Assignment 3/6: The social network

Object-Oriented Programming and Design B2-2011

Department of Computer Science at the University of Copenhagen (DIKU)

Deadline: 2011-12-10, 18:00 CET

Preface

This is one of the six weekly assignments in the course. Every assignment gets a score of either 0, 0.5, or 1, where 1 is full credit, and 0 is no credit. You are required to acquire a sum of at least 4 points (out of the possible 6) to be admitted to the exam.

Types of exercises

The assignments themselves consist of two types of exercises, those that require you to do coding only, and those that require you to respond in prose, or by taking a screenshot. Exercises that require you to do things other than coding are always postfixed with a *.

What we expect of your code

For the coding exercises we expect of you to follow the Code Conventions for the Java Programming Language[1]. Deviation from the conventions does not necessarily lead to a reduced score, but the conventions are in place to ensure that your instructor can read your source code quickly. Additionally, make sure to follow the explicit code specifications in the assignment text as your code might be subject to automated tests. Such tests usually rely on various classes, fields, methods, etc. to be named in a specific way. Also, make sure to structure your code in an intuitive way, and document your source code and interfaces where appropriate. What this last statement means exactly should become ever more clear as we advance through the course.

What to hand in

For each assignment you should hand in *one* zip archive. The actual structure of the archive, and hence the files we expect you to package into it, are specified at the end of every assignment. To allow us to easier grade the assignments, you are strictly required to follow the following naming convention:

`<full-name>-A.<number>.zip`

The name must additionally be lower-cased. `<full-name>` should be a dash-separated string of characters `a` to `z` representing your name as registered on Absalon. If you have special characters in your name, such as `æ`, `ø` and `å`, please use their `a` to `z` equivalents of `ae`, `oe` and `aa`, respectively. So for example, if your name is Kåre Søren Æbelsen, for the first assignment we'd expect you to hand in a file with the following name:

`kaare-soeren-aebelsen-A.1.zip`

Please make sure that you hand in your *source code* and *not* java bytecode. It is always a good idea to check that all of your files have been uploaded correctly by downloading them again and browsing them yourself.

As for the written exercises, your solutions should be written in plain text format. We make no explicit requirements as to whether they should be written in Danish or English, but try to stick to one. Additionally, this depends on the preferences of your instructor.

[1] http://www.oracle.com/technetwork/java/codeconv-138413.html

Collaboration and help

We encourage you to discuss your solutions with your peers, but your solution *must* be individual. We expect you not to share explicit code or to take credit for another student's work. All in all, we expect you to honor each other's, and your own work.

You may ask your instructor for general help with the assignment or for additional clarifications. You can also use the course discussion forum on Absalon for the exact same purposes. Please remember to read the forum rules before you post anything on the forum. You should especially remember to prefix the forum topics with e.g. A.1.2 if you're asking a question about exercise 2 in assignment 1.

Resubmissions

If you were so unfortunate to not get full credit for your solution on the first try, you can revise, and resubmit your solution. This goes for the first 5 the first time you submit assignments, but *not* the 6th due to time constraints.

The resubmitted file should follow the following naming convention (continuing our running example):

```
kaare-soeren-aebelsen-A.1.resubmit.zip
```

Deadlines

The initial assignment deadline is usually **Saturday, 18:00 CET**. The instructors should have graded your assignment before the following **Wednesday, 23:59 CET**. The deadline for the resubmission is hence the immediate **Sunday, 23:59 CET**. This gives you two full work days to work on you resubmission. You may have a special agreement with your instructor regarding these deadlines in general.

Some conventions

When we say "book", we refer to the main course literature, namely Objects First With Java 4th Edition, by David Barnes & Michael Kölling, and when we say "project library" we refer to the project library on the accompanying CD, also provided via Absalon under "Course material".

Handed out files

```
./A.3.pdf
./A.3.1.txt
```

Credit assignment

The intent of this assignment is for you to exercise your understanding of simple object-oriented design, hash tables and command line interfaces (CLI).

