COMP122 Assessment 4

Worth 25% of the final course mark.

Submission Deadline

Wednesday, 16 May 2018, 5:00pm (Wednesday of Week 13)

Learning Outcomes. This assessment addresses the following learning outcomes of COMP122:

- Describe object hierarchy structure and how to design such a hierarchy of related classes.
- Describe the concept of object polymorphism in theory and demonstrate this concept in practice.
- Demonstrate concepts of event-driven programming and be able to design simple GUI to demonstrate this understanding.
- Identify and describe the task and issues involved in the process of developing interactive products for people, and the techniques used to perform these tasks.

Part I (50% of assessment mark)

Introduction

The goal here is to develop a relatively simple GUI to demonstrate that you understand the general techniques used.

The Vending Machine. Obviously there's no actual vending machine that will be used here, but you will develop a frame that shows nine products, each having a stock level and a price. The idea is that when pressing a button, the corresponding item will have its stock decremented by one item, and the total sales that is being tracked will increase by the amount of the item that has been "sold".

The GUI will also include another button that will open up a "Vendor's Window". This window will show the total amount of sales (since the last time this button was pressed). This window will also have a button that will reset the stock levels to their start values, and will reset the total sales to 0.

Requirements

Your goal is to implement this GUI with the functionality described above. You should note the following:

(1) **(Requirement)** Design a main class that extends the JFrame Java class. You can model this on the examples discussed in the course notes, for example.

There are nine products to be "sold" in this vending machine. Each product will have its own button element that, when pressed, will decrement the stock of the particular item and will add its price to the total sales of the vending machine.

So you need to implement an appropriate ActionListener that will respond to these button presses.

The GUI that I have developed looks like the following:

Please note that the exact look and feel of your program will depend upon the operating system you are using, as well as (possibly) the version of Java, the libraries, etc.

The important things to notice are the prices of the items, and the initial stock of each item.

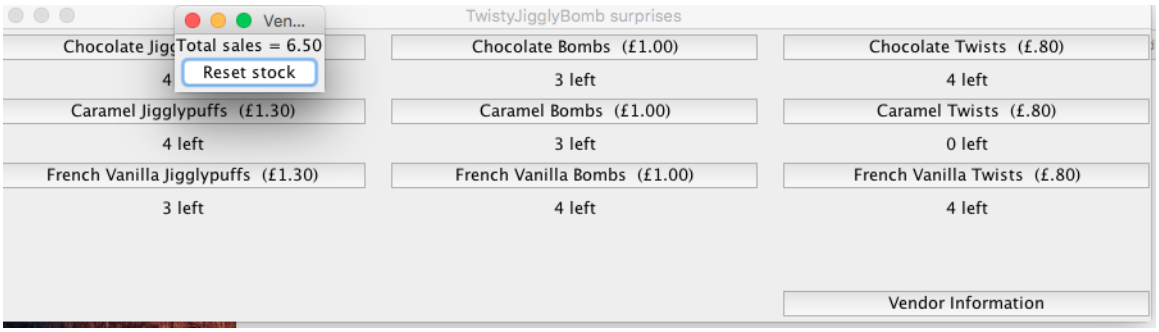
The exact design is left to you, but it should be clear what the price of each item is, as well as the stock level of each item. Each item is on a JButton element here, and clicking on that button will decrease the stock of the item and add the price to a running total of sales (which can be displayed as described later).

The stock levels of items should be updated on the main window (you can edit text contained in JLabels, for example, by using the `setText` method).

(2) **(Requirement)** Note that there is an additional button in the bottom right hand corner. Obviously this type of information won't be available to the "buyers" who would use this vending machine, but we put this here for convenience.

Pressing this button should bring up another separate window, that shows the total sales, and has a button to reset the stock levels (back to the starting values of 4). This "reset" button should also reset the total sales to 0.

As I stated, this should be in another window. You can see this (very small) window over top of the main one in the picture below.



(3) **(Requirement)** The other functionality that your GUI should have is that when a user presses a button for an item that has no stock remaining, they should get a window that tells them that there are no items remaining. (Obviously, stock shouldn't go negative, and no value will be added to the total sales in this case.)

