Programming Applications (PRA), Minor Coursework Exercise 1, "Making a Partnership" (2.5%)

Please read the document marked 'Q&A Lab Projects and Pair Programming' carefully, before attempting any lab project. It contains important information on how to complete each minor piece of coursework. You will be asked to officially declare that you have read this document prior to submission. False declarations are taken seriously by both the department and by the college.

This lab project counts for 2.5% of your mark for PRA minor coursework assessment, and is the first of four.

The release weeks for this assignment start 23rd January, at 23:55, and end 6th February, at 23:55. All submissions must occur before the end of the second release week.

If you have any questions about the structure of this assessment, please follow the 'Additional Support' steps listed on KEATS.

The aims of the first lab project are as follows:

1. To revisit basic object-oriented concepts from 4CCS1PRP;
2. To understand the relationship between the Law of Demeter, coupling and encapsulation;
3. To use packages and your chosen IDE for the first time, if you have not already done so;
4. To allow you to experience pair programming for the first time, if you have no previously undertaken the practice.

In this lab assignment, you and your partner will construct two versions of the same code. The versions will not be incremental, as we have seen previously. Instead, the first version will purposefully violate the Law of Demeter by being tightly coupled. You will then alter this code to produce a second version, which will utilise stricter encapsulation in order to obtain looser coupling, and adhere more closely to Demeter's law. By looking at these versions in parallel, we hope to illustrate more about the implications of Demeter's law, and the concept of coupling.

You must not complete any of these exercises without your chosen partner present.

1 Setting up the packages

Create a new *project* in your chosen IDE. Inside this project, create a new *package* named according to a concatenation of your first name, and the first name of your partner (<partnerApartnerB>). So, if Martin Chapman and Steffen Zschaler were embarking on this exercise, our project would contain a package called *martinsteffen*. The order of names here is unimportant.

Inside this package, create a *subpackage* called *tightlycoupled*. After creating this subpackage its name, in most IDEs, should appear in the package or project view as *<partnerApartnerB> .tightlycoupled*, with *<partnerApartnerB>* naturally replaced with your first name, and that of your partner. If you locate the actual *folder structure* created in your filesystem by your IDE for this project, you should now find a *tightlycoupled* folder inside a *<partnerApartnerB>* folder.

2 Gathering evidence

Inside your *tightlycoupled* package, create a new class called `Evidence`. This class is designed to reflect a piece of evidence collected from, or relating to, a crime scene or a crime. As such, you should ensure that this class can be used to store the `type` of evidence found (e.g. "broken glass"), and you should take appropriate steps to ensure that all instances of the `Evidence` class contain this `type` information. In addition, you should ensure that when any `Evidence` object is printed (i.e. passed as an argument to the `System.out.println` method), the content of the evidence's `type` is printed to the terminal, as opposed to the object's memory identifier. Finally, provide a means of `get`ting the `type` of evidence stored in any instance of the `Evidence` class.

Once you have done this, the following test case should successfully create a new `Evidence` object:

```
Evidence brokenGlass = new Evidence(‘‘broken glass’’);
```

To test that this is indeed the case, create another class, `Main`, within the *<partnerApartnerB>* package. Your package structure should now look something like this:

```
<partnerApartnerB>/
. . . . . . Main.java
. . . . . . tightlycoupled/
. . . . . . . . . . . . Evidence.java
```

If this package structure representation doesn't make sense to you, follow the appropriate channels listed in the 'Additional Support' section of KEATS for further clarification.

`Main` should be directly executable, and into `Main` you should paste the test case given above. **In addition**, in your partner's project specification (which they too should download from the link given on KEATS), they too will have a test case that creates an `Evidence` object, which may contain the same type of evidence, or a different type. Paste your partner's test case into `Main` also, so that `Main` contains both of your test cases, both of which should make brand new `Evidence` objects, that either pertain to two different, or two of the same types of evidence.

