National University of San Agustin
System Engineer School

TO - Lab 08 - Solid Principles: Dependency Inversion

Christian E. Portugal-Zambrano

November 20, 2018

1 Solid Principles

S.O.L.I.D is an acronym for the first five object-oriented design(OOD) **principles** by Robert C. Martin, popularly known as Uncle Bob.
These principles, when combined together, make it easy for a programmer to develop software that are easy to maintain and extend. They also make it easy for developers to avoid code smells, easily refactor code, and are also a part of the agile or adaptive software development. This principles are:

- Single Responsibility.
- Open-Closed.
- Liskov Substitution.
- Interface Segregation.
- Dependency Inversion.

2 Dependency inversion

The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions [1]

Every change to an abstract interface corresponds to a change to its concrete implementations. Conversely, changes to concrete implementations do not always, or even usually, require changes to the interfaces that they implement. Therefore interfaces are less volatile than implementations to achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other. Some author may say that if you consequently apply the Open/Closed Principle and the Liskov Substitution Principle to your code, it will also follow the Dependency Inversion Principle [1]. This principle promote the use of abstraction as parameters of constructors or methods (dependency injection. Let's see an example of how dependency inversion works.
In the code below we show the implementation of two concrete class CList and SList.

```
public class CList{
   public CList(){
      //building the list
   }

   public void insert(){
      //inserting into the Clist
   }

   public int get(int pos){
      return 5;
   }
   public int size(){
      return 3;
   }
}
```

```
public class SList{
   public SList(){
      //building the list
   }

   public void insert(){
      //inserting into the Slist
   }

   public int get(int pos){
      return 4;
   }

   public int size(){
      return 5;
   }
}
```

Is normal to do such thing, the problem rise when we try to make a service from this code, so we show the GraphViz class to implement a module to show the CList and SList with GraphVizCode. Observe that this class is a concrete class too and its method print only can receive SList objects.

[1] https://stackify.com/dependency-inversion-principle/

```
public class ListGraphVizPrinter{
    ListGraphVizPrinter(){

    }

    public void print(SList list){
        //add code for GraphViz
        System.out.println("Printing SList with ListGraphVizPrinter");
    }
}
```

What about the main program? We can observe that if we create two objects from different classes, we can use the same class to performe the service of printing with graphviz, the only way will be creating another class for print CList. So in this code we are violating Liskov substitution principle and Open-Closed principle because while this code works there is not any interface that support client need and code reutilization, if we change the concrete class implementation also other dependent class too,The implication, then, is that stable software architectures are those that avoid depending on volatile concretions, and that favor the use of stable abstract interfaces [1].

```
public class NoDip {
    public static void main(String [] args){
        CList myCList = new CList();
        SList mySList = new SList();
        ListGraphVizPrinter printer = new ListGraphVizPrinter();
        printer.print(mySList);
    }
}
```

Ok, let's do it better, next we are trying to implement the same problem but using abstraction, the code below show our first interface

```
public interface List{
    void insert();
    int get(int _pos);
    int size();
}
```

then our first concrete classes (sub-classes)

```
public class CList implements List{
    public CList(){
        //building the list
    }

    @Override
    public void insert(){
        //inserting into the Clist
    }

    //this method return the element at index pos inside the ListSR
    @Override
```

```
    public int get(int pos){
        return 5;
    }

    @Override
    public int size(){
        return 3;
    }
}
```

```
public class SList implements List{
    public SList(){
        //building the list
    }

    @Override
    public void insert(){
        //inserting into the Slist
    }

    //this method return the element at index pos inside the ListSR
    @Override
    public int get(int pos){
        return 4;
    }

    @Override
    public int size(){
        return 5;
    }
}
```

Both of the concrete classes implements the interface List, but now we need to trust on abstractions inside the next interface:

```
public interface ListPrinter{
    void print(List _list);
}
```

This way we are creating a service for print any member of the List family, so the code for the applications looks this way:

```
public class DIP {
    public static void main(String [] args){
        SList mySList = new SList();
        CList myCList = new CList();
        ListPrinter printer = new ListGraphVizPrinter();
        printer.print(myCList);
        printer.print(mySList);
    }
}
```

The last code raise up a question, which object is used inside the print method if we only work with abstractions? Ok, we can do it using abstract factory pattern, **instanceof**

keyword or think twice if is really necessary to know which object is sent, the next code show the use of **instanceof** keyword.

```
public class ListGraphVizPrinter implements ListPrinter{
   ListGraphVizPrinter(){

   }

   public void print(List list){
      //add code for GraphViz
      if (list instanceof CList ){
         System.out.println("ListGraphVizPrinter for CList");
      }
      if ( list instanceof SList){
         System.out.println("ListGraphVizPrinter for SList");
      }
   }
}
```

3 Warm up

Find another way to split the implementation of CList, SList and DList without using **instanceof** keyword.

4 To do

For this point all the data structure framework code developed through the course must be finished including Printers, Writers and Readers, remember to use all of the principles studied until now.

5 Deadline

This practice will be qualified in the next meeting for each group. Remember that plagiarism will be punished. All questions must be done to email or meeting hours (see schedule).

References

[1] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall Press, 2017.