As example of this type of message is also shown below:

(4) **(Requirement)** Closing the main window should exit your application. Closing the "Vendor Information" window should *not* close the application, nor should closing the other "no stock left" message windows.

Sample output

Sample output is essentially shown above. As stated, stock levels should be updated on the main screen as they decrease or are reset by the "reset" button. Users should get a message box if they select an item that has 0 stock remaining.

Hints

1. You could use several classes or one class, that choice is up to you.

2. As stated, you will need at least one ActionListener to capture events as buttons are pressed. In my implementation, I actually only had one action listener that worked for all buttons. You can get the source of the button press by using the `getSource` method of an event, which will return a `JButton` element (we know that we will get a `JButton` element since that is all we are using in this case. And then you can compare that element to see if it is equal to another `JButton`. For example, if `e` is an `ActionEvent`, we can get the source of the button press (since we know it will have been generated from a `JButton` element in this case) with this code:

   ```
   JButton pressed = (JButton) e.getSource();
   ```

   If `button` is another (already declared) `JButton`, we can test if the event was generated by that element with this type of test:

   ```
   if (pressed == button) ...
   ```

3. Using appropriate tests, I actually had *one* `ActionListener` that I attached to all button elements. You can use one or several such listeners. If you use several, you will likely need to implement one or more of them as external classes (similar to how I implemented the `CloseButtonListener` class, i.e. have a class that implements the apprpriate interface, define a constrcutor that could take parameters as input, etc,.

4. I would suggest using Java arrays. Make an array of strings for the button labels. Make an array of costs for the items that are being sold. Make an array of `JButton` elements for the nine items. Make an array of `JLabel` items for the "stock levels". And so forth... Use **`for`** loops to initialize many of these elements and insert them in the frame. With a little thought, you can use such **`for`** loops, rather than creating and inserting these elements individually. As stated before, a `JLabel` element has a `setText` method that can be used to set (alter) the text that a label is displaying.

5. Use a **`for`** loop to identify which button has been pressed for an item that has been "sold". As stated previously, you can get the source of an event, and then compare it to (previously defined) button variables to see if they are the same (to identify which button fired the event).

6. The arrays you create should probably be class variables so that you can access them and/or change them as appropriate. Declare them outside of the constructor, but create them inside the constructor. Then other methods can access those variables.

7. Likely you should create the main code to draw the GUI first (or at least the main window), and then add the code to get the functionality you want after that.

8. If you haven't figured this out by now, code incrementally. In other words, write a small bit of code and test that. Fix any problems before trying to write more code.

9. Check the examples given in class, and the couple of practicals given about GUIs and events.

Some small additional amount of coding

If you like, choose a (friendly to the eyes color scheme) for the buttons in the GUI. For example, make all of the "Jigglypuff" buttons have the same color. Or, have all the "Chocolate" buttons have the same color. This color shouldn't obfuscate the text on the buttons. If the stock of an item drops to 0, change the background color of the corresponding `JLabel` field to indicate this to the user. Again, this background color shouldn't obfuscate the label text. Be certain to change the background color back to "normal" when the item gets restocked by the "reset" button.

Part II

Doors

I see lots of doors...

Now we're going to consider a situation where there are people who like to open and shut doors in possibly strange fashions. What we are going to be interested in is how many doors are open when one (or more) of them is done with their task.

The setup

For a given integer $N \geq 1$, we have a set of $N + 1$ doors labelled (say, left to right) $0, 1, 2, \ldots, N$. For the purposes of this problem, we are going to have $1 \leq N \leq 1000000 = 10^6$. This will allow us to use a Java array to store the status of a door, being either closed or open. Note: I would actually suggest an array of **`boolean`** values, say where **`false`** means the door is closed and **`true`** means the door is open. I am going to use the word "toggle" to mean that someone changes the status of a door, i.e. if it's closed, they open it, and if it's open, they close it.

The players

We have three different people we are going to consider for this problem. These people have a sequence of actions that will mean they open and/or close doors as appropriate.

