Section B

Problem Description: a thread-safe ECO-Logic

In this section you will create a virtual market place where different agents can sell and buy plastic products. Every agent is characterized by an action it will repeatedly perform until interrupted.

There are four types of agents:

• Chemical plant: produce new raw plastic

batches

• Manufacturer: buy a certain amount of raw plastic batches and assemble them into plastic goods to be put on the market

• Consumer: buy plastic goods, use them, and them trash them

• Recycle center: collects trashed goods and recycle their plastic components producing new recycled raw plastic batches, which can be used to manufacture new goods

Your goal is to implement the behavior of each such class of agents, the market place where all the sell and buy operations happen, and some utility classes.

We will begin with these utility classes and then detail your tasks for implementing the agents and the market place.

As usual, for this part of the test you are required to implement your own (thread-safe) data structure, and you cannot therefore use built-in Java collections for these tasks.

Getting Started

The project containing the skeleton files is located in your Lexis home ‘AF: TODO’ directory at:

• AF: TODO ~/TODO/SectionB/

During the test, you will modify only the following files, among those provided in the skeleton. You are free to add as many files as you see fit, but not to modify provided source files not listed below. You can add tests or otherwise modify the test suite.

• utils/{SafeQueue.java, UnsafeQueue.java}: where you will implement a thread-safe and a non thread-safe queue data structure

• domain/agents/{ChemicalPlant, Manufacturer, RecycleCenter}: where you will im- plement the behavior of the different kinds of agents

• domain/MarketPlaceImpl.java: where you will implement the basic features of the mar- ket place

• Main.java: which you may implement to help you debug your code (not assessed)

Do not change the names or the contents of any other provided source class (in the folder src).

Testing

You are provided with a set of test cases in the test directory. The tests aim at exercising a variety of behaviors of your implementation and to further explain what your code is expected to do. The test suite is not exhaustive: even if your solution passes all the tests, your work will be assessed by the examiners, who may also use a different test suite to check your code.

1

What to do

1. Queues.

In the package utils, you will find two classes implementing the interface Queue. Queue provides three methods: void push(E) adds an element at the end of the queue, Optional<E> pop() returns the element at the beginning of the queue or an empty optional if the queue is empty, and int size() returns the number of elements in the queue. A queue imple- ments a First-In First-Out (FIFO) policy, i.e., the first element pushed in the queue will be the first one returned by a pop operation.

Your tasks are:

• implement the class utils.UnsafeQueue. This class is not required to be thread- safe. It should allow to store an arbitrary number of elements. You are free to select the internal representation that you see better fit. [10 marks]

• implement the class utils.SafeQueue, which is required to be functionally equiva- lent to utils.UnsafeQueue (i.e., to implement a queue), but this time it has to be thread-safe. A basic coarse-grained implementation sequentializing all the accesses is worth up to 7 marks. Any correctly implemented finer-grained access control, regardless of its performance, can be awarder full marks.

[15 marks]

2. Market place.

You are required to implement the class domain.goods.MarketPlaceImpl. This class implements three sets of methods (from the interface MarketPlace. sellRawPlastic and buyRawPlastic allow a producer of raw plastic batches (either a chemical plant or a recycle center) to sell them through the market. Manufacturers can then buy the available batches, if any. The market implements a FIFO policy with priority: recycled plastic batches have higher priority than new plastic batches when the method buyRawPlastic is invoked; among recycled raw plastic batches the order in which the batches are available for buying is the same as the order in which they have been placed on the market for selling. (See also MarketPlaceTest for some examples of expected behavior.) If no batches of raw plastic are available, buyRawPlastic returns an empty optional. From an external perspective, the methods sellRawPlastic and buyRawPlastic behaves as the push and pop methods of a priority queue, whose elements are sorted first on the base of their priority (recycled comes before new) and then FIFO.

The methods sellPlasticGood and buyPlasticGood allow a manufacturer to sell plastic goods on the market and a consumer to buy them, respectively. Plastic goods do not have priority values and are available for buying in the same order in which they have been registered fro selling (FIFO).

The methods disposePlasticGood and collectDisposedGood allow consumers to dis- pose of their plastic goods and recycle centers to collect disposed goods, respectively. Also in this case, disposed goods are available for collection in the same order as they have been disposed of.

A market place instance must be thread-safe. Multiple agents may invoke any of its methods at any time. (If useful, you may reuse (part of) your queues’ implementation.)

[15 marks]

2

3. Agents.

The different types of agents are described by corresponding classes in the package domain.agents. All the agents extend the abstract class Agent (that you should not modify). Agent extends Thread and its run() method repeatedly invokes doAction(), and waits a random time after each such invocations. The loop can be interrupted invoking the method interrupt or if the thread itself is interrupted. The class Agent also provides a method think() that realize the random waiting time, and gives access to its subclasses the protected field marketPlace. An agent is instantiated specifying the average thinking time (delay between to consecutive invocations of doAction()) and the market place in which the agent is expected to operate.

Each subclass of Agent is required to override the method doAction() to implement the logic of the different types of agents as described below (again, you should not modify any class in the goods package):

• ChemicalPlant: a chemical plant agent produces at each invocation of doAction() a new RawPlastic batch and sells it on the market place. The constructor of RawPlastic takes one argument specifying if the batch is from new or recycle mate- rial. ChemicalPlant uses only new material. Only for this class, the implementation is provided as an example. You are not required to modify it.

• Consumer: a consumer tries to buy a plastic good from the market. If such good is available (the optional is not empty), it uses the good for a certain time (you can invoke the method think() to simulate such wait), and then disposes it (MarketPlace.disposePlasticGood()).

[3 marks]

• Manufacturer: the signature of the constructor of this class requires the number of plastic batches needed by the manufacturer to produce each plastic good (if such argument is less than 1, an InvalidArgumentException should be thrown). At each invocation of its action, a manufacturer will repeatedly try to buy a raw plastic batch, until it has collected the number required to produce a new plastic good; if the invocation of MarketPlace.buyRawPlastic() returns an empty optional, the man- ufacturer waits some time (think()) before trying again. When enough raw plastic batches have been collected, the manufacturer produces a plastic good (passing the collected raw plastic batches as argument to the constructor of PlasticGood) and sells it on the market.

[3 marks]

• RecycleCenter: every time its action is invoked, a recycle center tries to collect a disposed good from the market. If such good is available, the recycle center iterates over the basic materials composing the good (PlasticGood.getBasicMaterials()) and recycle them according to the following rules. Each batch of raw plastic with origin new is used to produce one batch of recycled raw plastic. Every two batches of recycled plastic from disposed goods are used to produce one batch of recycled raw plastic. The produced batches of recycled raw plastic are sold on the market. Notice that it is not required that the two batches of recycled plastic (which are recycled again into one new batch) come from the same disposed good (and therefore collected during a single invocation of the method doAction()). See also the last test case in RecycleCenterTest for more details. The recycle center only needs to count how many batches of disposed material it processed and produce the corresponding number of recycled raw plastic batches to be sold on the market (by constructing

3

new RawPlastic instances).

[4 marks]

Total for Section B: 50 marks

4