



Agile design and design principles

Dr. Vu Thi Huong Giang



Covered topics: SOLID

- Single-responsibility principle (SRP):
 - There should never be more than one reason for a class to change
- Open-close principle (OCP):
 - Software entities should be open for extension but closed for modification.
- Liskov substitution principle (LSP):
 - Derived classes must be usable through the base-class interface without the need for the user to know the difference.
- Interface-segregation principle (ISP):
 - Many client-specific interfaces are better than one general-purpose interface.
- Dependency-inversion principle (DIP):
 - Details should depend upon abstractions; Abstractions should not depend upon details.



Objectives

- After this lesson, students will be able to:
 - Demonstrate the use of OO design principles for rapid development of softwares

1. Single-responsibility principle (SRP)



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should



1. Single-responsibility principle (SRP)

- Single Reason to Change Principle
 - A class should only have one reason to change.
 - Consequently most changes will affect a small proportion of classes.
- Closely related to Information Hiding
- Responsibility is considered from the service provider point of views



Example

- Sorter, a class that is used to sort an array
- Several potential axes of change that are implicit in this responsibility:
 - What array is to be sorted?
 - What order is to be used?
 - Is the sort stable?
 - What algorithm is used?



Refactoring

- “Refactoring” means improving the internal design without changing the external behavior
 - When a change is required that was not anticipated, we should identify a new axis of change
 - When a new axis of change is discovered, we should “refactor” first and then change
- Poor software engineers make software more brittle when they change it
- Good software engineers improve the flexibility of software when they change it



Attention!

Don't overdesign

- We can (and should) anticipate likely reasons to change.
- But: We should not make them up.
- There is no point protecting the design against classes change that are likely not to occur.

Don't underdesign

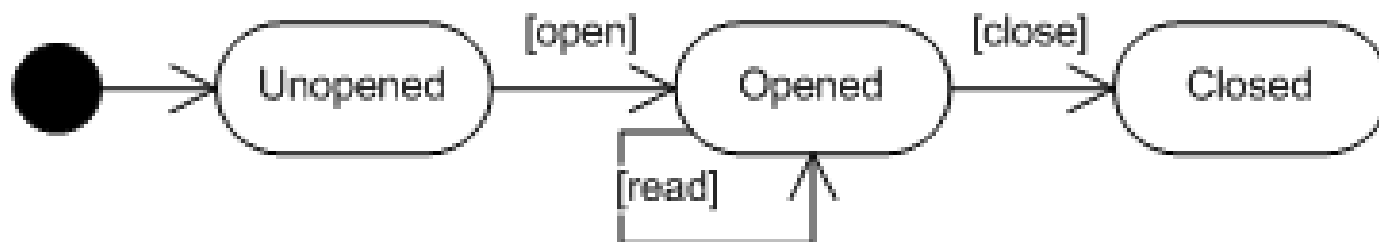
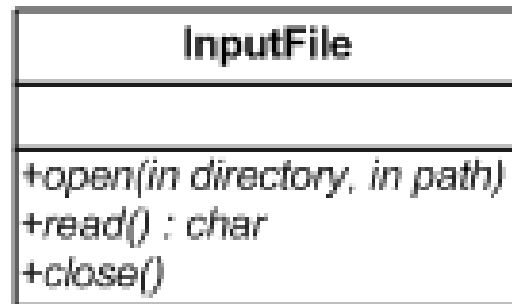
- One principle of agile design is "Do the simplest thing that could possibly work." (From Beck and Cunningham.)
- However the simplest thing is often unnecessarily brittle.
- It is very good to ask the question: "What's the simplest thing that could possibly work", but you should avoid building in brittleness

SRP in action

Example: Process some information that comes from a local file.

→ We need to be able to open files, close files, read from files.

```
InputFile f = new InputFile() ;  
f.open(directoryPath, relativePath) ;  
process(f) ;  
f.close() ;
```





SRP in action

- What might change ?



A change

- What will happen if the customer says: “We need to be able to process data from the web” ?



SRP in action: how to refactor ?

