CSSE2002 / CSSE7023 **Semester 1, 2013**

Assignment 3

Goal: The goal of this assignment is to gain practical experience with implementing GUIs.

Due date: The assignment is due at **5pm on Friday 24 May**. Late assignments will lose 20% of the total mark immediately, and a further 20% of the total mark for each day late. Only extensions on Medical Grounds or Exceptional Circumstances will be considered, and in those cases students need to submit an application for extension of progressive assessment form (http://www.uq.edu.au/myadvisor/forms/exams/progressive-assessment-extension.pdf) to the lecturer (email is acceptable) or the ITEE Enquiries desk on Level 4 of GPSouth Building) prior to the assignment deadline.

Problem description: In this assignment you will develop a GUI for simulating the interactions between a spy and their informants using the Model-View-Controller design pattern. This extends some of the work you have done in Assignments 1 and 2.

Exactly what the GUI looks like is up to you, but in order to meet testing requirements, it must have the following graphical components (as declared in skeleton code in the `SpyView` class).

For displaying the current knowledge state of the spy:

1. A `JTextArea txtSpy` that, at every stage of the simulation, contains the text

   "Spy thinks ... " + LINE_SEPARATOR + "Secret is true with probability " + p

   where p is a `BigFraction` defining the probability that the secret is true in the current knowledge state of the spy, and `LINE_SEPARATOR` is defined to be `System.getProperty("line.separator")`.

   When the simulation is initialised the `secret` is chosen to be true with probability 1/2 and false with probability 1/2, and the knowledge-state of the spy is set so that the spy believes that that the secret is true with probability 1/2 and false with the remaining probability. The knowledge state of the spy will be updated after meetings with informants or after the spy has guessed the value of the secret.

   The `JTextArea txtSpy` should be placed in `JScrollPane scpSpy`.

For displaying informant interactions and messages:

2. A `JTextArea txtInfo` for displaying informant interactions and messages.

   Initially `txtInfo` should contain the empty string "". It is updated when any one of the three buttons `cmdRead` , `cmdMeet` or `cmdGuess` are pressed (as per description below).

   The `JTextArea txtInfo` should be placed in `JScrollPane scpInfo`.

For reading in lists of informants:

3. A `JTextField txtInformantsFile` for entering the name of a file containing a list of informants. (Each informant is a `TwoCoinChannel`.) Initially, the text field

`txtInformantsFile` should contain the empty string "". After that, only the user should update and modify it.

4. A `JButton cmdRead` with text "*Read Informants*" for reading the informants from the user-specified file in `txtScheduleFile`, adding the informants from that file to those yet to meet the spy in the simulation.

   Initially there are no informants scheduled to visit the spy. Every time the "*Read Informants*" button is pressed, the informants are read from the file using method `InformantsReader.readInformants`. If that method throws an exception trying to read the informants then `txtInfo` is set to contain the string "*Error reading from file.*", and no changes are made to the simulation. Otherwise, the list of informants read is appended to the list of informants currently waiting to meet with the spy, and `txtInfo` is set to contain the string "*Informants added from file*.".

For getting the Spy to meet the next informant:

5. `JButton cmdMeet` with text "*Meet Informant*" for getting the spy to meet the next informant. If there is no informant to meet when the button is pressed (the list of informants to meet is empty) then `JTextArea txtInfo` is set to contain the string "*There is no informant to meet.*", and no further changes are made to the simulation.

   If there is an informant to meet when the button is pressed then the next `TwoCoinChannel informant` (the first one in the list) is removed from the list of informants, and the `informant` flips the coin `informant.getCoinBias(secret)` to determine the `outcome` of the coin flip (heads = true, tails = false), where `secret` was determined once at the start of the simulation, and the knowledge state of the spy is set to be `informant.aPosteriori(knowledgeState, outcome)` where `knowledgeState` is the knowledge state of the spy before meeting the informant. The `JTextArea txtInfo` is then set to contain the String:

   "Informant says ..." + LINE_SEPARATOR +
   "Heads-bias if true: " + informant.getCoinBias(true) + LINE_SEPARATOR +
   "Heads-bias if false: " + informant.getCoinBias(false) + LINE_SEPARATOR +
   "Result is ... " + (outcome ? "heads" : "tails") + "!";

   The `JTextArea txtSpy` should be updated if the knowledge state of the spy changed.

For getting the Spy to guess the secret:

`JButton cmdGuess` with text "*Guess Secret*" for getting the spy to guess the secret based on their current knowledge-state. When the button `cmdGuess` is pressed, then the spy guesses that the secret is true if she thinks that it is true with probability >= 1/2, and guesses that it is false otherwise. The truth-value of the secret is then revealed to the spy, and she updates her knowledge-state accordingly. The `JTextArea txtInfo` is then set to contain the String:

"Spy guesses that secret is " + guess + LINE_SEPARATOR +
"and is ... " + (secret == guess ? "correct" : "incorrect") + "!"

where `guess` is the guess that was made, and `secret` is the value of the secret.

