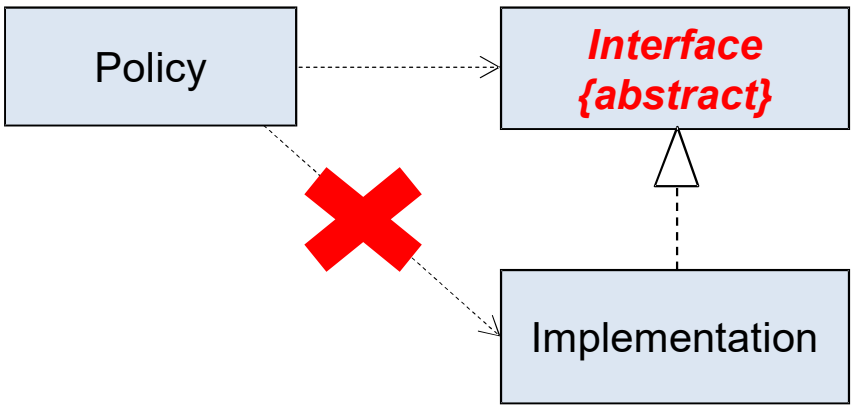
DESIGN PRINCIPLE-BASED REFACTORING: DIP

DIP: The Dependency Inversion Principle

High level modules should not depend upon low level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstraction.



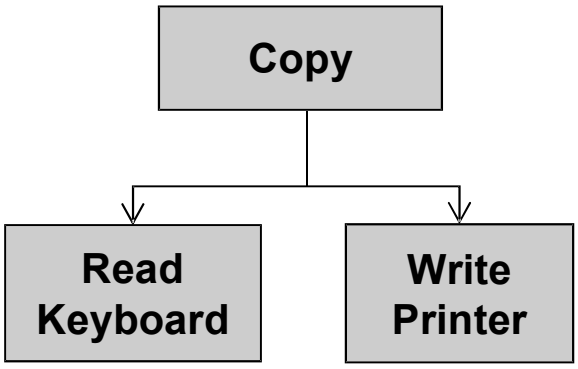
DIP Example

Dependency is the key obstacle to maintaining software.

Copy Program

– copy characters typed on a keyboard to a printer

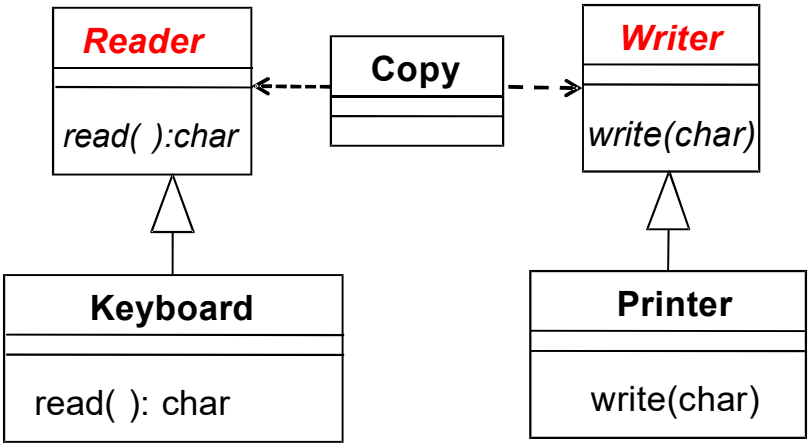
```
void Copy() {
  int c;
  while ( ( c = ReadKeyboard() ) != EOF )
    WritePrinter(c);
}
```



DIP - Solution

Introduce an abstraction for communicating Copy and its low-level modules and communicate only with it.

Now, **Copy** no longer depends on the details that it controls.




```
void Copy (
  char(*Reader_read)(),
  void(*Writer_write)(char) )
{
  int c;
  while ((c = Reader_read()) != EOF)
      Writer_write(c);
}
```

DIP – Practice

```
void printMinMax(int values[], int size) {
   for ( int i = 0 ; i < size-1 ; i ++ )
      for ( int j = i+1; j < size ; j ++ )
         if ( values[j] > values[i] ) {
            int temp = values[i] ;
            values[i] = values[j] ;
            values[j] = temp ;
         }
   printf("Min: %d, Max: %d\n", values[size-1], values[0]) ;
}
```


Refactoring Techniques

Smell	Refactoring
Inappropriate intimacy	Encapsulate collection
Dynamic type casting	Revise inheritance hierarchy
Switch statements	Strategy Pattern
	Command Pattern
	State Pattern

Encapsulate Collection

A method returns a collection

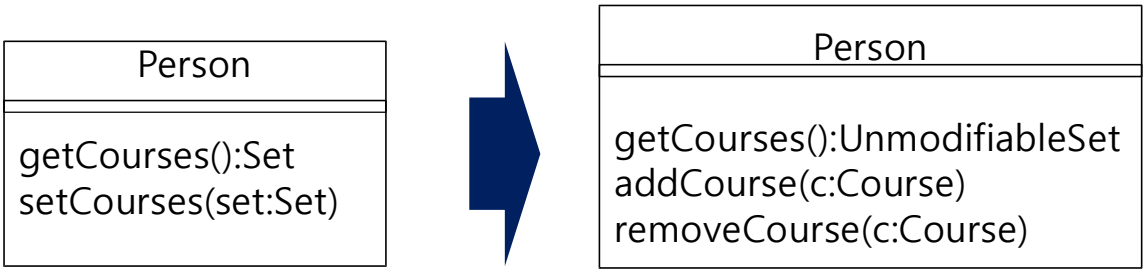
It reveals too much to clients about the object's internal data structures

```
Person kent = new Person();
Set s = new HashSet();
s.add(new Course ("Smalltalk Programming", false));
s.add(new Course ("Appreciating Single Malts", true));
kent.setCourses(s);
Course refact = new Course ("Refactoring", true);
kent.getCourses().add(refact);
kent.getCourses().add(new Course ("Brutal Sarcasm",
false));
kent.getCourses().remove(refact);
```

Also violate Law of Demeter

Encapsulate Collection

Make it return a read-only view and provide add/remove methods




```
Person kent = new Person();
kent.addCourse(new Course ("Smalltalk Programming", false));
kent.addCourse(new Course ("Appreciating Single Malts", true));
Course refact = new Course ("Refactoring", true);
kent.addCourse(refact);
kent.addCourse(new Course ("Brutal Sarcasm", false));
kent.removeCourse(refact);
```

LSP: The Liskov Substitution Principle

Derived classes must be usable through the base class interface without the need for the user to know the difference

- Barbara Liskov, 1988 -

Subtypes must be substitutable for their base types (super types).

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Liskov Substitution Principle(LSP)

`drawShape()` must know every subclass of Shape. ➔ violates LSP.

```
void drawShape(Shape& s) {
  if (typeid(s) == typeid(Square))
    dynamic_cast<Square&>(s)->drawSquare() ;
  else if (typeid(s) == typeid(Circle))
    dynamic_cast<Circle&>(s)->drawCircle() ;
}
```

In addition, `drawShape()` must be changed whenever new derivatives of the **Shape** class are added. ➔ violate OCP.

Violation of LSP leads to violation of OCP.

DIP-related Patterns

- Strategy Pattern
- Command Pattern
- State Pattern