As you solve assignment A.3.1 (page 2) , it might prove useful to devise unit tests before developing various subexercises. This will allow you to spare much time when testing your implementations. In general, when developing applications with user-friendly interfaces, it can prove extremely beneficial to development time to implement a suite of test-cases that can be run automatically instead of the developer having to waste time "navigating" to the various functions in the user interface.

A.3.1 Doing linear algebra

In this exercise we ask you to implement a small program that can do some linear algebra. It is clearly *disallowed* to use any of Java's built-in linear algebra libraries. We'll intentionally omit the details and tell you only what sort of functionality we expect, and how we want to be able to interact with the library. It is up to you to make the remaining design decisions.

However we do have a few explicit requirements:

1. Create a class `Program`.
2. Add a `main` method to the class `Program`.
3. Create a class `Matrix` to represent the actual matrices.

For the sake of simplicity we'll let all the values in the cells of the matrix be good old `ints`. That is, you can assume that the program isn't fed with arguments that will cause integer overflow, or imprecision due to flooring integer division, when performing matrix operations.

We want to be able to perform the following operations:

1. Define square matrices as variables, i.e. associate a name with a matrix for further backreference. If a variable with the same name already exists, it should be overridden. There should be two possibilities:
   - Define a square diagonal matrix by specifying it's size and the value along the diagonal.
   - Define a square matrix by specifying it's size followed by the individual rows.

   Hint: A variable associates a name with a value. This should indicate to you that it is a good idea to define e.g. an `HashMap<String, Matrix>` datastructure to keep track of the defined variables. "Defining" a variable is hence simply a matter of inserting the name (`String`) and value (`Matrix`) into the hash table and "using" a variable is a matter of getting the value corresponding to the given name from the hash table, if it exists.

   Note: You can assume that the user does not make use of undefined variables.
2. Print a matrix defined as some variable. The rows should be separated by line breaks and the columns should be separated by a single space.

   Note: Don't align the cells of the matrix as in e.g.

   ```
   1  2
   34 5
   ```

   For simplicity, keep to one space between the columns of a row, like so:

   ```
   1 2
   34 5
   ```
3. Transpose an $n \times n$ matrix defined as some variable. The result should be stored in the same variable.

   Hint: ${A^T}_{ij} = A_{ji}$.
4. Compute the dot product of two matrices with the sizes $m \times n$ and $n \times o$. Save the result under the same name as the first input variable. If the varible sizes don't match, tell the user and return to the command-accepting state.

   Hint: $z_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + a_{i3} \cdot b_{3j} + \ldots a_{im} \cdot b_{nj}$

We want to be able to interact with the program via a command-line interface (CLI). The interface should be *stateful*, in the sense that the program can only be in one state at a time, and when the user submits input, a *state transition* occurs, moving the program from one state to another. Figure 1 (page 4) shows a state transition diagram showing you what sort of states your program can be in.

Note: The diagram is not complete, some details are left up to you to decide.

Note: The `TechSupport` program in the book can be recommended as a guideline. See page 131-134 for an example of how to read input from `System.in`.

