Coursework CSC8016

Giacomo Bergami

24th of April, 2023

Use Case Scenario

We want to implement a virtual shopping system, whether the threads are either clients using the web app, or clients buying products on the physical shop using the mobile app. The lifecycle of any client interaction is summerised by `ClientLifecycle`: Each client interacts with the shop through the `RainforestShop` via the `login` method, through which each client can add items to its basket and perform re-shelving operations (moving the items back to the physical/virtual shelf from the basket). Each client can get a list of available items at the time of the query, basket a given product by name, (re)shelf the product (thus removing it from the basket), checkout and proceed with the payment (thus either buying all the items on the basket or none of those entirely), and logging out of the system while losing the information of all the items being previously put in the basket from the real/virtual shelf (thus entailing implicit re-shelving).

Each time an allowed user logs in, a non-empty transaction will be created with an unique transaction ID. The unavailability of the product shall be confirmed not while basketing the items, rather than after purchasing those. At this stage, the shop supplier (`SupplierLifecycle`) might be notified that some products are missing (`getNextMissingItem`) and refurbish the show with a non-zero amount of products of the same type (`refurbishWithItems`). For simplicity sake, the `refurbishWithItems` method will be in charge of creating the number of desired product and to place them on the shelf.

As in any industrial setting where teams split up the duties, you are assigned an API that you need to implement. Such an API is provided both on Canvas and at `https://github.com/jackbergus/NCL_CSC8016/tree/main/src/main/java/uk/ncl/CSC8016/jackbergus/coursework/project2`. This will then require to finalise the implementation of `RainforestShop` and the integration of concurrency mechanisms in `ProductMonitor`; the `Transaction` class shall not be changed! The `Testing` solves a twofold task: showing how Clients, Suppliers, and the Shop system are communicating, as well as providing some preliminary guidelines on how the coursework is going to be assessed. The student is free to choose whichever is the best way to pass the tests (not fully disclosed to the students) in the `Testing` class. E.g., the RainforestShop could be either modelled as a finer-grained monitor, but inside this monitor at least one thread per logged user should be running; also, such a RainforestShop could be also implemented as a consumer threads handling all of the clients' messages.

Assumptions

- In a realistic example, communications happen between processes via UDP messages. In this module, we don't require that. We can freely assume that each client (Physical person buying items in the show using the mobile app, OnLine Shopper) is mimicked by one single thread. We assume they directly exploit such an interface (no FrontEnd is required!)

- If the RainforestShop is implemented as a server, such a thread might receive the "client messages" through shared variables.
- The RainforestShop already comes with a Transaction class keeping track of the transactions that are performed for handling basketing operations. You are not required to tolerate the server crash (this is more of a back-up task rather than a concurrent programming one), but you must correcly handle client log outs (withdrawn items from the shelves after log-out should be automatically re-shelved with a cookie-free assumption, where the basket is not "rembered" after re-logging in)!
- We assume that the `RainforestShop` class is initialized with the users allowed to shop using the mobile app or OnLine website (`Collection<String> client_ids`), the association between the name of the product (`String`), its cost (`Double`) and a non-zero `Integer` number of available items to purchase (`Map<String, Pair<Double, Integer» available_products`). The students are encouraged to change the `studentId` so to return their student ID, as well as setting `isGlobalLock` to **true** if the students use a pessimistic protocol, and **false** otherwise. in The system should not allow to register/unroll new users/shoppers.
- The server should allow a single user contemporarily loggin in with the same username **as far as different transaction IDs are given to distinguish different concurrent operations**. In order to maximise seriality and concurrency requirements, the students might investigate *optimistic protocols* for transactions, but this is not strictly required.
- A solution might be deemed incorrect for the following reasons:
  1. Products that were originally basketed cannot be bought any more (e.g., both users attempted to basket a product but only one of them was able to buy it).
  2. The same product name with the same product id cannot be bought multiple times.
  3. Somehow, the computations are "logically" correct with single-threaded scenarios, that is basket, reshelf, checkout, and logout implement the expected semantics. Still, this is not sufficient for passing the coursework with full marks.

Submission Requirements

1. `RainforestShop` and `ProductMonitor` should be finalised, as the current implementation does not pass the provided tests!
2. Submit the code as a zipped *Maven* project. with **no** *jar* and *classes*. The source code will be recompiled from scratch, and no jar/class is going to be run.
3. If you want to use an external Java library, please consider the following:
   - The Java library should be explicitly described as a `<dependency>` in the `pom.xml` file, and should only access the libraries from the default *Maven Central Repository*.
   - A library might provide single concurrency mechanisms primitives, but not ready-made solutions already composing those: semaphores, monitors, locks, just logs, thread barriers, thread pools, passing le baton mechanisms are allowed. Code reuse from the exercises and examples seen in class is permitted.
   - Systems completely solving the coursework for you are **strictly prohibited**: e.g., any kind of (data) management system having concurrency control (ensuring safe concurrent thread access to any data representation) and supporting concurrent transactions (implementing

any kind of transaction protocol, either pessimistic or optimistic) **must be avoided**, as they both implement commit/aborts and thread-safe operations on data.