- We should refactor so that this unanticipated change becomes an *anticipated change*.



SRP in action

- Now add the new functionality
- Now each class represent a commitment to a point on a single axis of change

2. Open/Closed Principle (OCP)




OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat



2. Open/Closed Principle (OCP)

- Software entities (classes, modules, functions etc) should be open for extension but closed for modification
 - The most straight-forward way to extend software is often to modify it. This often introduces brittleness and does not promote reusability.
 - Instead we should design software entities so that future changes on the same axis require no modification.
- 



OCP in action

- Suppose that we need to sort a table representing email messages by time

```
class Sorter {  
    public void sort( TableEntry [] a ) {  
        for( int i = 0 ; i < a.length-1 ; ++i ) {  
            int j = i ; TableEntry min = a[j] ;  
            for( int k = i+1 ; k < a.length() ; ++k ) {  
                if( min.getTime().compareTo(a[k].getTime() ) > 0 ) {  
                    j = k ;  
                    min = a[j] ;  
                }  
            }  
            a[j] = a[i] ;  
            a[i] = min ;  
        }  
    }  
}
```




A change: sort by time or
fromAddress



Refactor: how to make a better plan?

- Factoring out the comparison into another class



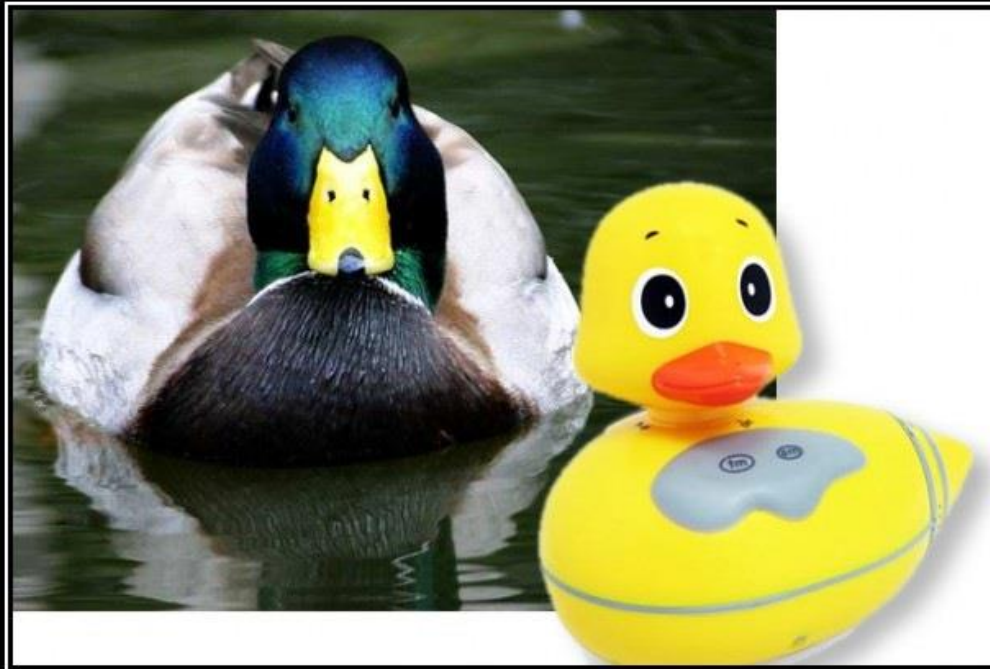
Extend: how to extend the call options ?



An Open and Closed Case

- The Sorter class is now Open/Closed with respect to the axis of “comparison method”
- Closed: We should never have to modify it to accommodate other methods of comparison
- Open: It can be extended with new comparison methods

3. The Liskov Substitution Principle (LSP)




LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction



3. The Liskov Substitution Principle (LSP)

- If S is a subtype of T , objects of type S should behave as objects of type T are expected to behave, if they are treated as objects of type T
 - Note: LSP is all about expected behaviour of objects. One can only follow the LSP if one is clear about what the expected behaviour of objects is.
- 



Recall: Subtypes and instances

- For Java
 - S is a **subtype of T** if
 - S is T,
 - S implements or extends T, or
 - S implements or extends a type that implements or extends T, and so on
 - S is a **direct subtype of T** if
 - S extends or implements T
- An object is a **direct instance of a type T**
 - If it is created by a “new T()” expression
- An object is an **instance of T**
 - if it is a direct instance of a subtype of T.



