# COMP47480 Practical 4: Object-Oriented Principles

Sukrat Kashyap (14200092)

3 April 2018

## 1 Work Done

The first task was to find the violation of the open-closed principle of PostageStamp and Square Stamp such that PostageStamp is closed for client change which is the Square and open for extensions. The violation found in the OCP.java was that the PostageStamp was tightly coupled with the Square class. This tight coupling closed the PostageStamp class for further extensions as changing the shape of stamp requires changing the PostageStamp class itself. To fix the issue, an interface named Shape is created which becomes the argument and shape of stamp for the PostageStamp. Now, other classes which can act as a shape to the PostageStamp class must implement the shape interface and can be passed without changing the PostageStamp and is extensible.

The second task requires finding the violation of Single responsibility principle. We know that the Hexapod is actually a dog and a human. Hence, we created two classes namely Dog and Human which dealt with task related to themselves only. These two objects were then created and combined by the Hexapod.

The third task, wanted us to find and correct the violation of the law of Demeter. We found that Customer was depended on Wallet. The Shopkeeper was depended on the Customer. But the Shopkeeper was accessing the Wallet class on which it was not directly dependent and hence violating the law of Demeter. To fix the issue, 2 new methods corresponding to the usage in Shopkeeper class were created in Customer and used in Shopkeeper class making dependency of Shopkeeper only up till Customer.

## 2 Reflections

The co-author of the Agile Manifesto Robert Cecil Martin is promoter of SOLID design principles. SOLID is mnemonic for five design principles for object oriented programming. It is intended to make software designs more understandable, flexible and maintainable.

The five design principles are as follows:

- Single responsibility principle: A class should have only a single responsibility. The responsibility should be entirely encapsulated by the class.

- Open-closed principle: Software entities must be open for extension but closed for modification.
- Liskov substitution principle: Objects in a program should be replaceable with the instances of their subtypes without altering the correctness of that program.
- Interface segregation principle: client-specific interfaces are better than one general-purpose interface.
- Dependency inversion principle: objects and specification should depend on abstractions and not on the concretions.

There were 2 other principles that were shown in the class. They were "No concrete superclasses" and "Law of demeter". No concrete superclasses recommends that all superclasses in a system be abstract and Law of demeter states that each unit should have only limited knowledge about other units that is only units closely related to current unit. It also states that each unit should only talk to its friends and not to strangers and can talk to their immediate friends.

In this, practical we worked on three design principles out of the 7 principles stated above namely: Single responsibility principle, Open-closed principle and law of demeter.

First we worked on Open-closed principle. I felt Open-closed principle extensively uses the concept of polymorphism. Polymorphism is an object oriented abililty to process differently depending on their data type or class. Open-closed principle uses this nature of OOPs to allow open part of the principle. The open nature refers to the open to extensions. Using polymorphism we are able to extend our code by creating abstract class or interface which then can be derived by many different concrete classes doing different things but coming under one name. Now to use this extension we need a module that uses the abstract class instead of the concrete class. This makes the module closed which refers to change in implementation should not affect the module using it. According to me, open-closed principle is the most widely used principle as it is easy to implement. The only part that needs thinking is which class requires extension or which class can have many forms and which class will remain the same and use only certain parts of the form. This principle is usually used closely with IoC (inversion of control) where the objects are created by an outside entity. All the concrete classes implement certain forms which are configured in the IoC container. Making the whole application closed whereas open for extension at the same time as the another concrete class can be created and replaced with the old concrete class by changing the configuration of the container.

Second we worked on Single responsibility principle. One of the hardest principle to bring into practice, according to me. This principle uses the encapsulation nature of OOPs by stating that each module or class should have responsibility over a single functionality provided by the software. This principle is best applied in places where a

number of small modules are required which have almost no connection or dependency between each other. The hardest task of this principle is to isolate the different mutually exclusive responsibilities becuase software is mostly an intertwined mesh. Single responsibility principle is applied on inter linking modules as well the only condition is that the services provided should be narrowly aligned with the responsibility. The problem in this case is to find those isolated narrowly aligned responsibilities. Dividing responsibility can sometimes lead to too many classes and modules which later becomes overhead to manage. Even though this division actually makes the softare more robust and makes the testing easier.

Third we worked on Law of demeter, this law constricts the usage of non-close objects. It basically restricts unnecessary dependency that usually a software easily has due to easy access of those unnecessary dependency from real dependencies. I feel it helps in making more maintainable and adaptable software as due to this restriction unnecessary dependencies are blocked. Due to this blocking, changes made in one object require to change to code to only classes which directly depends on it. Other wise, one small change in one class would lead to changes in several classes that the module does not even depend on properly. I feel even though it does welcome changes, but it increase the amount of wrapper methods to propagate calls to non-dependent components which leads to wider interfaces.