Adnar Lozano
CSE 320
Lab #8
11/12/17

Lab #8

The following are my explanations for the SOLID principle in Object-Oriented Design.

Single Responsibility Principle

This principle says that the class should only have a single purpose, that it should only be responsible for a single change of the functionality of the program.
This implementation is made up of two classes, Check_Even and Number. The Check_Even class has only one method or responsibility, which is to check if a number is even or not.
The Number class has the ability to check if a number is even by accessing the Check_Even mehod (is_Even). So the Check_Even class does only one thing, checks whether a number of type Int is even or not and return true or false respectively.

Open/Closed Principle

This principle says that classes should be open for extension but closed for modifications. This basically says that a particular class should be able to be extended (in order to add more features) but prohibited to be modified (changing the implementation code of the original class).
In this example, the Shopping_Cart class is accessing the buy method from the Item class without any modifications. The Shoes class extends Item so that it can be used for the Shopping_Cart class without modifying it.
This is a very important principle because it can help programmers with organization and safety. It can prevent certain classes to be mutable by other programmers, thus allowing the program to behave the way it was intended it to be.

Liskov Substitution Principle

This principle says that a derived class extends a base class. Also, that for some type T, all subtypes of T must be able to be substituted in place of T. Basically, we should be able to substitute the parent class with the child class.
Here, both the Fire and Burglar implement the Alarm class and its method. This allows the sound_The_Alarm method to substitute either Fire or Burglar alarm. This principle is also useful for interfaces that are implemented by classes.
This principle goes hand in hand with the Open/Closed principle due to allowing extensions but denying modifications.

Interface Segregation Principle

This principle says that users of an interface should not depend on interfaces they don't use. In this example, we have four interfaces: Device, ICall, IText, and IGetEmail. Two classes implement these interfaces, the Cellphone and the Telephone class. We can see that the Cellphone class implements the ICall, IText, and the IGetEmail interface while the Telephone class only implements the ICall interface.
This principle is very helpful because it allows roles to be split up into different implementation of different interfaces. Therefore, users have to implement functionality only to the features they use. We can use various types of interfaces to create a particular or very specific type of class. The principle is in the name itself, it segregates interfaces to particular classes.

Dependency Inversion Principle

This principle says that high level modules should not depend on low level ones, but that both of them should depend on abstractions. All of the details should depend on abstractions. Basically, the user should create different modules as separate entities that are then combined into a full program. This process has to go through a few layers of abstraction in order for them to talk to each other without dependency of one another, only on the abstractions. The NewDevice class does not depend on the NewCellPhone or the NewTelePhone class because it doesn't have access to their make_Call implementation. It simply calls their make_Call method without knowing all the details. This principle can enable programmers to write flexible code with abstract dependencies that other programmers can later modify. It's always the details that are dependent on abstraction.