Example: Bags and Stacks

- Each Bag has
 - State: a bag of Objects (b)
 - isEmpty() : boolean
 - take() : Object
 - Precondition: ! isEmpty()
 - Postcondition: result is an arbitrary member of b. The final value of b is the initial value with the result removed.
 - put(ob : Object)
 - Precondition: true
 - Postcondition: The final value of b is its initial value with ob added.
- Each Stack has
 - State: a sequence of Objects (s)
 - isEmpty() : boolean
 - take() : Object
 - Precondition: ! isEmpty()
 - Postcondition: result is the first item of s. The final value of s is the initial value with the first item removed.
 - put(ob : Object)
 - Precondition: true
 - Postcondition: The final value of s is its initial value with ob prepended.



Example: Bags and Stacks

- Stacks are more constrained than Bags.
- A Stack object could be used where a Bag object is expected without violating our expectations of how a bag should behave.
 - Stack is a behavioral subtype of Bag.

Example of the LSP

```
void someClientCode( Bag t) {
    Assertion.check( t.isEmpty() );
    t.put( newRange(0,N) ) ;
    while( !t.empty() ) {
        Range r = (Range) t.take() ;
        if( r.size() > 2 ) {
            int m = part(r) ;
            t.put( newRange(r.low(), m) ) ;
            t.put( newRange(m, r.high())) ;
        }
    }
}
```

- Clearly the designer has some *expectations about how an instance of Bag will behave*
- *Let S be any subtype of Bag .*
- *If we pass in a direct instance of S , this code should still work.*
- *The expectations we have for instances of Bag should hold for instances of S .*



Expectations about behaviour

- Behavioural specification
 - The behavioural specification of a class explains the “allowable behaviours” of the instances of a class.
- S is a behavioural subtype of T if
 - an instance of type S behaves only as allowed of type T objects
- The LSP then says “subtypes should be behavioural subtypes”
- So where do these expectations about behaviour come from?
 - They have to come from the documentation.
 - Such expectations can not come from the code, because:
 - method implementations may be abstract
 - even if not abstract, method implementations can be overridden



Some cases where the LSP is difficult

- Like a healthy diet the LSP is obviously good for you, but it can be tempting to cheat a little.

- Consider a class

```
public class Point2D {  
    protected double x ;  
    protected double y ;  
    ...  
}
```

Some cases where the LSP is difficult

- Consider Java's toString method (inherited from Object)

```
class Point2D { ...
```

```
/** Return a string representation of the point.
```

```
* Postcondition: result is a string of the form ( xrep, yrep)
```

```
* where xrep is a string representing the x value and
```

```
• yrep is a string representing the y value
```

```
*/
```

```
    @Override public String toString() {  
        return "(" + Double.toString(x) + ", "  
            + Double.toString(y) + ")" ;
```

```
    }
```

```
    ...
```

```
}
```



Some cases where the LSP is difficult

- And Java's "equals" method.

```
class Point2D { ...  
/** Indicate whether two points are equal.  
 * Returns true iff the x and y values are equal. */  
@Override public boolean equals(Object ob) {  
    if( ob instanceof Point2D ) {  
        Point2D that = (Point2D) ob ;  
        return this.x == that.x && this.y == that.y;  
    }  
    else  
        return false ;  
}  
...  
}
```

- So far so good.



