NWEN303 Assignment 4 Due date: 19 May at midnight

Topic: Solving Mailbox overflow **[Total Marks: 100]**

In this project you are required to edit the code of **Cakes** in order to solve the problem of Mailbox overflow. You need to implement a very precise specification.

Task1 [20 Marks]

Understand and explain the given code
In your report, explain how the various classes and methods inside **Cakes** work.
This includes **Alice**, **Bob**, **Charles**, **Tim**, **OpenAkka**, **AkkaConfig** and **Cakes**: the class containing the main method.
We expect about 1 page of text.

Task2 [50 Marks]

Solving Mailbox Overflow

Following very closely the specification below edit the file **Cakes.java**:
Alice, **Bob** and **Charles** are **Producer**s.
A **Producer<T>** has a list of stored products, has a maximum size of the list of products, and can be in running or non-running state.
A **Producer** knows how to answer to three messages: **T**, **MakeOne** and **GiveOne**.
Here is how a **Producer** needs to answer those messages.

- **T (a product)**:
  - just add it to the list of products (thus, it can go over the limit).
- **MakeOne,** when the list is full (the size is at or over the maximum size limit):
  - sets the state to not-running.
- **MakeOne,** when the list is not full:
  - **pipe**s a future product to **self**;
  - when the future product is completed, **tell MakeOne** to **self**.
- **GiveOne,** when the list is empty:
  - **pipe**s a future product to the **sender**.
- **GiveOne,** when the list is not empty:
  - removes a product from the list and **tell** that product to the **sender**.
- (additionally) **GiveOne** when the state is not-running and the list is not full:
  - set the state as running;
  - **tell MakeOne** to **self**.

To answer those messages, a **Producer** can **make** future products.
(Since Actors need non-blocking computation, what kind of **Future** should it be?)

The computation actually making the product could be very long, and thus it must be asynchronous with respect to the **Producer** actor, so that they will be able to keep answering other messages.

In more detail: **Alice** and **Bob** will just **make Wheat** and **Sugar**; **Charles** will **ask Alice** and **Bob** for **Wheat** and **Sugar**, he will then **combine** the ingredients to produce a future **Cake**.
Tim now needs to **ask Charles** for **cake**s.

The overall process should keep the same behavior as before.

Edit the file **Cakes.java**:

- Introduce an abstract actor **Producer<T>**, with an **abstract make()** method.
- **Alice**, **Bob** and **Charles** now extend **Producer** and implement **make()**.
- **Tim** code is adapted as needed.
- The method **Cakes.computeGift(int)** can be adapted, for example to satisfy different constructors for **Charles** and **Tim**.
- The code as a whole keeps the same behavior as before. In particular, it can still transparently run on a single machine or on many machines.

NOTE:
Producing cakes with sugar and wheat is, of course, just a metaphor to represent some complex computation that needs to be performed by the various actors.
To respect this metaphor, **only code under the control of Alice can produce Wheat**.
In the same way, only **Bob** can produce **Sugar**, and only **Charles** can produce **Cakes**. Those cakes must be produced using the **Wheat** and **Sugar** objects created by **Alice** and **Bob**.
A solution that breaks the metaphor, where for example **Charles** makes also the **Wheat** and the **Sugar** is invalid and will not receive many marks. If you *'just try to make your code compile'* you are likely to break the metaphor.

Task3 [30 Marks]

Load balancing on multiple machines

- Edit the class **Sugar** so that it now takes at least 200 milliseconds to create a **Sugar** instance.
- Then, modify **Cakes** so that 4 copies of **Bob** (the **Sugar** producer) are used, and reside on 4 different machines.
- Edit the main method to instantiate those 4 **Bob**s and edit **Charles** so that he takes advantage of the 4 **Bob**s.
- In the report, describe the improvement in performance between the solution with 1 **Bob** and the solution with 4 **Bob**s; briefly describe how you measured the performance difference.

TO SUBMIT:

Your submission should include:

1. A jar file with all your code for Task2, without the tweaks of Task3.
   Thus Task2 can be marked independently of your performance while encoding Task3.
2. A jar file with all your code for Task3, including an option to run with 1 Bob or with 4 Bobs. This means you will submit a fair amount of duplicated code that is common for Task2 and 3.
3. Your report in pdf format
4. A txt file stating any bugs in your code and how to run your code (i.e. a readme file)