George Suarez

CSE 320

November 14, 2017

Lab 8: Object-Oriented Design

SRP.java

The Single Responsibility Principle says that every class should be doing only one thing, and that thing only. That means that a class should only know its own implementation rather exposing another class's implementation. In my example, I have a class named *Student* that has a name, an id, and a grade. The grade is an object from a class named *Grade* which handles the validation of the grade. This is desirable because it does not make sense for a student to be able to validate their own grade, so the implementation of the grade object is hidden from the student.

OpenClose.java

The Open and Closed Principle says that classes should be open for extension, but closed for modification. In other words, a class can extend its' behavior without changing its own implementation. In my example, I have an abstract class called *Car* which contains an abstract method called *startCar*. Then, I have a class named *Vehicle* that has a method called *startVehicle* that takes in a Car object where the car object is calling *startCar* since they both have the same behavior, and it is not modifying the source code of *Car* which is desirable. Now we can create different types of *Car* that share the same behavior like for example a sports car, and so on.

Liskov.java

The Liskov Substitution Principle basically says that methods that uses references to the base class must be able to use the objects of the derived classes without knowing it. In my example, I have an interface called *IPerson* which contains a method called *eat*. Then I created two classes called *BlindPerson* and *NonBlindPerson* which both derive from *IPerson,* but the *NonBlindPerson* has a method called *see* while the *BlindPerson* does not. This is desirable since it would not make sense for the *BlindPerson* object to also have the *see* method, so we abstracted it away from it without it knowing it which follows the Liskov Substitution Principle.

ISP.java

The Interface Segregation Principle states that a user should not implement an interface if the user does not have to use it. Let say we have an interface called *ITransaction* that extends three classes called *Depositable, Withdrawable, and Transferable*. Then we have three other interfaces called *IDepositable, IWithdrawable,* and *ITransferable* which contains one or two methods that

relates to them. Then we can create classes that can implement any of those interfaces to it. For example, we can have a class called *ATM* which can do deposits and withdrawals, so we can implement *IDepositable* and *IWithdrawlable*. We can also create a class called *Bank* which can do deposits, withdrawals, and transfer funds, so we can implement all three interfaces to that class. This is desirable because it takes in consideration for all possible uses instead of just one general purpose one.

DIP.java

The Dependency Inversion Principle basically says that we should depend on abstractions, but do not depend on the details of how it came to be. For example, we have an interface called *ITransaction* that contains a method called *transact*. Then we can create two classes called *ATM* and *Bank* where *ITransaction* is implemented to both classes. Now we can create a class called *Money* where it has a *ITransaction* construct added which means that the *Money* class does not need to make any changes to add an *ATM* or *Bank* construct since *ITransaction* is an abstract layer to those classes which is what we want.