Some cases where the LSP is difficult

- Now consider extending Point2D to Point3D

```
public class Point3D extends Point2D {  
    protected double z ;  
    ...  
}
```

- We define toString as


```
@Override public String toString() {  
    return "("+ Double.toString(x) + ", "  
        + Double.toString(y) + ", "  
        + Double.toString(z) + ")" ;  
}
```



Some cases where the LSP is difficult

- Consider:

```
void printPoint( Point2D p ) {  
    p.setX(1.0) ; p.setY(2.0) ;  
    System.out.println( p.toString() ) ;  
}
```

- The behavior will not be as expected if a Point3D is passed in.
 - Surely there is no problem with our code though!
 - The problem is with our expectations.
- 



Two solutions

- Lower expectations

```
/** Return a string representation of the point.
```

```
* Postcondition: result is a string indicating at least the x and  
y values. */
```

```
@Override public String toString() {...as on slide 28...}
```

- Prevent overrides. It would be poor practice to prevent an override of toString(), so use another name

```
/** Return a string representation of the point.
```

```
* Postcondition: ... as on slide 28... */
```

```
public final String toString2D() {... as on slide 28 ...}
```



What about equals?

- Naturally, equals is also overridden in Point3D.

```
@Override public boolean equals(Object ob) {  
    if( ob instanceof Point3D ) {  
        Point3D that = (Point3D) ob ;  
        return this.z == that.z && super.equals(that) ;  
    }  
    else return super.equals( ob ) ;  
}
```

- By the way, the reason for not just returning false, when the other object is not a Point3D, is that “equals” should be symmetric when neither object is null. i.e.

```
p2.equals(p3) == p3.equals(p2)
```



What about equals?

- In the case where $x_0 == x_1$, $y_0 == y_1$, and $p.z \neq q.z$, the following code don't behave according to our expectations.

```
void thisOrThat( Point2D p, Point2D q ) {  
    p.setX( x0 ) ; p.setY( y0 ) ;  
    q.setX( x1 ) ; q.setY( y1 ) ;  
    if( p.equals( q ) ) { ...do this... }  
    else { ...do that... }  
}
```

- Again we have violated the LSP.



Two Solutions

- Reduce Expectations.

- We should reword the documentation of equals for Point2D to be more flexible

```
/** Do two Point2D objects compare equal by the
standard of their least common ancestor class? At
this level the standard is equality of the x and y
values.*/
```

- Prevent overrides

- We wouldn't want to prevent overrides of equals.
We are better off providing a new name

```
/** Are points equal as 2D points? */
public final boolean equals2D( Point2D that ) {
    return this.x==that.x && this.y==that.y;
}
```



Lessons

- Lesson 0:
 - Every class represents 2 specifications
 - One specifies the behavior of its direct instances
 - And this can be reverse engineered from the code.
 - One specifies the behavior of its instances
 - And this can only be deduced from its documentation.
 - It is important to document the behavior that can be expected of all instances.
 - It is less important to document the behavior that can be expected of direct instances.
 - However sometimes it is useful to do both.



Lessons

- Lesson1:
 - When documenting methods that may be overridden, one must be careful to document the method in a way that will make sense for all potential overrides of the function.
- Lesson 2:
 - One should document any restrictions on how the method may be overridden.
 - For example consider the documentation of “equals” in Object. It consists almost entirely of restrictions on how the method may be overridden and thus it describes what the clients may expect.

Example: Documentation of `Object.equals(Object)`

- Indicates whether some other object is "equal to" this one.
- The equals method implements an equivalence relation on non-null object references:
 - It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return true.
 - It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
 - It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
 - It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
 - For any non-null reference value `x`, `x.equals(null)` should return false.
- The equals method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns true if and only if `x` and `y` refer to the same object (`x == y` has the value true).



Lessons

- Lesson 3

- It is particularly important to carefully and precisely document methods that may be overridden because one can not deduce the intended specification from the code.
- For example, consider the implementation of equals in Object

```
public boolean equals( Object ob ) {  
    return this == ob ;  
}
```

- compared to the documentation on the previous slide.
- There may not even be any code.