Call `System.out.println` with each of your `Evidence` objects so that when `Main` is executed, the two types of evidence that these objects represent are printed to the console in your IDE. If your partner was to have the evidence type "DNA", then the output from your program should look like the following:

```
broken glass
DNA
```

3 Storing evidence

Implement a class called `EvidenceBox`, that will represent a box that contains multiple pieces of evidence collected from a crime scene or relating to a crime. This class should also exist in the *tightlycoupled* package:

```
<partnerApartnerB>/
. . . . . . Main.java
. . . . . . tightlycoupled/
. . . . . . . . . . . . . Evidence.java
. . . . . . . . . . . . . EvidenceBox.java
```

`EvidenceBox` should maintain a record of all the `Evidence` it currently contains. In addition, it should be possible to:

1. `Add` new pieces of `Evidence` to the box.
2. `Extract` the record of *all* the `Evidence` currently in the box.
3. `Display` each `type` of `Evidence` currently in the box on a new line in the console of the IDE. If there are multiple instances of the same type of evidence in a box, then the output line for that type should be `Nx type`, where `N` is the number of occurrences of that type (e.g. `2x broken glass`).

In addition, every `EvidenceBox` should have a `caseNumber` and a `caseName` (written on the side, we might imagine). It should be possible to `get` the case number from an `EvidenceBox`.

Inside `Main`, after the code which creates your test case `Evidence` object, and your partner's, and prints them, create an `EvidenceBox` with the case number 2005000381 and the case name "S. Avery", and then `add` your two `Evidence` objects to it.

Then, `display` all the types of evidence in the `EvidenceBox` you just created, so that when `Main` is now executed the output from your program appears as the following (again assuming your partner has the evidence type "DNA"):

```
broken glass
DNA
1x broken glass
1x DNA
```

So, the types of both test case `Evidence` objects should be printed (as implement in the previous section), and then summarised by adding these objects to an `EvidenceBox` and `display`ing the contents of that box (as implemented in this section).

4 Examining the evidence

Implement a class called `Policeman` (or `Cop,` if you prefer to use the vernacular from across the pond):

```
<partnerApartnerB>/
. . . . . . Main.java
. . . . . . tightlycoupled/
. . . . . . . . . . . . . Evidence.java
. . . . . . . . . . . . . EvidenceBox.java
. . . . . . . . . . . . . Policeman.java
```

Every `Policeman` should have a `name`, and this name should be printed to the console in your IDE whenever a `Policeman` object is printed.

Policemen can have, in their possession at any one time, a collection of `EvidenceBox`es for examination. When a `Policeman` `take`s an `EvidenceBox`, it should be stored in this collection.

Inside `Main`, create two `Policemen` with the names "J. Lenk" and "A. Colborn". Then, make the `Policeman` object that represents J. Lenk `take` the `EvidenceBox` you created in the previous section, with the case number 2005000381 and the case name S. Avery.

Next, we want to give a `Policeman` the ability to `get` *all* pieces of evidence of a certain type from a box in their possession with a certain case number. For example, J. Lenk may wish to find all pieces of DNA evidence in the box with the case number 2005000381. The general procedure required within the `Policeman` class to do this is as follows:

1. Assume the presence of a desired `caseNumber` (e.g. 2005000381) and a desired `type` of evidence (e.g. "DNA").
2. Define a list that will store all the objects from an evidence box that match the desired type.
3. Go through each `EvidenceBox` in the `Policeman`'s possession.
4. Determine whether the case number of the current `EvidenceBox` matches the desired case number. If it does, select this box.
5. `Extract` the record of all pieces of `Evidence` in the selected `EvidenceBox` (this functionality was provided inside `EvidenceBox` in Section 3, Point 2).

6. Examine each piece of `Evidence` in the record extracted from the selected `EvidenceBox` in turn.
7. Check whether the current piece of `Evidence` being examined matches the desired `type` of evidence that is being searched for. If it does, add it to the list of objects that match the desired type.

