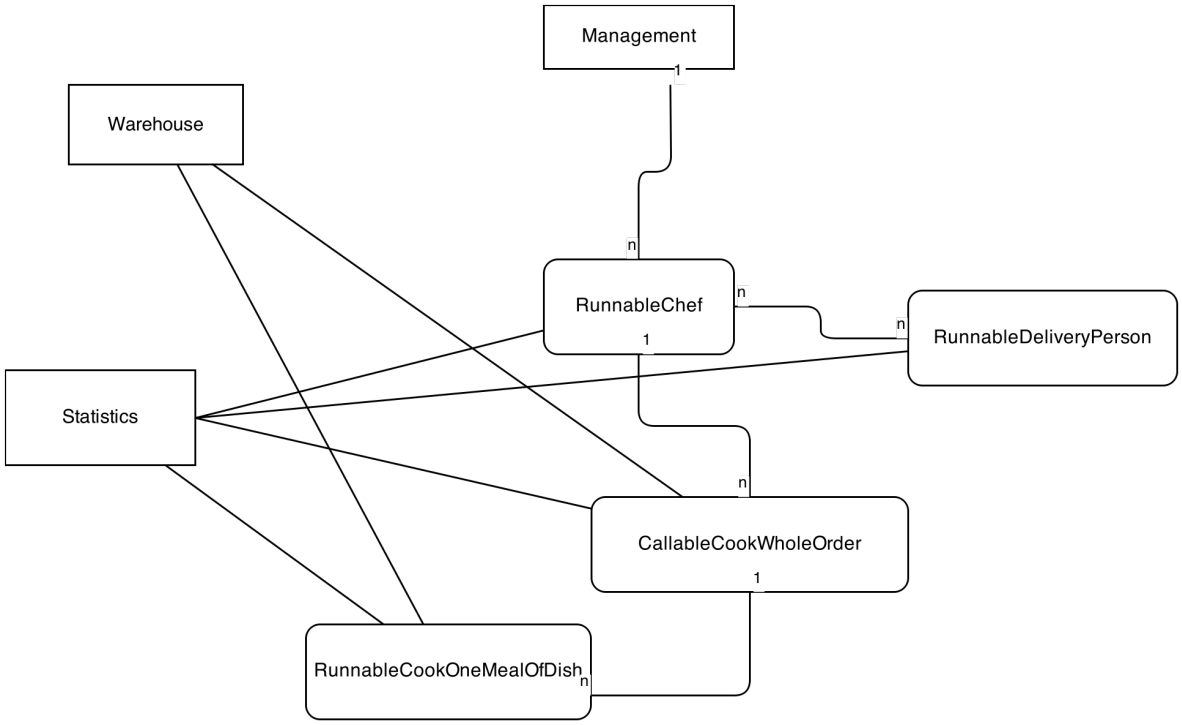
The Management will retrieve the first order received. The Management finds an appropriate Chef in order to give him the order to cook. An appropriate chef is a chef that is currently able to receive this order without being burdened with work. The chef cooks each order by using CallableCookWholeOrder callable object, which uses one RunnableCookOneDish for each dish found in the order. Once the order is **fully** cooked. The chef adds it to the delivery queue. A Delivery Person, retrieves the cooked order, delivers the order to its destination, and updates the Statistics object with the reward.

Once all Orders have been cooked, your application must shutdown all threads related to cooking the orders, and once all cooked orders are sent to delivery, your application must shut down all threads related to delivery. After everything is delivered, your application must shut down completely. Be sure to shutdown all your executors, as well as threads that may be working in your application.

5 Shutdown Process

Remember: Exiting your main does not make any other thread end. You need to allow for graceful shutdown once the application completes its life cycle. When the manager knows that the last order has been delivered you must print out the contents of the Statistics object, and shutdown your application. Which means you need to handle shutting down all the threads found in your application, as well as shutting down all the executors you may be using, and it must be done gracefully. Meaning no abrupt exits of threads. The thread flow must reach the end. Give great thought for threads that are in wait mode when you attempt to shutdown your application. How do you wake them up?

6 Relations Diagram



7 Input Files and Parsing

7.1 Input Files

You *may* assume correct input. You *may* assume the order of information is as you see in this example. There are 3 different input files:

1. InitialData: This file contains restaurant location, a collection of kitchen tools and ingredients in the warehouse, a collection of chefs and a collection of delivery people:
   (a) RestaurantAddress: (x, y)
   (b) KitchenTools: tool name, quantity
   (c) Ingredients: ingredient name, quantity.
   (d) Chefs: chef name, efficiency rating, endurance rating
   (e) Delivery People: delivery person name, speed

```
<Restaurant>
        <Address>
                <x></x>
                <y></y>
        </Address>
        <Repository>
                <Tools>
                        <KitchenTool>
                                <name></name>
                                <quantity></quantity>
                        </KitchenTool>
                </Tools>
                <Ingredients>
                        <Ingredient>
                                <name></name>
                                <quantity></quantity>
                        </Ingredient>
                </Ingredients>
        </Repository>
        <Staff>
                <Chefs>
                        <Chef>
                                <name></name>
                                <efficiencyRating></efficiencyRating>
                                <enduranceRating></enduranceRating>
                        </Chef>
                        <Chef>
                                <name></name>
                                <efficiencyRating></efficiencyRating>
                                <enduranceRating></enduranceRating>
                        </Chef>
                </Chefs>
                <DeliveryPersonals>
                        <DeliveryPerson>
                                        <name></name>
                                        <speed></speed>
                        <DeliveryPerson>
                                        <name></name>
                                        <speed></speed>
                        </DeliveryPerson>
                </DeliveryPersonals>
        </Staff>
</Restaurant>
```