4. Interface Segregation Principle



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

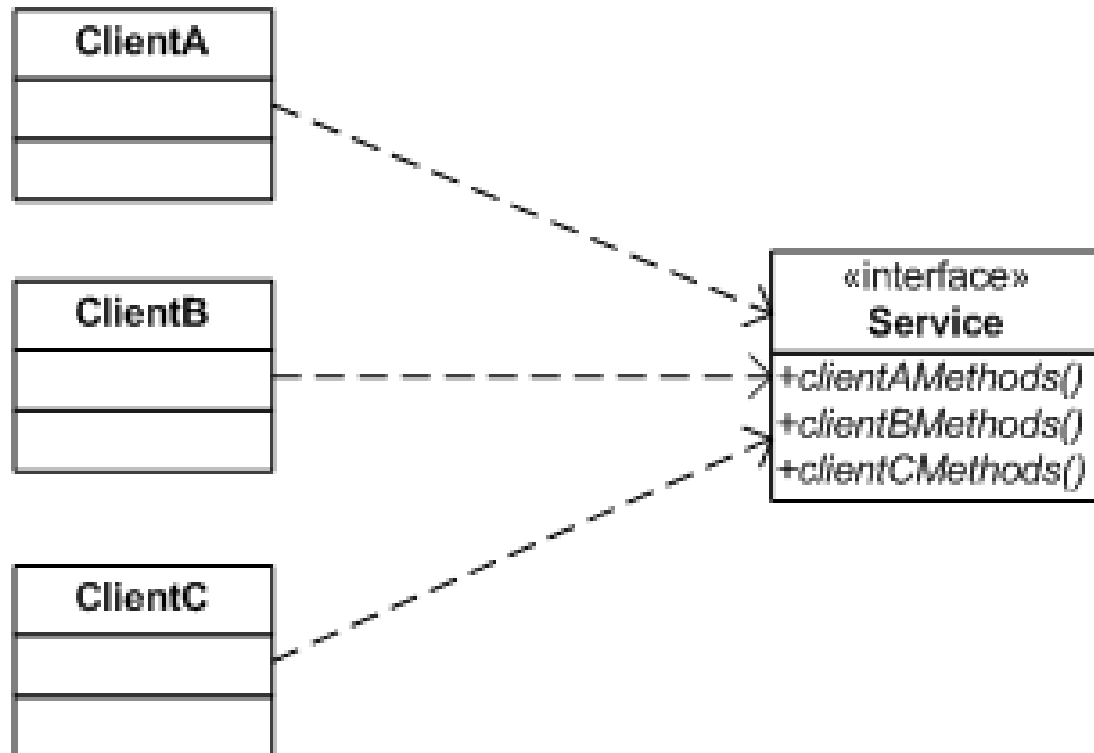


4. Interface Segregation Principle

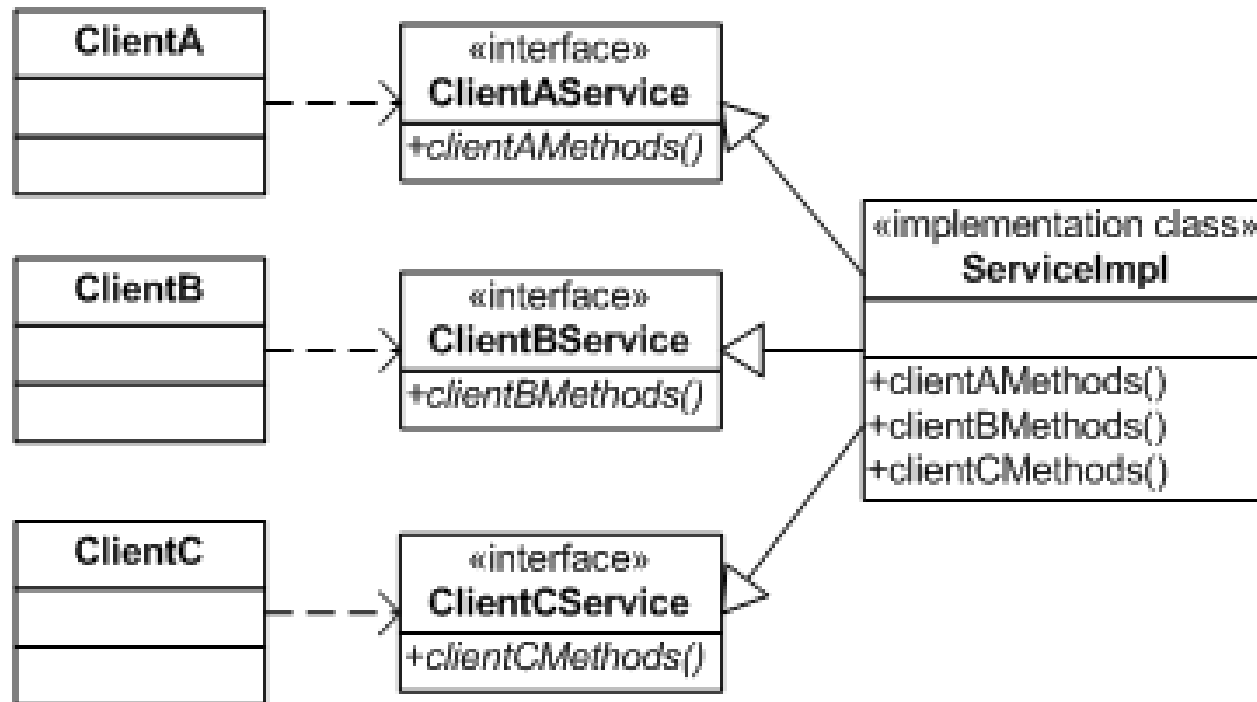
- Assumption
 - You have a class with several clients
 - Each client needs different methods in the class
- Rather than loading the class with all the methods that the clients need:
 - create specific interfaces for each type of client and use multiple inheritance to get functionality into the class

Anti ISP example: "Fat" Service

- Whenever a change is made to one of the methods that ClientA calls, ClientB and ClientC may be affected. It may be necessary to recompile and redeploy them.



ISP Example: Segregated Interfaces



With segregation if the interface for ClientA needs to change, ClientB and ClientC will remain unaffected.

The ISP does not recommend that every class that uses a service have its own special interface class that the service must inherit from.