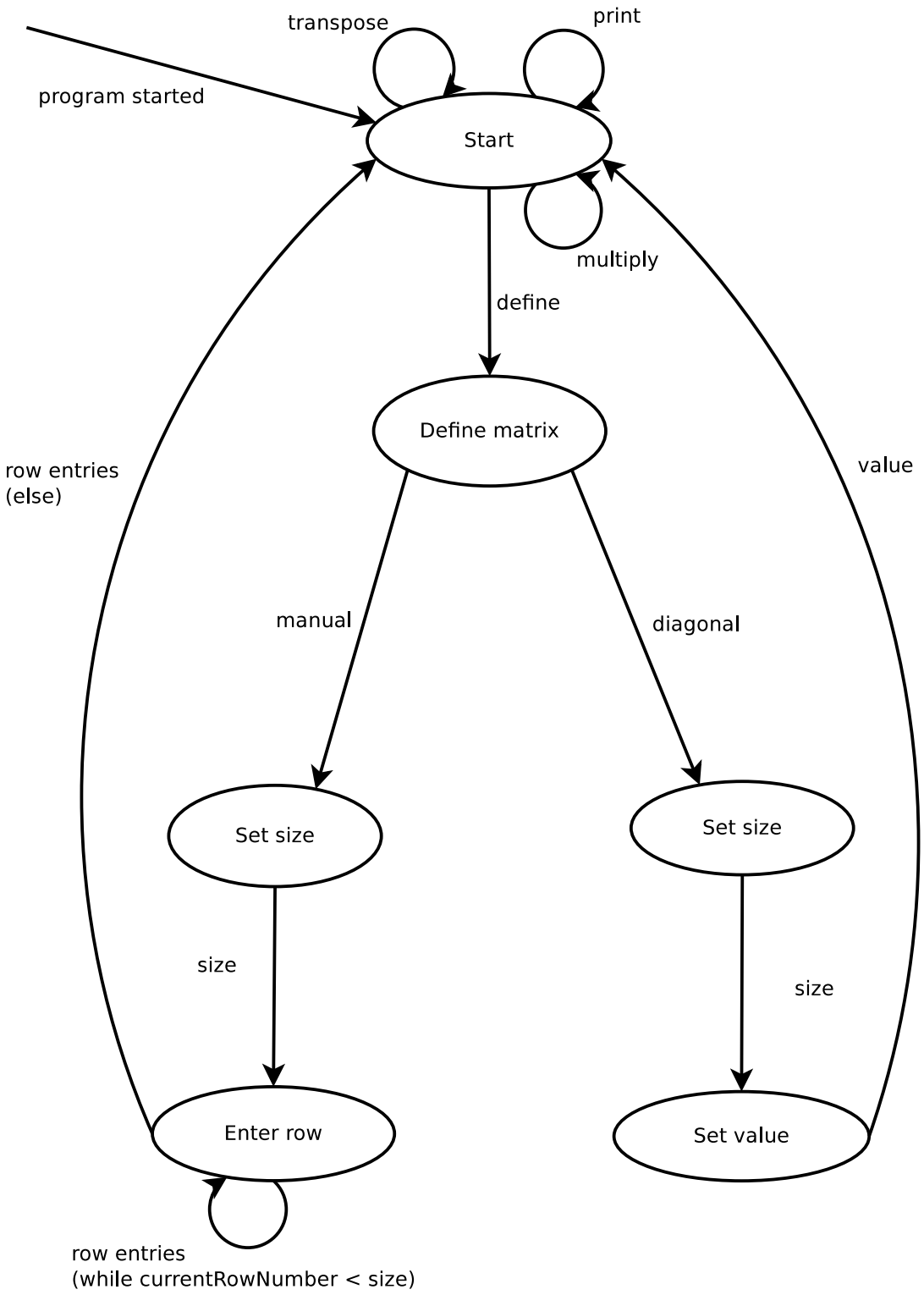

Figure 1: A state transition diagram for the desired program.

For commands you can assume that the function names and operators are split by a single space, and variable names are strings of characters in the range `a-z` in the regular English alphabet. For instance, multiplication is done using the command `multiply alpha beta`.

A sample session with the desired program has been attached as `A.3.1.txt` in the handed out `zip` file. We provide an example instead of making an explicit specification for simplicity. Please make sure to produce all lines as indicated.

Note: The lines starting with > are those where the user enters a command, and hence not generated by your program.

A.3.2 When the Robbers Came to Kardemomme Town

In this exercise we'de like you to simulate what happened in the play "When the Robbers Came to Kardemomme Town"[2]. In our simulation, the town will have 3 residents, and there will be 3 robbers. We'll only simulate the part of the story where the robbers come to the town every day and each visit a house to try to rob it.

In our simulation, the residents of the town have their belongings stored in their own instance variables. A belonging has only a name, and can hence be stored in a `java.util.HashSet<String>` for each resident and robber. Every belonging should be unique, that is, have a unique name with regards to the entire simulation[3].

All 3 thieves visit the town every day. Each day a thief visits one house and no house is visited twice in a day. Which thieves go to which houses on any given day should be chosen pseudorandomly. The residents are always in their houses.

