Entry Four*:* Object-Oriented Principles

Student Name: Gregory Sloggett

Student Number: 14522247

Work Done:

This practical involved three programming tasks. The first task was to identify the violation of the open closed principle in the OCP.java class. This class involved initialising a postage stamp in the shape of a square but needed to allow for circular and other shaped stamps without needing to do change the postage stamp class itself. For this to happen the postage stamp class needed to take in a shape object (that could be any shape), rather than a specific shape (like a square which is how the class was originally coded). By making an abstract Shape class and adding an abstract method to it, we can use that abstract class to pass into the Postage Stamp class.

```
19 abstract class Shape {
20     public abstract String toString();
21 }
```

Then we create shape objects, such as square and circle, which must implement everything from the shape class so that it conforms and can be used by the postage stamp class. This means we can add any shape type to the postage stamp class, without making any changes to the postage stamp class.

The second task involved the single responsibility principle and ensuring the SRP.java file conforms to that principle. The alien, 'Zorg', witnesses a human walking their dog, and perceives that duo to be one mammal, a 'hexapod'. This makes it unclear as to what part of the hexapod performs what action (method). We have four methods, walk, throwStick, fetchStick and bark. It is straightforward to interpret what action the human and dog perform here given that we understand the domain, however the alien is unfamiliar, and therefore it interprets them together. We need to separate these methods from the hexapod class as they shouldn't be modelled together like they are. As such, we specify two separate classes, a human class and a dog class.

As the 'name' variable is used in the throwStick method, we know that the human takes the name variable, and as such this is taken in the human class constructor. The human class contains simply the methods for throwStick and walk. The dog class does not require a constructor and only contains the bark and fetchStick class. By separating up the hexapod class into a human and dog class it makes the code clearer and it reduces the ambiguity over new requirements that may need to be added to the system, such as a run method.

Finally, the third task involved the Law of Demeter. We are expected to understand the class and pick out the violation of the Law of Demeter in this instance. The sketch coded here shows a customer making a purchase from the shopkeeper. Three objects make up the sketch, Customer, ShopKeeper and Wallet. From the outset the interaction looked clean and straightforward, but upon closer inspection there are some steps that violate the Law of Demeter. The ShopKeeper class is too imposing on the other objects in the system, in other words, it's too closely coupled. What I mean here, is that the shopkeeper charges the customer for the product, but rather than the customer class handling the checking, withdrawing and payment of the money, this task is performed by the ShopKeeper class. This violates the Law of Demeter and required change.

The payment of money needs to be performed by the customer, upon probing or request from the shopkeeper. So, in my new Demeter.java class, the shopkeeper checks with the customer to see if

they have enough money. If this is true, then the customer pays that money, as probed by the shopkeeper (as represented figure 2).

```
if(cust.hasEnoughMoney(amount) == true) {
    cust.payMoney(amount);
}
```

This change involved the addition of two new methods in the Customer class, where the user checks they have sufficient funds to make the purchase (hasEnoughMoney), and where they physically make the transaction (payMoney).

The key aspect of this task is that the shopkeeper should not withdraw the customers money from their wallet, as this is not how it works, but rather the customer should be the only party that interacts with the wallet.

Reflection:

This assignment highlighted some of the various object-oriented principles. These tasks show these principles in their most simplistic form, and as such, their affect on the system is not that prominent. However, as Mel noted, it is in much larger, far more complex systems, where these principles must be adhered to, and remain intact. Otherwise, the system can become very tricky to build on, especially if the dependencies between classes are too much. I think it's important to remember as students, when we're working with these principles, that they're real importance does become serious and of major importance the larger the system gets. I think it is very easy to dismiss these concepts in such fundamental problems and coding tasks, without considering the effects of how a bigger system might influence things.

Relating back to the assignment, one of the fundamental principles Mel has taught us (in this module but also in his third-year module, object-oriented programming), was the concept of loosely coupled but tightly cohesive code. I feel as though it is this concept which many of the principles are conforming to in these tasks I have just implemented in this assignment. Cohesion refers to the degree to which the elements inside a module belong together, or the strength of a relationship between classes and their methods. Coupling is a measure of how closely connected two modules are, or the strength of the relationship between those two modules. Essentially, you don't want two modules closely connected, but you want the classes and methods to be tightly knitted together performing a single task and maintaining a streamlined focus.

The Single Responsibility Principle does showcase this focus in the example I completed in the practical. The class presented to us, as perceived by the alien, showed a Hexapod class that was a misrepresentation of what was seen. It also showed a class that was not loosely coupled, as the responsibilities of the class covered numerous bodies. I changed the class to represent the human and dog separately on that occasion, and thus produced cleaner, and looser coupled classes compared to the hexapod class.

The Law of Demeter also attributes to this focus, but rather appeals to higher cohesion. The shopkeeper should not be performing the role of the customer, and vice versa. You wouldn't go into a shop and give your wallet or purse to the shopkeeper and tell them to take the money out, which is how the code was presented to us on this occasion. You would withdraw the money, and you and only you. The module that was represented to us reflected a loosely cohesive class. It represented a shopkeeper class that was heavily dependent on the other classes. To remove these dependencies,

we restructured the classes to make each class conform to their own responsibilities. This meant that the customer withdrew and paid for the products with their own actions. The class became more cohesive.

From my understanding of the open/closed principle, it essentially means that the code that is written should be written in a way that does not require changing when added new functionality, or at least minimum changes. The reality of this is that it is near to prevent previous code from being changed later down the line given new and updating requirements to the system. Different demands may lead to different functionality and different method outputs needed. I think the idea is that as much as possible this change should be kept to a minimum. This doesn't necessarily attribute to the cohesion and coupling principle I mentioned earlier, but it is no less important as an object-oriented principle.

Overall, this assignment reinforced and harnessed my knowledge and awareness on some object-oriented principles and the importance of loose coupling and tight cohesion. Every practical by Mel I always feel as though I learn something positive in the way in which I code, and this was no different. To be able to work in a team, with a large code base, and develop clean, concise, loosely coupled and tightly cohesive code is important, and these concepts certainly help in that regard.