Implement this procedure, so that inside `Main` you can instruct J. Lenk to `get` all the DNA evidence in S. Avery's case, 2005000381 (there may not be any DNA evidence in your `EvidenceBox` instance, depending on the types of evidence you and your partner have on your coursework specification, but this is ok). To confirm that this evidence has been located, you should print the result of the procedure above (i.e. in this example, the list of `Evidence` objects that are of DNA type in your `EvidenceBox`), with a suitable prefix. Again, this may be a null result if you do not happen to have any DNA evidence in your evidence box.

Your sample output should therefore now look like the following:

```
broken glass
DNA
1x broken glass
1x DNA
Pieces of DNA Evidence in Evidence Box number 2005000381 found  by J. Lenk:
[DNA]
```

5 Starting to visualise things (optional)

At this point, you may like to implement a simple console-based GUI to allow a user to interact with the code you have produced so far. In other words, you might like to extend your program in order to use `Scanner` and `System.in` to read input from a user in order to carry out the tasks you have done so far in `Main` in an ad-hoc and dynamic manner. For example, you might like to offer the user a simple choice whereby they can choose to create a new piece of evidence with a type, a new evidence box with a case number and case name, or to add an existing piece of evidence to an evidence box. Feedback can be given to the user via `System.out`.

Essential: Your console-based code must not interfere with your existing classes. Use `Main` as a model; a class that simply makes and interacts with the objects created, without those objects being dependent upon `Main`'s existence. Separating the 'view' of a program from the 'functionality' was discussed briefly in PRP last semester, and will be discussed in more detail in this module later in the semester.

Console-based GUI task originally provided by Steffen Zschaler and Andrew Coles. Edits and updates by Martin Chapman.

6 Documenting your code

At this stage, most of your implementation for this project is complete, so you can think, if you haven't already, about commenting your code in order to help you understand what's going on (essential for your upcoming lab assessment) and to help other people understand what's going on. Comments should be familiar to you, and take the form of single or multiline statements placed directly into your source code in order to explain the surrounding lines.

Documentation differs slightly from commenting, as it suggests the presence of an external document that complements your program by providing details of the *interface* (meaning the public methods and constructors, in this context) available to someone if they wish to make an object of one of your classes. You have already seen examples of interface documentation when you have used the Java API, which tells you the public methods and constructors available to us when we choose to make objects of the classes provided to us by Oracle (formerly Sun) in the Java libraries.

Java provides a tool called *JavaDoc* to help you generate documentation similar to what you find in the official Java API. JavaDoc does this by detecting special types of specifically placed and formatted comments in your program – known as JavaDoc comments – and then uses these comments as the basis for generating formatted HTML pages. In this section, you are going to experiment with JavaDoc to produce some documentation for the `Evidence`, `EvidenceBox` and `Policeman` classes.

Firstly, make yourself familiar with the format of JavaDoc comments. If you are already comfortable with the principles of writing JavaDoc comments, the following serves as a useful overview of the available tags: `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#javadoctags`. If not, substantial information is available at `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`. The most relevant subsection is probably `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html` (you do not need to read this completely, but you should understand the basic structure of JavaDoc comments).

Secondly, make yourself familiar with the use of the JavaDoc generator (either from the command line or by using the appropriate facility from your IDE) which will use the comments you have written, and the JavaDoc tags you have selected, to produce custom documentation for your program:

- A quick guide for generating JavaDoc in Eclipse is `http://www.java-forums.org/blogs/java-tip/286-creating-javadocs-eclipse.html`.
- Another more detailed but slightly rougher guide is `http://www.eclipse-blog.org/eclipse-ide/generating-javadoc-in-eclipse-ide.html`.

Once comfortable, provide JavaDoc comments for the public methods of `Evidence`, `EvidenceBox` and `Policeman` and generate the JavaDoc documentation from them. Inspect the resulting documentation, play around with the options for generation and with the comments you have written.