The following sub-exercises will provide you with further details.

1. The class `Kardemomme` represents the simulation. It should keep the 3 thieves in an array (of `Theif` objects), and the 3 residents (as defined below) as individual instance variables. This class should also provide a public method `simulate()` that simulates the play as discussed above.

2. The class `Thief` representing a thief, according to the story, either Kasper, Jesper or Johnathan.

3. The class `OldTobias` represents the old man Tobias that can't protect himself from the thieves. His set of belongings has public access.

4. The class `AuntSofie` represents the aunt Sofie character, whom is strict and brave when it comes to stopping thieves. Her set of belongings is private, but she has a public method `steal()` that allows a thief to steal a belonging from her set. However, Sofie sometimes (chosen at random) catches the thief in action and hits him on the head. The thief gets nothing in such a turn and is so badly injured that he/she can't steal anything for another three turns.

5. The class `PolicemanBastian` represents the town policeman. His set of belognings is also private, but he too has a `steal()` method and he sometimes (like `AuntSofie`) catches the thief red handed. His punishment is more severe, since not only does thief get nothing in the round, the policeman takes all of the thiefs items and gives half of them back to himself and half of them to `OldTobias`. He doesn't like `AuntSofie` for being so private so she doesn't get anything.

Files expected to be handed in

We expect for *at least* the following list of files to be handed in for full credit. `./` resembles the root of your zip archive.

```
./src/linearalgebra/Matrix.java
./src/linearalgebra/Program.java
```

[2] http://en.wikipedia.org/wiki/When_the_Robbers_Came_to_Cardamom_Town#Plot_summary.

[3] We'll look aside from the fact that thieves and residents can counterfeit belongings by adding them to their sets.

```
./src/kardemomme/Kardemomme.java
./src/kardemomme/Thief.java
./src/kardemomme/OldTobias.java
./src/kardemomme/AuntSofie.java
./src/kardemomme/PolicemanBartian.java
```

Challenge of the week

Challenge of the week is an optional part of the weekly assignment for students that have completed the mandatory exercises above. You should feel confident enough about your solutions to the exercises above before taking up the challenge, as there is *no extra credit* for solving it. However, the most elegant solution is rewarded recognition and a free beverage from the canteen. The elegance of the solution is judged by it's exploitation of object-orientation, the Java language and it's libraries.

The deadline for completing the challenge is the same as the assignment deadline, however there are *no* resubmissions. You should upload your solution to the challenge into a separate folder, which you'll find on Absalon.

The challenge

1. Extend the definitions of transpose and multiply in assignment A.3.1 (page 2) so that the result can be assigned to a new variable. For instance, assume that we've already defined the matrices `a` and `b`, then we want to be able to do the following:

   ```
   c = multiply a b
   ```

   Note: You shouldn't break the already working functionality.

   Note: You can assume for all tokens to be separated by a single space.

2. Extend the syntax of the commands so that multiple commands can be specified on one line if separated with a semicolon. For instance, the following code:

   ```
   > define a; diagonal; 3; 5; print a
   ```

   Should have the same effect as:

   ```
   > define a
   > diagonal
   > 3
   > 5
   > print a
   ```

   Note: You can assume for the semicolon to always be followed by a single space.

   Note: Some output from the program has been supressed for simplicity.

3. Implement unary functions.

   We want to be able to do the following:

   ```
   > define a; diagonal; 3; 5
   > f x <- transpose x; multiply x a
   > define a; diagonal; 3; 3
   > define b; diagonal; 3; 6
   > define x; diagonal; 3; 8
   ```

```
> y = f b
> print y
30 0 0
0 30 0
0 0 30
> print x
8 0 0
0 8 0
0 0 8
```

Note: Some output from the program has been supressed for simplicity.

The line where we define `f` means the following: define a unary function `f` that takes in an `x`, transposes it, and multiplies it with `a`, whatever `a` is defined to before we entered the function definition.

Note: For simplicity, we'll let only unary functions to be defined.

Note: For simplicity, you can assume that the the single possible function argument and the operator `<-` are followed by a single space.