Ginny

Ginny will toggle doors based on the "greatest common divisor" of two numbers. In case you don't know/remember what the "greatest common divisor" of two integers is, it's the largest integer that evenly divides both of them. We will use "$\gcd(a, b)$" to denote the greatest common divisor of two integers a and b, where at least one of a and b is not equal to 0.

So, for example $\gcd(12, 5) = 1$, $\gcd(36, 28) = 4$, and $\gcd(240, 36) = 12$.

Thankfully, it is relatively easy to find the $\gcd$ of two numbers. In fact, it's the so-called "Euclidean algorithm" that does this. (I know that Prof. Wong has told you that lots of algorithms have people's names associated with them. This is another such example.)

Pseudocode for the Euclidean algorithm is given below (recall that "$a \bmod b$" is the remainder when a is divided by b):

```
GCD(a,b)         (The Euclidean Algorithm)
   Input Two nonnegative numbers a and b, not both 0.
   Output The greatest common divisor of a and b.

   If b==0:
      return a
   else:
      return (b, a mod b)
```

For a given N (which we are assuming is at least 1), Ginny will toggle all doors labelled k where $\gcd(N, k) = 1$. (Note that this means that Ginny will never toggle the door labelled 0, since $\gcd(N, 0) = N$ because $N \geq 1$.)

For example, suppose we start with all doors closed when $N = 9$. (Remember we will have 10 doors, since there are doors labelled $0, 1, 2, \ldots, N$.) Here we use 0 for a closed door, and 1 for an open door. Suppose that all doors start closed, so we have this initial configuration:

$$0000000000$$

In this case we note that $\gcd(9, 1) = \gcd(9, 2) = \gcd(9, 4) = \gcd(9, 5) = \gcd(9, 7) = \gcd(9, 8)$. Therefore Ginny will toggle doors $1, 2, 4, 5, 7$, and 8. This will give this configuration:

$$0110110110$$

With $N = 10$, starting with all closed doors, after Ginny does her task, we will end in this configuration:

$$01010001010$$

Petra

Petra likes prime numbers. For those that don't remember, an integer $x \geq 2$ is a prime number if the only numbers that evenly divide x are 1 and x. So $2, 3, 5, 7, 11, \ldots$ are the first few prime numbers. Note that 1 is not considered to be a prime number. Numbers such as $10, 27, 143$ and 156 are not prime (they are composite numbers).

How can we tell if a number $N \geq 2$ is a prime number? We can try all numbers k, where $2 \leq k \leq N-1$ to see if k evenly divides N. (Actually, we don't need to try $N - 1$.) In fact, we only really have to check the integers that are less than or equal to $\sqrt{N}$, since it N is not prime, then at least one of those numbers will evenly divide N. So here's some pseudocode that will check if N is a prime number:

```
isPrime(N)
   Input A positive integer N >= 2.
   Output A boolean value to denote if N is prime or not.

   If n==2 or n ==3:
      return true
   else:
      for i = 2 to sqrt(n):
         if (N mod i) == 0:
            return false
      return true
```

Petra does the following action for a given value of $N \geq 1$:

For <u>each</u> prime number p that is less than or equal to N, Petra toggles all doors with labels $0, p, 2p, 3p, 4p, \ldots$. This is all multiples of p that are less than or equal to N. As stated, Petra will do this for each prime number

less than or equal to N. Therefore, some doors may get toggled multiple times during Petra's task. Note that it doesn't matter the order in which Petra considers the prime numbers, as long as she gets all the necessary ones. (Why?)

As before, let's consider the case when $N = 9$, with all doors initially closed:

$$0000000000$$

The first prime is 2. So Petra toggles doors $0, 2, 4, 6,$ and 8 giving:

$$1010101010$$

With 3, she toggles $0, 3, 6,$ and 9 giving:

$$0011100011$$

With 5, she toggles 0 and 5 giving:

$$1011110011$$

And finally, with 7 Petra toggles 0 and 7 resulting in this final configuration:

$$0011110111$$

Sven

Sven likes numbers that are perfect squares, $1, 4, 9, 16, 25, \ldots$. Sven does the following:

Starting at 0, he walks a square number of doors away from his position (to the right). When there, he toggles the door. Then he walks another, larger square number of doors away from his new position (to the right). Sven toggles the new door he faces. Then Sven returns to door 0. Sven never repeats the same pair of numbers, but does all possible pairs that will not take him past the last door.