Links to documentation originally provided by Steffen Zschaler and Andrew Coles. Edits and updates by Martin Chapman.

7 Decoupling your classes

Go to your `Evidence` class, and find the name of the method that allows you to discern the type of evidence currently stored in any instance of this class. In my solution, this method is called `getEvidenceType`, and I will refer to it as such from now on. Try changing the name of this method. I changed mine to `retrieveEvidenceType`. What happens? If you have followed the instructions above to the letter, you should receive errors in both your `Policeman` and `EvidenceBox` classes. If you do not, because you perhaps deviated from the specification slightly to anticipate this issue, that's fine, read on anyway to confirm your understanding.

For most of you however, these two errors will appear. `EvidenceBox` complains when it tries to `display` all the types of evidence in a box, and `Policeman` complains when searching for all evidence of a particular type in a box. At first these errors may seem unavoidable: a method name has been changed, so all references to it need updating. However, because of the way we have structured our current solution, references to the `getEvidenceType` method need updating in an *unnecessary* number of places.

Let's consider the interaction between `Policeman`, `EvidenceBox` and `Evidence`. To represent a policeman searching for all pieces of evidence of a certain type in a chosen evidence box, the `Policeman` class extracts the record from the `EvidenceBox` class of all the pieces of `Evidence` in that box. `Policeman` then interacts *directly* with each `Evidence` object in that record in order to determine its type (see Section 4 for clarification), which violates Demeter's Law. Remember Demeter's Law tells us that a method M of an object O may only invoke methods of the following kinds of objects:

1. O itself
2. Objects in M's parameters
3. Any objects created/instantiated within M
4. O's direct component objects (objects in a 'has-a' relationship with O)
5. A global object, accessible by O, in the scope of M

Because `Policeman` (O), specifically its method that allows a `Policeman` to get all pieces of evidence of a certain type from a box in their possession with a certain case number (M), interacts with `Evidence` – an object that doesn't meet any of these criteria – we can confirm that this Law is violated. We can think about this in another way: `Policeman` and `EvidenceBox` are 'friends', because `Policeman` contains a collection of `EvidenceBox`es. `EvidenceBox` and `Evidence` are 'friends', because `EvidenceBox` contains a collection of pieces of `Evidence`. Therefore, to `Policeman`, `Evidence` is a 'friend of a friend', so when `Policeman` communicates with `Evidence` (to discern its `type`), this violates the natural interpretation of Demeter's Law: only talk to your friends.

The consequence of violating Demeter's law is *tight coupling*, or an unnecessary interdependency between classes – in this case between `Policeman` and `Evidence`[1] – requiring additional, and unwanted maintenance when small updates are made, as we have seen.

To solve this problem we are going to decouple `Policeman` from `Evidence`. Firstly, if you changed the name of a method in the above paragraph in order to identify the tight coupling (for me, this involved changing `getEvidenceType` to `retrieveEvidenceType`) then change it back so that you no longer have any errors in your program.

[1]It is worth conducting some further reading in order to understand when a large amount of interdependency is actually required.

Next, create a new subpackage below <*partnerApartnerB*>, called *looselycoupled* and copy all your existing classes into this package so that your overall package structure now looks like the following:

```
<partnerApartnerB>/
. . . . . . Main.java
. . . . . . tightlycoupled/
. . . . . . . . . . . . . Evidence.java
. . . . . . . . . . . . . EvidenceBox.java
. . . . . . . . . . . . . Policeman.java
. . . . . . looselycoupled/
. . . . . . . . . . . . . Evidence.java
. . . . . . . . . . . . . EvidenceBox.java
. . . . . . . . . . . . . Policeman.java
```

To decouple `Policeman` from `Evidence`, we can more strictly encapsulate the collection of `Evidence` inside `EvidenceBox`, so that `Policeman` can no longer interact directly with this collection, and thus with the `Evidence` objects contained within it. This involves removing one of the pieces of functionality from `EvidenceBox`, originally listed in Section 3 as '`Extract` the record of *all* the `Evidence` currently in the box.'. In other words, the version of `EvidenceBox` stored under the *looselycoupled* package should be altered so that it is no longer possible to access the list of `Evidence` objects held within it. However, make sure this functionality remains inside the version of `EvidenceBox` in the *tightlycoupled* package, as a record of you completing this section of the coursework.

