**Exercise 7: Backward error recovery**

This is the first of a two-part exercise: This time we will look at one way of doing backward error recovery, and next time we will modify this code to use forward error recovery instead. Since we will be needing a rather unique language feature of Ada for our forward error recovery solution, we will be using Ada for both parts, so it is easier to see the similarities and differences between the two approaches.

**Practical information.**

* Have a look at [Intro to Ada](http://www.adaic.org/learn/materials/intro)
* Use the [Ada Reference Manual](http://www.adaic.org/resources/add_content/standards/12rm/html/RM-TOC.html) if you need to look up something.
* Compile your program using gnatmake.
* The code you are completing [can be found here](https://github.com/TTK4145-students-2019/exercise-7-ttk4145_group14/blob/master/exercise7.adb).

**Desired functionality:**

We are modelling a transaction with three participants, where each performs a calculation that is slow when it works correctly, but quick when it fails. The work from each should not be committed (in this case: printed to the standard output) unless *all* participants succeed. When any participant fails, the work from all the others will need to be reset, and the transaction has to start from the beginning.

**Create the transaction work function**

The "work" the participants are doing is adding 10 to a number. Unoriginal, perhaps, but we can use random numbers to have it simulate work that either success or fails. We will call this function Unreliable\_Slow\_Add.

* A random number generator Gen is defined and seeded for you. Call Random(Gen) to get a random number between 0.0 and 1.0. Compare it with the Error\_Rate, and have the function either perform:
  + The intended behaviour: Most of the time, the addition takes up to 4 secs. Use delay Duration(d) (where d is floating-point number) to pause execution for d seconds (can use Random(Gen) multiplied with a constant as the value for d). Then, add 10 to x and return the value.
  + The faulty behaviour: The rest of the time, the operation takes significantly less time (say, up to half a second), but raises an exception instead. A Count\_Failed exception has been defined for you. (Note: Ada uses raise, not throw)

**Do the transaction work**

Now that we have the unreliable slow adder, we need to call it, and also catch the exception it throws.

* The variable we are modifying is called Num, and its previous value is called Prev.
* The structure for exception handling in Ada is [begin-exception-end](http://en.wikipedia.org/wiki/Exception_handling_syntax#Ada). There is only one exception to catch: Count\_Failed. When counting fails, we need to tell the transaction manager that everyone has to revert, by using Manager.Signal\_Abort;(already implemented).
* Both in case of success and failure, we need to know what happened to the other participants. The exit protocol lies in the Finished entry of the transaction manager, but is not yet implemented (We'll get back to this in part 3). Call it, and trust your future selves that it will be implemented properly.
* We then ask the manager if we should commit the result. If not, we have to revert to the previous value.

**Finish the Manager exit protocol**

The exit protocol requires that all participants show up, and all votes are counted. We store the vote using two booleans: Aborted, which is set true by the first participant that aborts the transaction (by calling Signal\_Abort), and Should\_Commit, which stores this value until the next round starts.

The Transaction\_Manager is a protected object. You can find a quick summary of protected objects [here](http://www.adaic.org/learn/materials/intro/part5/##protect). We are also using one additional feature: The Count attribute. We can get the number of tasks blocked on the condition of the Finished entry by using Finished'Count.

* The exit protocol needs to function as a "gate", letting participants through only when all of them have arrived (when Finished'Count = N makes sure no task enters until N of them have arrived).
* When the first participant enters, we open the gate for the remaining participants. When the last one enters, the gate is closed (for next round). Use Finished'Count (or another counting variable) to see how many participants are waiting to enter.
* We also need to update and reset the voting variables. When the first participant enters, it can set Should\_Commit. When the last participant enters, it can reset Aborted (this is the reason we store Should\_Commit as a separate variable).

**Approval**

* Show the student assistants that your program produce the expected results: All participants either add 10, or all participants revert.

**Next time**

These Questions will be answered next time (so think about them now, before you encounter any spoilers):

* We can have a situation where we have multiple failures in a row, denying any progress from happening. What are the real-time consequences of this?
* When one of the participants fails early, all the other participants are doing futile work, yet we cannot do anything but wait until all participants are finished. How can we solve this?