The `JTextArea txtSpy` should be updated if the knowledge-state of the spy changed.

Task: You must implement the simulator `SpySimulator.java` by completing the skeletons of the three classes: `SpyModel.java` , `SpyView.java` and `SpyController.java` that are available in the zip files that accompanies this assignment.

`SpyView.java` has a specially commented section containing variables required for testing. These variables are declared as public, which is normally bad practice! This is done in order to allow the automated testing to work easily. You should be careful to use private variables when you add more.

You should try to make your user interface look intuitive to the user. Most importantly, the user should be able to easily discern the state of the simulation and use it.

Here is an example of how the user interface might look.

There is a simple test driver in the zip file provided on the Blackboard website in order to show how the automated testing will be done to evaluate your assignment. You should also do your own testing. The zip file also contains other classes that you will need to implement the assignment. We will test your code using our versions of these particular classes, in addition to your submitted files.

As in Assignment 1 and 2, you must implement `SpyModel.java` , `SpyView.java` and `SpyController.java` as if other programmers were, at the same time, implementing the classes that instantiate them and call their methods. Hence:

- Don't change the class names, specifications provided, or alter the method names, parameter types or return types or the packages to which the files belong.
- Don't write any code that is operating-system specific (e.g. by hard-coding in newline characters etc.), since we will batch test your code on a Unix machine. (Use `System.getProperty("line.separator")` when you want to add a newline character to a String.)

- Your source file should be written using ASCII characters only.
- You may define your own private or public variables or methods in the classes `SpyModel.java` , `SpyView.java` and `SpyController.java` (these should be documented, of course).

You also must implement those methods as if other programmers are going to be maintaining them. Hence, to improve readability:

- You are expected to use private methods to stop any method doing too much.
- Classes, constructors, private methods and public methods must be commented using Javadoc, including a description of what the class/method does. The only exception is when overriding a method from a superclass or interface, in which case Javadoc is optional. For this assignment, the use of Javadoc tags such as @param is optional, as is the use of pre- and post-conditions.
- Allowable values of instance variables must be specified using a class invariant when appropriate.

Hints: You should watch the newsgroup (uq.itee.csse2002), and the announcements page on the Blackboard, closely – these have lots of useful clarifications, updates to materials, and often hints, from the course coordinator, the tutors, and other students.

The assignment requires the use of a number of components from the Swing library. You should consult the documentation for these classes in order to find out how they work, as well as asking questions on the newsgroups and asking tutors and the course coordinator.

User interface design, especially layout, is much easier if you start by drawing what you want the interface to look like.

Submission: Submit your files `SpyModel.java` , `SpyView.java` and `SpyController.java`, electronically using Blackboard according to the exact instructions on the Blackboard website:

https://learn.uq.edu.au/

You can submit your assignment multiple times before the assignment deadline but only the last submission will be saved by the system and marked. Only submit the source (i.e. .java) files for `SpyModel` , `SpyView` and `SpyController.`

You are responsible for ensuring that you have submitted the files that you intended to submit in the way that we have requested them. You will be marked on the files that you submitted and not on those that you intended to submit. Only files that are submitted according to the instructions on Blackboard will be marked.

Evaluation: Your assignment will be given a mark out of 15 according to the following marking criteria.

Testing of SpyModel, SpyView and SpyController (8 marks)

- All of our tests pass 8 marks

- At least 85% of our tests pass — 7 marks
- At least 75% of our tests pass — 6 marks
- At least 65% of our tests pass — 5 mark
- At least 50% of our tests pass — 4 marks
- At least 35% of our tests pass — 3 mark
- At least 25% of our tests pass — 2 mark
- At least 15% of our tests pass — 1 mark
- Work with little or no academic merit — 0 marks

Note: code submitted with compilation errors will result in zero marks in this section.

Code quality (7 marks)

- Code that is clearly written and commented, and satisfies the specifications — 7 marks
- Minor problems, e.g., lack of commenting or private methods — 4-6 marks
- Major problems, e.g., code that does not satisfy the specification, or is too complex, or is too difficult to read or understand — 1-3 marks
- Work with little or no academic merit — 0 marks

Note you will lose marks for code quality

a) for breaking Java naming conventions
b) for failing to comment classes, methods, variable and constant declarations (except for the index variable of a for-loop),
c) failing to comment tricky sections of code,
d) inconsistent indentation or embedded white-space
e) lines which are excessively long (lines over 80 characters long are not supported by most printers) and
f) inappropriate structuring: Failure to break the program up into logical components
g) Monolithic methods: if methods get long, you should find a way to break them into smaller, more understandable methods using procedural abstraction

School Policy on Student Misconduct: You are required to read and understand the School Statement on Misconduct, available on the School's website at:

http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct

This is an individual assignment. If you are found guilty of misconduct (plagiarism or collusion) then penalties will be applied.

If you are under pressure to meet the assignment deadline, contact the course coordinator **as soon as possible**.