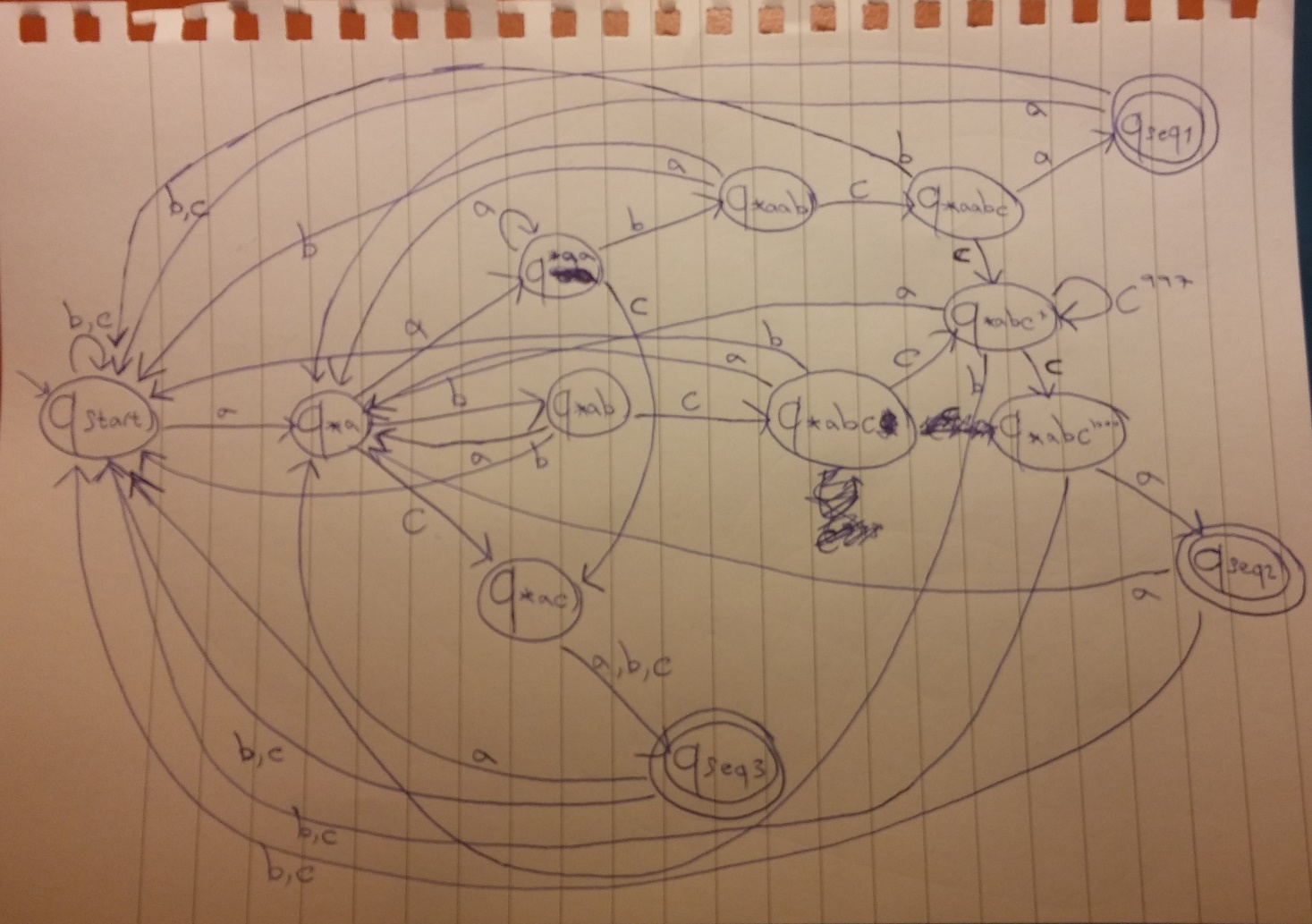
First of all, apologies if the code doesn’t rely on standard conventions of Java coding. I am originally a .NET programmer; but I tried to use the conventions as I remembered them from the last time I wrote code in Java. Hope it’s looking ok.

To implement the state machine framework I used the State design pattern, roughly taken from: <http://www.journaldev.com/1751/state-design-pattern-in-java-example-tutorial>, with a few adjustments to better fit the task’s description of components. For instance, instead of the machine (“context”) or the client changing the machine’s next state, it is being determined by the current state after each event occurs (to fit the requirement that it is the state that should “indicate transitions to a new state”, whereas the machine should be very generic).

The core components in the framework are: State, Event and Machine. Made them as flexible as possible, with a common “doAction” method in the State (when receiving a new event). A specific implementation would define and create subclasses for all types of states and events, and create an instance of the framework’s common machine class, which passes new events to its current state and takes care of persistence. Besides them are helper classes mainly to separate certain groups of actions and for handling files.

The testing project is simply a coded version of the following finite state automaton (sorry for the scribbles, I modified it a little while drawing):



As for the testing machine’s second sequence: the (C \* 1000) requires many logical states, but they are represented by one actual state (class ABCMultiple, or state \*abc+ in the above sketch) that keeps the number of C appearances in a counter variable. Also, there are no classes for the 3 accepting states – seemed redundant as they all have the same transitions as the starting state.

In order for the machine to be persistent, it saves its current state to a file (whose path should be provided by the framework’s client, I suggested the user’s %appdata% folder) after each event, and loads it from the file when the process starts. This is done with serialization of the current State object (which requires it to be serializable, as well as all its subclasses, with a specified serialVersionUID for each class – as explained in <http://www.mkyong.com/java-best-practices/understand-the-serialversionuid/>). On the first time the machine is being run, or if a problem has occurred while loading the state from the file, the machine sets its current state to the default state (also given by the framework’s client).

General comments:

* In some places where exception handling was required I chose to catch the general ‘Exception’ instead of catching each possible exception individually, for the code to be somewhat cleaner, and because all types of exceptions in those places are being handled by the same action. I realize the disadvantages of this approach in some cases, but it seemed to fit here.
* Created a MachineConfig class as a wrapper to the serialized state file, so that additional values can be added there as fields in the future. Could take it one step forward and use a Java .properties file (as demonstrated in <http://crunchify.com/java-properties-file-how-to-read-config-properties-values-in-java/>) but it would take longer to implement and after all it didn’t seem like the point of the assignment.
* For any comments or if anything should be improved please contact me anytime.