Week 3 Lab

Refactoring and Arrays

Objectives

In this week's lab, you will:

- refactor code to improve its design and readability
- learn to use your IDE's refactoring tools
- use Java arrays
- use loops
- make some design choices
- justify your design choices

You must draw a UML class diagram showing the system that you plan to implement *by the end of the lab*. Note that this system will incorporate work you have done in multiple tasks, so **read the entire lab sheet** carefully.

This diagram should be quick and simple. It only needs to show the classes you plan to create and the relationships between them. You don't need to show fields or methods. You are strongly encouraged to draw this diagram by hand, either on paper or on a whiteboard.

You must get your design in UML approved by your demonstrator before you start coding, and you must get your completed design approved by your demonstrator before you leave. This will mean you can work at home without coding up a poorly-designed solution.

Task 1. Redesign `Lair` to use an array

This week's task uses your complete code from the previous week as a starting point. All following labs will do the same, building on the code from the preceding week. It's a good idea to save a snapshot of your previous week's solution before you start to make changes (you learned how to do this in the Week 1 lab).

As we observed at the end of last week's lab, using three hard-coded attributes is a terrible way for the `Lair` to keep track of its `LairLocation`s. It would be completely unmanageable to have a separate attribute in the `Lair` class for every room in your villain's headquarters. It is also painful to access them all individually, in methods such as `displayRooms()`. A better solution is to use a *collection* of references to `LairLocation` objects that is stored in a single container. The simplest built-in container in Java is the `array`.

Modify the `Lair` class so that instead of having three attributes of type `LairLocation`, it has a single attribute that is an array of `LairLocation`s.[1]

[1] Java offers you several other ways to store a collection of `LairLocation`s. For now, we'll keep it simple by just using an array. We'll implement other ways of doing this in the next few weeks.

Modify the methods `createRooms()` and `displayRooms()` to use the array of `LairLocation`s.

- `createRooms()` must create four new `LairLocation` objects, each with a different name and description, and store references to them in the array.
- `displayRooms()` must display the descriptions for the locations. Get the room's description by calling `getLairLocationDescription()` for each of the `LairLocation` objects in the array. Use a loop rather than repeated code, and avoid hard-coding the array length into `displayRooms()`.[2]

Something to think about: why is it a good idea to keep I/O code (e.g. printing to the console) out of classes that represent domain concepts, such as `LairLocation`?

Task 2. Refactor `LairLocation`'s method names using Eclipse refactoring tools

Last week, you created an `LairLocation` class with a method called `getLairLocationDescription()` that returns a `String` consisting of the location's name and description. This is not really a good name for this method. By convention in Java, methods starting with `get` return the value of an attribute. That's not what this method does. Also, since it is a part of the `LairLocation` class, we really don't need the "LairLocation" part of the method name. We can just call it `description()`.

Making this sort of change using a simple text editor can be a lot of work. We need to change the method name not only in the `LairLocation` class, but also in every client of `LairLocation` that uses that method. We don't want to change any method that has the same name, but belongs to a different class. A global search-and-replace is likely to break things. Fortunately, most modern IDEs, such as Eclipse, provide language-aware refactoring tools to help.

Go to the `LairLocation` class and find the method `getLairLocationDescription()`. Select the text `getLairLocationDescription()` and then right-click on it to bring up a menu. Choose `Refactor` → `Rename...`. Edit the method name to be simply `description()`. The IDE will analyse the code for the system and change the method name everywhere it occurs. Now open the `Lair` class and check that the method name has changed where it is used in the method `displayLairLocations()`.

If you are using an IDE other than Eclipse, work out how to use the refactoring tools in your IDE to make the same changes.

Once you have made the changes, informally test your program to ensure that the functionality is unchanged. Refactoring should not change *what* your program does, only *how* it does it. In a larger project, you would use automated *unit tests* and possibly *system tests* to ensure that your refactoring has not affected any functionality.

More refactors

Notice that although we've called the spaces within our lair `LairLocation`s, the methods inside `Lair` seem to be using the word "Room" to mean the same thing. This kind of difference often crops up as the system evolves during development, and it can be confusing.

Use your IDE's refactoring tool to rename `createRooms()` and displayRooms() to something that is consistent with your class names. Once again, check that this has worked by rebuilding and rerunning your program. If you like, experiment with changing the name of the class – but please change it back when you're finished as other lab exercises will depend on it.

Task 3. Create a `Minion` class

Every evil genius needs a support staff of minions[3] and henchmen to operate traps, put evil plots into action, and clean up the mess after the good guys have visited. We need to add a `Minion` class to our system.