2. Menu: This file contains information regarding the different dishes the restaurant is able to cook and deliver. In case you have missed it, you may assume that the Warehouse contains enough kitchen tools and ingredients for a successful and complete simulation process.

```
<Menu>
	<Dishes>
		<Dish>
			<name></name>
			<difficultyRating></difficultyRating>
			<expectedCookTime></expectedCookTime>
			<reward></reward>
			<KitchenTools>
				<KitchenTool>
					<name></name>
					<quantity></quantity>
				</KitchenTool>
				<KitchenTool>
					<name></name>
					<quantity></quantity>
				</KitchenTool>
			</KitchenTools>
			<Ingredients>
				<Ingredient>
					<name></name>
				<quantity></quantity>
				</Ingredient>
				<Ingredient>
					<name></name>
				<quantity></quantity>
				</Ingredient>
			</Ingredients>
		</Dish>
	</Dishes>
</Menu>
```

3. OrdersList: This file contains a list of Orders, these orders are to be cooked, and delivered.

   Format:

```
<OrderList>
	<Orders>
		<Order id="">
			<DeliveryAddress>
				<x></x>
				<y></y>
			</DeliveryAddress>
			<Dishes>
				<Dish>
					<name></name>
					<quantity></quantity>
				</Dish>
				<Dish>
					<name></name>
					<quantity></quantity>
				</Dish>
			</Dishes>
		</Order>
	</Orders>
</OrderList>
```

7.2 Parsing

Parsing is the *act* of reading text and converting it into a more useful in-memory format, "understanding" what it means to some extent. XML parsers for example, parse XML-formatted files. HTML parsers, parse HTML-formatted files. In our case, we wish to parse XML files of the format detailed above.

The given files follow a *strict* XML structure, and the correctness of input is assumed, which adds to the simplicity of the parsing process. Be sure to read Java XML Parser API and manual before attempting to parse any files.

You must implement a class called "Driver" which implements your "main" function. It receives the file names as arguments, calls the appropriate static functions to parse the different files and create the needed objects, and once all the needed objects are created, the main function executes the simulation process (Management). Your Management is **not** to parse any files. Your parsing **must** be done in *one* place only. Parsing may be done using multiple static functions in a new **static** class which contains the different static parsing functions, and it must be called from your main before launching the simulation process. Your main is **not** to parse any files. It just receives the objects and launches the simulation process.

8 toString()

All objects found in the Statistics object must implement a toString() method which returns a concatenated string of the object's fields. When running the toString() method of the Statistics object, a printout of all its contents in a human readable manner needs to be returned.

StringBuilder is available for your disposal in order to concatenate elements in a loop. Using "+" is bad. *Why*?

When you wish to print the contents of the Statistics object you hold, all you need to do is to print the result of the toString() method of the Statistics object, and inside it, you will run the toString() method of all the objects contained in it, concatinate, and return.

9 Mandatory Requirements

You are *required* to use *all* the following in your application:

1. Blocking Queue
2. Semaphore
3. Guarded Blocks
4. CountDownLatch
5. Executor Service
6. Threads

   You need to be ready to explain where have you used each one of them, as well as explain the reasons behind your decision.

Also, note the differences between: ArrayList and Vector. Be sure to understand which one of the two fits better your needs in each situation.

10 Deadlocks, Waiting, Liveness and Completion

Deadlocks You should identify possible deadlock scenarios and prevent them from happening. Be prepared to explain to the grader what possible deadlocks you have identified, and how you have handled them.

Waiting You should understand where wait cases may happen in the program, and how you've solved these issues.

Liveness Locking and synchronization are required for your program to work properly. Make sure not to lock too much so that you don't damage the liveness of the program. Remember, the faster your program completes its tasks, the better.

Completion As in every multi-threaded design, special attention must be given to the shut down of the system. When all orders are sent to be executed and nothing is waiting in queue, we will need to shut down the program. This needs to be done gracefully, not abruptly.

11 Unit Tests

You are required to write tests and a design by contract interface for the *Warehouse* class, as well as unit testing, according to the test driven development rules. Please note that your submission does not have to include a full implementation of the *Warehouse* class, but only an empty one. Submission of this part is due 12/12/2013, 23:59PM

12 Documentation and Coding Style

There are many programming practices that **must** be followed when writing any piece of code.