- Rather, clients should be categorized by their type, and interfaces for each type of client should be created.
- If two or more different client types need the same method, the method should be added to both of their interfaces

5. Dependency Inversion Principle



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

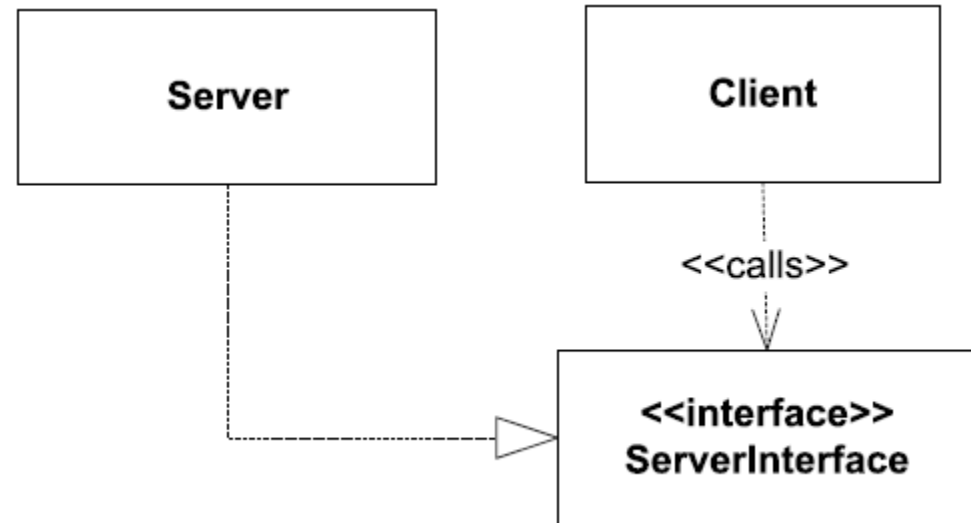


5. Dependency Inversion Principle

- “Depend upon abstractions, not upon concrete entities.”
- “Details should depend upon abstractions. Abstractions should not depend upon details.”
- Dependency Inversion:
 - The strategy is to depend upon interfaces or abstract functions + classes, not upon concrete functions + classes.
 - High-level modules should not depend on low-level modules. Both should depend on abstractions.
 - Abstractions should not depend on details. Details should depend on abstractions.