[2] Hint: use your IDE to investigate what properties an `array` has. What queries can it respond to?
[3] https://tvtropes.org/pmwiki/pmwiki.php/Main/Mooks

The `Minion` class needs to have private attributes to store:

- the minion's ID (`minionId`)
- the minion's given name (`givenName`)
- the minion's (`familyName`)

Note that we have not specified the type of `minionId`. You can choose to use a numeric integer type, or a Java string.[4] This kind of decision is actually a design decision. In a comment, briefly list the advantages and disadvantages of using both, and justify your final decision. Something like "I don't know how to use Java integers very well" is not in itself a sufficient justification.

Additionally, add a comment explaining why we should use attribute names `givenName` and `familyName` rather than `firstName` and `lastName`.

Implement two constructors for `Minion`, with these signatures:

- `Minion(*type* newMinionId)`
- `Minion(*type* newMinionId, String newGivenName, String newFamilyName)`

Replace `*type*` with the name of the type you have chosen for the attribute `minionId`.

These constructors provide two different ways of creating and initialising a `Minion` object.

Next, implement two *mutators* (methods that change attribute values):

- `setGivenName(...)`
- `setFamilyName(...)`

These should both take `String` arguments.[5]

Finally, implement an accessor method called `description()` that returns a `String` containing the minion's ID, given name, and family name concatenated together, in that order and separated by spaces.

Task 4. Assigning Minions to Locations

The purpose of our software is to keep track of where the villain's assets go, including both personnel and equipment. That means we will need to know the locations of our minions. Modify the `LairLocation` class to support this as follows:

- Add an attribute called `minions` to `LairLocation` to store a collection of `Minion` objects. You can use an array or another Java collection type if you prefer.
- Add a method `assignMinion(Minion minion)` to `LairLocation` to assign a minion to a location. It must add the `Minion` to the `minions` collection.
- In the `Lair` class, add a method that creates some `Minion` objects and assigns them to one or more locations. In `printStatus()`, this must be called after `createLairLocations()`. Make sure you include code that tests both of the `Minion` constructors as well as both the mutators.
- Modify `displayLairLocations()` so that, as well as displaying the description for each `LairLocation`, it also displays a list of the minions assigned to each location.

[4] Actually, there are several other possible representations, but you need only consider these two.

[5] Most modern IDEs have built-in tools to automatically generate "getters" and "setters". Eclipse does. Feel free to find it and use it. Try the Source menu.

You will need to decide how to access the minions assigned to an `LairLocation` from the `Lair` class. There should be no I/O in any class other than `Lair`, and the mechanism should not allow the `minions` collection stored in `LairLocation` to be modified by any other class[6].

The output should look something like this:

```
Welcome to the Supervillain's Lair Management System.

Shark Tank: Full of sharks with lasers on their heads
Assigned Minions:
12345678 Mini Me
12345679 Chum Berley

Boardroom: Where we plot world domination
Assigned Minions:
12345678 Mini Me
12345680 Donna Matrix

Janitorial: Cleans up the mess
Assigned Minions:
12345666 Domestos McBleach
12103464 Pyne O'Kleen

Good-bye. Thanks you for using the Supervillain's Lair Management System.
```

Getting marked

Once you have completed these tasks, call your lab demonstrator to be marked. Your demonstrator will ask you to do a "walkthrough" of your program, in which you explain what each part does. Note that to receive full marks, **it is not sufficient merely to produce correct output**. Your solution must implement all structures specified above *exactly* — as software engineers we must follow specifications precisely. If you have not done so, your demonstrator will ask you to fix your program so that it does (if you want to get full marks).

If you do not get this finished by the end of this week's lab, you can complete it during the following week and get it marked at the start of your next lab — but make sure that your demonstrator has at least given you feedback on your design before you leave, so that you don't waste time implementing a poor or incorrect design. That will be the last chance to get marks for the exercise. You cannot get marked for an exercise more than one week after the lab for which it was set.

Marking

This lab is worth 2% of your final mark for FIT2099. Marks will be granted as follows:

- 0 marks if you do not attempt the tasks, or if you could only complete them once given the solution
- 1 mark if some tasks are incomplete, if the design or the implementation is flawed, or if you needed to see a significant part of the solution to complete them
- 2 marks if you were able to complete all tasks independently, with good design and correct implementation

[6] It is not enough to rely in the fact that *your code* doesn't modify it. You need to show that code in other classes, possibly written by less-competent programmers, won't *be able* to modify it.