CSSE2002 / CSSE7023 **Semester 1, 2013**

Assignment 2

Goal: The goal of this assignment is to gain practical experience with procedural abstraction – how complex functionality can be broken up between different methods and different classes.

Due date: The assignment is due at **1pm on Tuesday 7 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 Overview: In this assignment, you will continue to implement security analysis software started in Assignment 1.

The super-power has decided the bias of the coin that will be used to decide whether or not to contact the aliens, and knows the identity of each spy, and the sequence of informants that will visit each spy.

From this, the super-power can calculate - as per the first assignment - the knowledge distribution of each spy. For example, if the secret will be true with probability 1/2 and the informants of spy A are:

<if true@1/2 then (0, 1/6), if true@6/11 then (1/6, 1/5) >

then the knowledge distribution of spy A will be

{{true@0, false@1}@1/12, {true@1/2, false@1/2}@1/6, {true@5/9, false@4/9}@3/4}

and if the informants of spy B are:

<if true@1/2 then (1/6, 1/3), if true@5/9 then (1/5, 1/4) >

then the knowledge distribution of B will be

{{true@1/3, false@2/3}@1/4, {true@1/2, false@1/2}@1/6, {true@4/7, false@3/7}@7/12}

Knowing this information, the super-power is troubled by the fact that the spies will learn about the secret, but there is nothing they can do to stop that!

More troubling, however, is the fact that a given spy might have a better chance of learning more about the secret than another spy, given their respective informants. So that no one spy has an advantage over another, the super-power would like the knowledge-distribution of each spy to be equal.

To achieve this aim, the super-power wants a computer program that, given the bias of the decision-making coin, and two spies and their informants, extends their respective lists of informants with zero or more informants, so that the knowledge-distributions of each spy are equivalent. Since the super-power doesn't want to leak any more information than what is necessary, the method has an extra constraint. The extra informants for each spy must be chosen so that any other choice of additional informants that equalises the knowledge-distribution of the spies, produces spies that "*know at least as much as*" the spies produced by an actual solution.

For a given initial-coin bias, we say that a spy A, "*knows at least as much as*" another B, if B's sequence of informants can be extended with zero or more informants so that the knowledge-distribution of A and B become equivalent.

For instance, for decision coin with heads-bias 1/2, and spy A with informants

<if true@1/2 then (1/6, 1/3), if true@5/9 then (1/5, 1/4)>

and spy B with informants

<if true@1/2 then (1/3, 1/2)>

Spy A "knows as least as much as" B since we can extend B's sequence of informants with

<if true@2/5 then (1/2, 2/3)>

so that the final knowledge-distribution of B:

{{true@1/3, false@2/3}@1/4, {true@1/2, false@1/2}@1/6, {true@4/7, false@3/7}@7/12}.

becomes that of A.

This sounds like quite a challenge, but luckily the following algorithm can be used to solve it. It works by, one step at a time, reducing the problem down to one of a smaller size.

Using the algorithm, you will be able to construct either a recursive or iterative solution to the problem.

How to solve the problem:

Let the bias of the decision making coin be aPriori (the probability that the secret is true), and the list of informants of spy A be informantsA, and those of spy B be informantsB.

We will write kdA for the knowledge-distribution of A given informantsA and coin-bias aPriori, and similarly kdB will be the knowledge-distribution of B, etc. For a knowledge-state ks and a knowledge-distribution kd, we write kd(ks) to mean the probability of ks in kd.

We represent knowledge-states by their likelihood that the secret is true, and order them in ascending order of that likelihood.

Base case: If kdA and kdB are equivalent, then no more informants need be added to the list of either spy and so the solution has been found.

Recursive case: If kdA and kdB are not equivalent then we can reduce the problem size by performing the following step as many times as necessary to reach the base case.

Let

i. ks be the smallest knowledge-state for which kdA(ks) != kdB(ks)
ii. X be the spy with the greater weight at ks, and kdX their knowledge-distribution.
iii. Y be the spy with the smaller weight at ks, and kdY their knowledge-distribution.
iv. w be the difference in weight kdX(ks) – kdY(ks).
v. ks0 be the least element in the support of kdY greater than ks.
vi. ks1 be either 1 if ks0 is the largest element in the support of kdY, or the next-largest knowledge-state after ks0 in the support of kdY, otherwise.
vii. r = w **min** kdY(ks0)*(ks1-ks0)/(ks1-ks)

Given these definitions, append an additional informant ch to Y's informants such that

i. ks0 = ch.getCondition()

ii. ks = ch.aPosteriori(true)

iii. r = kdY(ks0)*ch.outcomeProbability(true)

That is, the channel will update kdY in such a way that it breaks ks0@kdY(ks0) into two pieces. One of those pieces is ks@r and the remainder is

(ks0*kdY(ks0) – ks*r)/(kdY(ks0) –r)@(kdY(ks0) – r).

(For interests sake: r was chosen such that (ks0*kdY(ks0) – ks*r)/(kdY(ks0) –r)<= ks1. This guarantees that the coins for the channel exist, and that this is a locally optimal step.)