You should now have an new error in your `Policeman` class, because the ability for a `Policeman` to search for all instances of a particular type of evidence in a particular evidence box (a method I will refer to from now on as `getEvidence`) is currently based on the extraction of the record of all the `Evidence` held within an `EvidenceBox` (Step 5 in the procedure given in Section 4), which is no longer possible. This error is good, however, because it confirms that `Policeman` and `Evidence` can no longer be tightly coupled.

To address this error, and to restore the correct functionality to `getEvidence`, we must first move the search for all instances of a particular type of evidence into `EvidenceBox` itself; the only place where a record of all the evidence in a box is now accessible. To achieve this, add a new method into `EvidenceBox` that, when supplied with a certain type of evidence, returns a list containing all the pieces of evidence inside the box that are of this type. I will refer to this method as `getEvidenceByType`. This should not be a difficult method to implement, as the code to achieve this simply needs to be moved from `getEvidence` in `Policeman` into `getEvidenceByType` in `EvidenceBox`. Review Section 4 if necessary.

Following this, we can update `Policeman`, so that when `getEvidence` is called with a specified case number and a specified evidence type, the evidence box is located in the same way as before (using the specified case number), but rather than extracting the list of evidence from this box and examining it inside `getEvidence` directly (which is no longer possible), `getEvidence` instead calls `getEvidenceByType` inside this `EvidenceBox` to perform this task. To `getEvidenceByType` `getEvidence` should pass the specified evidence type, and `getEvidenceByType` should then in turn determine which pieces of evidence exist in the box of this type, and return a list of these back to `getEvidence` (which it already does). `getEvidence` can then return the list of evidence back from `Policeman`, as it did before. So, to decouple `Policeman` from `Evidence` here is reasonably simple: force `Policeman` to *delegate* the task of locating all pieces of evidence of a certain type to `EvidenceBox`, rather than performing this task itself.

Because you have not altered the interface of any of your classes (i.e. the method names, their parameters and return types) while making these modifications, the test code in `Main` should work whether you choose to use the version of your classes inside the *looselycoupled* or *tightlycoupled* packages. Test this by swapping which version of these classes `Main` refers to. If reference errors occur, try and discern which element of your class interface has been unnecessary altered.

Finally, go back and change the name of `getEvidenceType` in `Evidence` again. Now, you should only have an error inside `EvidenceBox`, where you will need to update any references to `getEvidenceType`. Updates are no longer required inside `Policeman`, because we decoupled it from `Evidence`, and therefore changes in `Evidence` do not affect it directly. On this small scale the difference in work is not noticeable, but on a larger scale effectively managing coupling can dramatically improve the maintainability of your code.

7.1 Submitting your work

If, once finished, you wish to submit your work to us for preliminary feedback, you should do so by following the 'Additional Support' steps from Step 3.2 onwards (i.e. direct contact), in order to avoid sharing your solution with your peers unnecessarily.

If you are happy with your solution, then you both you and your partner should submit identical copies of it to KEATS, adhering to all the submission rules given in the Q&A document referenced at the beginning of this project brief. The folder you compress for submission should be your top-level package, or the first folder inside your project source. For this assignment, the top-level folder is <*partnerApartnerB*>. Compress this folder directly, so that the file you submit is of the form <*partnerApartnerB*>.<compression_extension>

(e.g. martinsteffen.zip). Your compressed folder should therefore contain your complete package structure, detailed in the previous section, and including both copies of the *tightlycoupled* and *looselycoupled* code.