Dependence Inversion

- Every dependency in the design should target an interface, or an abstract class.
 - No dependency should target a concrete class.



Dependence in Procedural Programming

- Module can be considered as a class with all fields and methods being static, i.e. classes without objects
- In a procedural language, if a procedure in module C calls a method in module S, there is a dependence between C and S.

In java terms

```
classC { ... S.f() ... }
```

```
classS { ... public static void f() { ... } ... }
```

- Since callers are directly coupled to their servers, the callers are not reusable.
- People would make reusable subroutines but it was awkward to make reusable callers.
- Thus dependence traditionally follows the direction of the calls.

Dependence in Procedural Programming

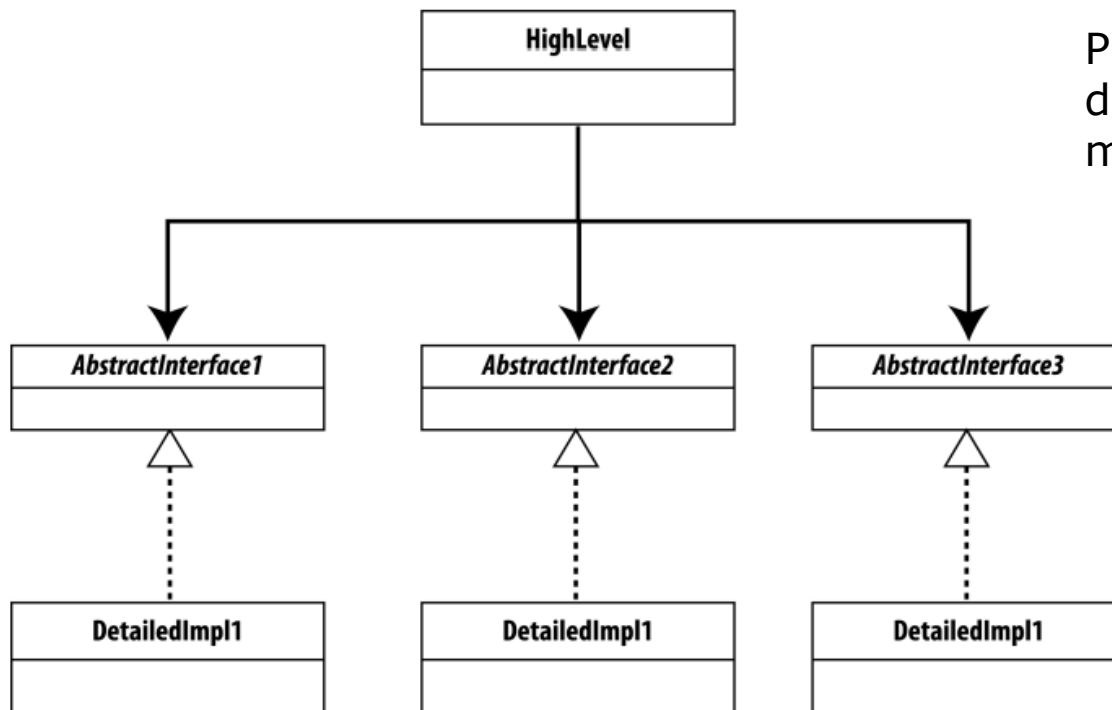
- One exception is that some languages allowed pointers to subroutines as parameters. So in C, for example, we can do the following:

```
double integrate( double(*f)(double),  
    double low, double high, int steps ) {  
    ...  
    sum += f(x)* width ;  
    ...  
}
```

- So integrate is a reusable caller

Object Oriented Architecture

- One of the most common places that designs depend upon concrete classes is when those designs create instances.
 - By definition, you cannot create instances of abstract classes.
 - There is an elegant solution to this problem named Abstract Factory.