HINT: you can calculate the coins you need from (i), (ii) and (iii).

<u>**A worked example:**</u>

Consider the example introduced earlier in which: the secret will be true with probability 1/2 and the initial informants of spy A are:

<if true@1/2 then (0, 1/6), if true@6/11 then (1/6, 1/5) >

and the initial informants of spy B are:

<if true@1/2 then (1/6, 1/3), if true@5/9 then (1/5, 1/4) >

Step 1:

kdA = {{true@0, false@1}@1/12, {true@1/2, false@1/2}@1/6, {true@5/9, false@4/9}@3/4}

kdB ={{true@1/3, false@2/3}@1/4, {true@1/2, false@1/2}@1/6, {true@4/7, false@3/7}@7/12}

ks = 0, X = A, Y = B, w = 1/12 , ks0 = 1/3, ks1=1/2, r = 1/12

and so we want to find an extra informant ch to add to spy B such that

1/3 = ch.getCondition()

0 = ch.aPosteriori(true)

1/12 = 1/4 * ch.outcomeProbabiltiy(true)

and so we calculate the extra informant to be if true@1/3 then (0, 1/2) and add it to spy B.

Step 2:

kdA = {{true@0, false@1}@1/12, {true@1/2, false@1/2}@1/6, {true@5/9, false@4/9}@3/4}

kdB ={{true@0, false@1}@1/12, {true@1/2, false@1/2}@1/3, {true@4/7, false@3/7}@7/12}

ks = 1/2, X = B, Y = A, w = 1/6 , ks0 = 5/9, ks1=1, r = 1/6

and so we want to find an extra informant ch to add to spy A such that

5/9 = ch.getCondition()

1/2 = ch.aPosteriori(true)

1/6 = 3/4 * ch.outcomeProbabiltiy(true)

and so we calculate the extra informant to be if true@5/9 then (1/5, 1/4) and add it to spy A.

Step 3:

kdA = {{true@0, false@1}@1/12, {true@1/2, false@1/2}@1/3, {true@4/7, false@3/7}@7/12}

kdB = {{true@0, false@1}@1/12, {true@1/2, false@1/2}@1/3, {true@4/7, false@3/7}@7/12}

This is the base case and so we are finished.

In total, the informants are

informantsA = <if true@1/2 then (0, 1/6), if true@6/11 then (1/6, 1/5), if true@5/9 then (1/5, 1/4)>

informantsB = < if true@1/2 then (1/6, 1/3), if true@5/9 then (1/5, 1/4), if true@1/3 then (0, 1/2))>

<u>Task Overview</u>:

In brief, you will write a method for reading a list of informants from a file, and you will write a method that the world super-power can use to find the additional informants, as described above.

More specifically, you must code methods *readInformants* and *findAdditionalInformants* from class *SpyMaster* in the zip file that accompanies this handout, according to their specifications in the SpyMaster skeleton file.

(Note that to simplify reading informants, the format of the informants to be read has been simplified: see the SpyMaster specification of the method.)

<u>Practical considerations:</u> Starter files for the class you must implement are on the course website in a zip file. This file include specifications of the methods you need to complete. You will also need to add import clauses, and add comments and javadocs as required.

As in Assignment 1, you must complete the SpyMaster class as if other programmers were, at the same time, implementing classes that use it. 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.
- You are encouraged to use Java 7 SE classes, but no third party libraries should be used. (It is not necessary, and makes marking hard.)
- 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
- Any new methods that you add should be private.
- Your source file should be written using ASCII characters only.

You also must implement the SpyMaster class as if other programmers are going to be maintaining it. Hence, to improve readability:

- You are expected to use private methods to stop any method doing too much.
- Private methods must be commented using preconditions and postconditions (require and ensures clauses). Informal description is OK.
- All non-trivial for and while loops should include a loop invariant.

The Zip file for the assignment also includes some other code that you will need to compile your class as well as a junit4 test class to help you get started with testing your code.

Do not modify any of the files in packages csse2002.security or csse2002.math other than SpyMaster, since we will test your code using our original versions of these other files. Do not add any new files either that your code for these classes depends upon, since you won't submit them and we won't be testing your code using them.

The junit4 test class as provided in the `package csse2002.security.test` is *not intended to be an exhaustive test for your code.* Part of your task will be to expand on these tests to ensure that your code behaves as required by the javadoc comments. We will test your code using our own extensive suite of junit test cases. (Once again, this is intended to mirror what happens in real life. You write your code according to the "spec", and test it, and then hand it over to other people … who test and / or use it in ways that you may not have thought of.)

If you think there are things that are unclear about the problem, ask on the newsgroup, ask a tutor, or email the course coordinator to clarify the requirements. Real software projects have requirements that aren't entirely clear, too!

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.

Submission: Submit your file `SpyMaster.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 **SpyMaster** class.

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 SpyMaster (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 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 some printers and are problematic on small screens), and

f) 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**.