- None of the (direct or indirect) dependencies of the coursework should rely on external servers or processes to run or to be installed.
- The solution should **not** include external jar files.
- If unsure whether the solution might be exploited, please ask before submitting.

4. Attached to the source code, please provide a short report motivating the compliance of the source code to each point and sub-point of the marking scheme. Providing such report in form of comments in the implementation is also fine. New classes might be created for supporting the implementation, but existing classes should be neither renamed or moved to a different package.

Marking Scheme

The marking scheme is capped at **100%**.

- Single-Thread Correctness **[+52%]**

  +4%: I cannot open a transaction if the user does not appear in the users' collection.

  - (You) cannot login (and therefore, start a transaction) if a user is not listed (no user appearing in the initial RainforestShop collection).
  - Cannot login (and therefore, start a transaction) if a user is not listed (User not appearing in the RainforestShop collection).

  +3%: I can always open a transaction if the user, on the other hand, appears on the users' collection.

  +7%: I cannot log-out multiple times using the same transaction, but it should be possible to re-log in, and the novel transaction shall have a different id.

  - Can immediately log-out after logging in on the same transaction.
  - Logging out multiple times on the same transaction is not permitted (returns false).
  - The same user can open multiple transactions (not necessarily being contemporarily open).
  - Each transaction opened by the user should come with a distinct transaction id.

  +7%: I must neither basket nor purchase unavailable products.

  - Cannot basket items when the map is empty.
  - Cannot basket items not listed in the map.
  - Cannot basket more items than specified in the map.
  - Can basket available products.
  - Can shelf previously-basketed products.
  - Can re-basket a product that was previously shelved.
  - After the last point, I cannot re-basket the same item another time.

+3%: Logging out automatically re-shelves all the remaining product non-purchased in the basket, and therefore it shall be possible to re-basket the products.

+3%: Logging out should also automatically disable all the remaining operations available through the transaction (mainly `basketProduct`, `shelfProduct`, and `basketCheckout`).

+20%: Correctly purchasing the available items (single-threaded).

- It should be possible to basket checkout when the basket is empty, with any given amount of money
- After successfully purchasing one item, the checkout returns the correct information.
- After successfully purchasing two items, the checkout returns the correct information.
- When attempting to purchase three items where only two are available, the checkout correctly purchases two items.

+5%: Correctly shelving the products.

- Cannot shelf a product that did not originally exist.
- Cannot shelf a product that was not basketed (empty basket).
- Cannot shelf a product that was not basketed (non-empty basket).
- Can shelf a product that was originally basketed.

- Multi-Threaded Correctness **[+38%]**

+6%: The same user can log-in multiple times.

- The same user can open multiple transactions (contemporarily open).

+6%: Two threads shall never be able (in any possible run) to contemporary access to the same object on the shelf.

+5%: A client running without a supplier shall always dispose the available resources.

- In particular, a `ClientLifecycle` should be able to buy all the available elements if given an adequate amount of money.

+6%: Correct Client/Shop/Supplier interaction. This might be tested with at least one `ClientLifecycle` and one `SupplierLifecycle` running.

- The supplier shall not be triggered if the products are basketed but not bought (as they can be later on re-shelved).
- The supplier shall be triggered when a shelf for a given product is emptied.
- After refurbishment, a client shall be able to buy at least one more product.

+12%: Correct Client/Client interaction (two distinct users and two transactions from the same user).

- The clients bought the maximum number of available items.
- The clients bought 3 distinct items.
- The clients cannot contemporarily buy the same item.

+3%: Correct Supplier Stopping.

    - The supplier is stopped by receiving a `@stop!` message only when the `stopSupplier` method is invoked.
    - Currency is handled correctly.

- Advanced Features
    - **[+5%]** The RainforestShop is emulated realistically as a separate thread.
    - **[+1%]** The code exploits Java's concurrent collections.
    - **[+1%]** The program allows to visually determine the correctness of the operations performed by the threads (e.g., terminal prints or graphical user interfaces).
    - **[+1%]** The student correctly uses ReentrantLocks and Conditions.
    - **[+2%]** The student correctly exploits semaphores.
    - **[+2%]** The student exploited the optimistic transaction principle, where multiple users can log-in (not only the same user multiple times!).
    - **[+2%]** Usage of monitors or multithreaded producers and consumers on the interaction with the supplier (semaphores might be also exploited).
    - **[+3%]** Thread pools are used to handle multiple requests from multiple users.
    - **[+3%]** Any Java library imported via `pom.xml` 'not violating the 3rd Submission Requirement.