Note that if Sven can't toggle <u>both</u> doors for a given pair of square numbers, he won't use that pair at all.

Suppose with $N = 9$, we again start with all closed doors:

$$0000000000$$

Sven will consider the pair $(1, 4)$ (they are both square numbers, and the second number is larger than the first). So he will toggles doors 1 and 5 (he walks 1 door, toggles it, and then walks 4 more doors, and toggles that one). This gives:

$$0100010000$$

In this case, that is all Sven will do. Any other pair, such as $(1, 9)$ will take him past the last door numbered 9, since he would want to toggle doors numbered 1 and 10 (but there is no door numbered 10). A pair like $(4, 9)$ will obviously also take him past the largest numbered door.

Note that Sven never toggles door 0.

Requirements

Your goal is to implement the three behaviors of these people describe above.

1. **(Requirement)** Call your application program "Doors.java".

2. **(Requirement)** As in Assessment 3, you should design an abstract class, one that can have some common behavior for all three people. It is up to you to decide the methods to put in the base abstract class, but you should put as much as you can into the base class.

3. I would suggest having (at least) two constructors, one that will take an integer as a parameter (it could have other parameters too). This parameter would be N. (Recall that there are $N + 1$ doors in total.)

   A second constructor can take an array of values as a parameter. (You will see why when you get to the "extra question" later.)

4. I would also suggest that the abstract class has a method that will handle the behavior of the person. This (abstract) method should do all of the "door toggling" that person will do. So, for example, for Ginny it will find all numbers k with $\gcd(N, k) = 1$ and toggle all those doors. Each person can then be a subclass that implements this method.

5. **(Requirement)** Your program should receive the value of N as a command-line parameter. You should check that this is an integer, and is between 1 and 1000000 (inclusive). If your program receives a non-integer parameter, or something outside these bounds, your program should stop and give the user an appropriate error message.

   Use some Java exception handling to help with this checking. You can choose to ignore any parameters beyond the first that are given to the program.

6. **(Requirement)** For each person, have them start with a configuration of all closed doors. Then after performing the full procedure for that person, report on the number of doors that are open. To be clear (stating it once again), each person should start with all closed doors.

Sample output

Note that I am showing the final configuration of doors. You do not need to do this. And you shouldn't, as your progam is supposed to handle integers up to 1000000. I did it here for testing purposes.

```
$ java Doors 9
Ginny
0110110110
6 doors open

Petra
0011110111
7 doors open

Sven
0100010000
2 doors open


$ java Doors 30
Ginny
010000010001010001010001000010
8 doors open

Petra
001111011101010011010001010101 1
17 doors open

Sven
000011000110010001001000011001 0
10 doors open


$ java Doors

   Oops, not enough arguments!
   Usage: java Doors N
```

```
$ java Doors 23
Ginny
01111111111111111111110
22 doors open

Petra
10111101111010010011010001
14 doors open

Sven
01000100001001000010001000
6 doors open


$ java Doors 1
Ginny
01
1 doors open

Petra
00
0 doors open

Sven
00
0 doors open


$ java Doors 25000
Ginny
10000 doors open

Petra
12593 doors open

Sven
4084 doors open


$ java Doors 67325
Ginny
53840 doors open

Petra
33899 doors open

Sven
9880 doors open
```

An extra question to consider

Modify your program to accept a second parameter. This second parameter is *optional*, so your program should still work correctly if only given one parameter. You can choose to ignore a third, fourth, fifth, etc parameter that the program might receive in this case. So, again, one parameter is required (the number N as before), the second is optional.