- Follow Java Programming Style Guidelines. A part of your grade will be checking if you have followed these mandatory guidelines or not. It is important to understand that programming is a team oriented job. If your code is hard to read, then you are failing to be a productive part of any team, in academics research groups, or at work. For this, you must follow these guidelines which make your code easier to read and understand by your fellow programmers as well as your graders.
- Do *not* be afraid to use long variable and method names as long as they increase clarity.
- Avoid code repetition - In case where you see yourself writing same code block in two different places, it means this code must be put in a private function, so both these places may use it.
- Full documentation of the different classes and their public and protected methods, by following Javadoc style of documentation. You may, if you wish, also document your local methods but they are not mandatory.
- Add comments for code blocks where understanding them without, may be difficult.
- Your files must *not* include commented out code. This is garbage. So once you finish coding, make sure to clean your code.
- Long functions are frowned upon (30+ lines). Long functions mean that your function is doing *too many tasks*, which means you can move these tasks to their own place in private functions, and use them as appropriate. This is an important step toward increased readability of your code. This is to be done *even* in cases where the code is used **once**.
- Magic numbers are **bad**! Why? Your application must not have numbers throughout the code that need deciphering.
- Do *not* reveal the internal implementation of your objects.
  - For instance: You wish to add an Order object to a collection object inside Management object. You do not expose (to other classes) the inner collection object (by using a getter) then add the Order to it, but intead, you send the Order to the Management object, which imeplements a method (such as: addOrder(Order order)) that adds the Order to the collection. This is to be done for all access attempts to internal data of objects. You do not retrieve the internal data from that object, instead, you implement a method for that object which fulfills your desire.

- Another example: You wish to calculate the distance between two addresses? You do not retrieve the (x, y) values from each Address object and do the calculation yourself, but instead implement a function called calculateDistance in Address class, which takes another Address object and calculates the distance internally, and returns the desired value.

Your implementation must follow these steps in order to keep the code ordered, clean, and easy to read.

13 Application Progress: Logger

Logging is the process of writing log messages during the execution of a program to a central place. This logging allows you to report and persists error and warning messages as well as info messages (e.g. runtime statistics) so that the messages can later be retrieved and analyzed. Loggers are a great asset especially for a thread oriented application. See guide here, on how to use a logger appropriately.

Your application needs to implement a logger mostly for informative reasons, and to follow the progress of the application and its different threads. Please store logging data to a file in order to explain the progress of your application to the grader. Make your logger print informative and readable text, so it is easier to follow and understand.

14 Compiling the Application: Another Neal Tool or ant

Did you love makefile? Great! We're going to do the same thing for Java.

You are required to create an Ant file for your application in order to complete it. Ant file is equivalent to makefile that you have learned in class. You are required to read these two small guides:

- What is ant?
- Ant Tutorial

For further, and complete information regarding ANT, you may refer to Ant Manual.

Once you are done reading, and understanding, what Ant is all about, you are required to create an ant build file for your application.

The only mandatory target for Ant build file is "run": *ant run -Darg0=InitialData.xml -Darg1=Menu.xml -Darg2=OrdersList.xml"*. When running ant, you need to compile your application and run it using the showed above input files. The order of the files is *important*.

This is a mandatory requirement, your application must compile without warnings when launched via "ant".

Compiling via Eclipse alone is unacceptable!

15 Submission Instructions

- Submission is done *only* in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.
- You *must* submit one .tar.gz file with all your code. The file should be named "assignment3.tar.gz". Note: We require you to use a .tar.gz file. Files such as .rar, .zip, .bz, or anything else which is not a .tar.gz file will *not* be accepted and your grade will suffer.
- Your tar.gz file *must* include an ant build file configured for your application. The grader will extract your file, build your application using ant build fie, and execute it.
- Extension requests are to be sent to SPL141. Your request email must include the following information:
  - Your name and your partner's name.
  - Your id and your partner's id.

- Explanation regarding the reason of the extension request.
- Official certification for your illness or milium.

Requests without a *compelling* reason will not be accepted.

16 Grading

Although you are free to work wherever you please, assignments will be checked and graded on Computer Science Department Lab Computers - so be sure to test your program thoroughly on them before your final submission. "But it worked fine on my windows based home computer" will not earn you any mercy points from the grading staff, if your code fails to execute at the lab.

Grading will tackle many points, including but not limited to:

- Your TDD design and implementation.
- Your application design and implementation.
- For each point in 9, where have you imeplemented it, and the reasons behind your decision.
- Automatic tests will be run on your application. Your application must complete successfully and in reasonable time.
- Liveness and Deadlock, causes of deadlock, and where in your application deadlock might have occured, without your solution to the problem.
- Synchronization, what is it, where have you used it, and a compelling reason behind your decisions.
- Checking if your implementation follows the guidelines detailed in 12.

17 Questions and Assistance

- In order to help as many students as possible, please ask your questions in the designated forum for the assignment.
- We will *not* answer emails related to the assignment. Please refrain from sending them.
- F.A.Q will be updated at the website of assignment 3. It is mandatory to read the F.A.Q *daily* for updates. All changes noted there are *binding*.
- We will visit the labs and help the students at the times published on the website.
- ENJOY! :-)