Prevent programmers from depending upon volatile modules.

Typically, concrete things change a lot, abstract things change much less frequently.

Dependence in OO programming

- In OO programming, the simplest thing to do is often to have dependence follow the direction of calls:

```
class S { ... public void f() { ... } ... }
```

and

```
class C { ... void g(S s) { ... s.f() ... ; }
```

or

```
class C { S s = new S() ; ... s.f() ... ; }
```

or

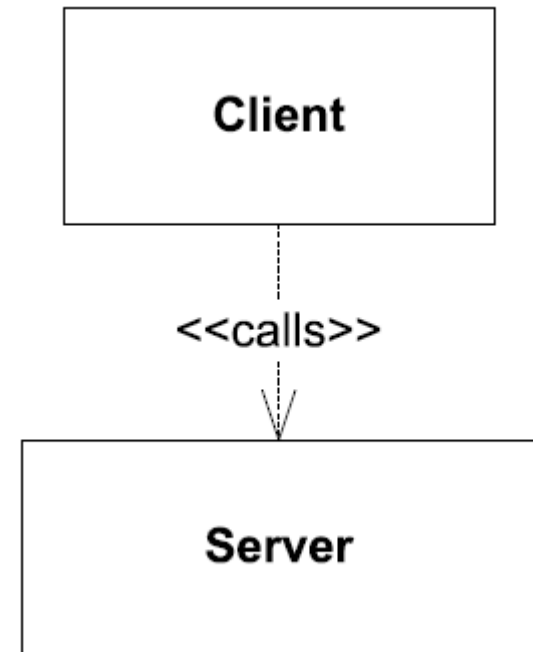
```
class C {  
    S s ;  
    C(S s) { this.s= s ; }  
    ... s.f() ...  
}
```

}

or

```
class C {  
    S s ;  
    setS(S s) { this.s= s ; }  
    ... s.f() ...  
}
```

}

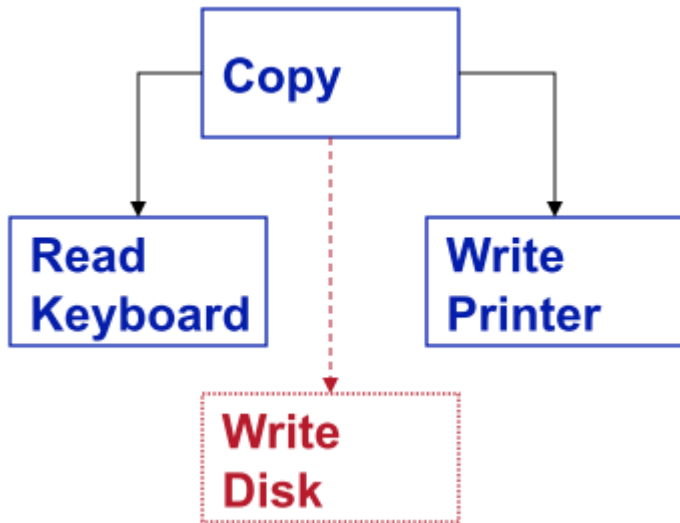




Dependence in OO programming

- This style
 - makes it impossible to reuse the caller independently and
 - discourages the designer from viewing the task of the client without reference to the details of what one specific server will do.
 - I.e. it discourages the separation of the concrete interface that one server happens to provide from the abstract interface that the client requires.

Example:



```
enum OutputDevice {printer, disk};  
void Copy(OutputDevice dev){  
    int c;  
    while((c = ReadKeyboard())!= EOF)  
        if(dev == printer)  
            WritePrinter(c);  
        else  
            WriteDisk(c);  
}
```

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



How to apply DIP on this example ?



Quiz and Exercises

- Now let's go over what you have learned through this lesson by taking a quiz.
- When you're ready, press Start button to take the quiz