The second parameter (if present) should be a single character or String (your program should handle both cases). Each element of this String should be a letter, one of "g", "p", or "s" in lower- or upper-case. If there is a second parameter, then instead of the original program behavior above, you should do the following:

Start with a configuration of all closed doors. For each letter in the String, have the corresponding person apply their procedure to the doors. **DO NOT** reset the doors in between people in this case. Print out the number of doors that are open after this whole sequence of operations.

Note several things: (1) You should reject characters that aren't (lower- or upper-case) "g", "p", or "s". Have your program print out an error message in this case (and otherwise stop, doing nothing). (2) Each letter could occur more than once, if desired. (Hint: If there are two "g"s in the input it is the same as no "g" at all. Three "g"s is hte same as only one "g". (3) It doesn't matter who goes first, etc. In other words, Ginny first, followed by Sven, results in the same configuration as Sven first followed by Ginny. (Why?)

Submission Instructions

Your submission should consist of a report (a PDF file) and implementation (source code) files. Be certain to include all Java source files (i.e. the ".java" files) needed to run your application.

- Submit one compressed file, using only the "zip" format for compression, that includes all files (report and Java source code) for your submission.
- The report (a PDF file) should consist of

  Requirements: Summary of the above requirements statement in your own words. Do this for each part of the assessment.

  Analysis and Design: A short (one paragraph) description of your analysis of the problem including a Class Diagram outlining the class structure for your proposed solution, and pseudocode for methods. Again, do this for each part of the assessment. Pseudocode need not be a line-by-line description of what you do in each method, but an overview of how methods operate, what parameters they take as input, and what they return. Ideally, pseudocode should be detailed enough to allow me (or someone else) to implement a method in any programming language of their choosing, but not rely on language-specific constructs/instructions.

  For Part I and Part II you must submit your Java file(s). **Do not submit your compiled class files, only the source code!**

  Testing: A set of proposed test cases presented in tabular format including expected output, and evidence of the results of testing (the simplest way of doing this is to cut and paste the result of running your test cases into your report). You should describe carefully how you tested in your code in each part to determine that it is running correctly.
- The implementation should consist of

  Your Java source files, i.e. the relevant .java files, not the class (.class) files.

Upload your files (as a single compressed zip file) to

https://sam.csc.liv.ac.uk/COMP/Submissions.pl

(you will need to log in using your Computer Science username and password).

Submission Deadline

Wednesday, 16 May 2018, 5:00pm (Wednesday of Week 13)

Notes/Penalties

- Because submission is handled electronically, ANY FILE submitted past the deadline will be considered at least ONE DAY late (penalty days include weekends since assessment is performed electronically). Late submissions are subject to the University Policy (see Section 6 of the Code of Practice on Assessment).
- Please make sure your Java classes successfully compile and run on ANY departmental computer system. For this reason, the use of IDEs like NetBeans and Eclipse is discouraged. **If your Java source files do not successfully compile/run, you could suffer a penalty of 5 marks from your grade.** This penalty could be applied for Part I and/or Part II of the assessment, but only once per part. (However, this will not turn a passing grade into a failing grade.)
- If your report is not a PDF file, you will lose 5 marks from your final grade. (However, this will not turn a passing grade into a failing grade.)
- If you use any form of compression other than "zip", you risk your files not being read by the assessors, resulting in a mark of 0! If your files can be read, you will still lose 5 marks from your final grade. (As before, this will not turn a passing grade into a failing grade.)
- **Note this is an individual piece of work!** Please note the University Guidelines on Academic Integrity (see Appendix L of the Code of Practice on Assessment). **You should expect that plagiarism detection software will be run on your submission to compare your work to that of other students.**

Mark Scheme

Marks for each part of the assessment will be awarded for:

- Analysis and Design 10% (This includes things like UML diagrams for your classes/application, justification for how/why you structure your classes, appropriate pseudocode for methods, etc.)
- Implementation 25% (This includes correctness of your programs, appropriate level of comments, correct indentation so your code is readable, having useful identifiers, etc.)
- Testing 10% (Have you done suitable testing in terms of checking different paths through your code, checking what you do with "unexpected" inputs, a sufficient number of tests, etc.?
- Extra questions in Part I or Part II 5%

Please see the module web page for the feedback form.