Basic OO Principles
OO Design Principles

Toni Sellarès
Universitat de Girona

Basic OO Principles

- Abstraction
- Encapsulation,
- Inheritance,
- Polymorphism,
- Composition.

Abstraction and Encapsulation

Abstraction and encapsulation:

- are highly-related concepts that often became confused with one another.

Abstraction is a technique that:

- helps us identify which information should be visible (essential), and which information should be hidden (non essential).

- focuses on obtaing interfaces (outside view) of objects.

Encapsulation is a technique for:

- packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible.

- hiding implementation details.

Abstraction and Encapsulation

The end goal is to have a software architecture that facilitates both substitution and reuse.

Helps the system evolve with minimal collateral damage from changes:

– We can change one part of the system without having to change others.

Easyier maintainability, flexibility and extensibility of code.

Inheritance [IS-A relationship]

Method of reuse in which a new functionality is obtained by extending the implementation of an existing class.

The generalization class (the superclass) explicitly captures the common attributes and methods.

The specialization class (the subclass) extends the implementation with additional attributes and methods.

Polymorphism (1)

Is the ability of objects belonging to different classes to respond to method calls of methods of the same name, each one according to an appropriate classe-specific behaviour.

The different objects involved only need to present a compatible interface to the clients: there must be public methods with the same name and the same parameter sets in all the objects.

Polymorphism (2)

The program does not have to know the exact classe of the object in advance, so this behavior can be implemented at run time.

Polymorphism allows client programs to be written based only on the abstract interfaces of the objects which will be manipulated (interface inheritance).

This means that future extension in the form of new classes of objects is easy, if the new objects conform to the original interface.

Composition [HAS-A relationship] (1)

Method of reuse in which a new functionality is obtained by creating an object composed of other objects.

The new functionality is obtained by delegating functionality to one of the objects being composed.

Composition encapsulates several objects inside another one.

OO Design Principles and Heuristics

- ***Minimize the Accessibility of Classes and Members.***
- ***Encapsulate what varies.***
- ***Favor Composition over Inheritance.***
- ***Program To an Interface, Not an Implementation.***
- ***Software Entities (Classes, Modules, Functions) should be Open for Extension, but Closed for Modification.***
- ***Functions that use references to base classes must be able to use objects of derived subclasses without knowing it.***
- ***Depend On Abstractions. Do not depend on Concrete Classes.***

Principle

Minimize the Accessibility of Classes and Members

Private Methods

Provide a way of designing a class behavior so that external objects are not permitted to access the behavior that is meant only for the internal use.

Accessor Methods

Provide a way of accessing an object's state using specific methods.

This approach discourages different client objects from directly accessing the attributes of an object, resulting in a more maintainable class structure.

Principle

Encapsulate what varies

Encapsulate what varies

No matter how well you design an application, over time it must grow and change or it will die.

Identify the aspects of your application that may vary, say with every new requirement.

Take the parts that vary and encapsulate them: hide the details of what can change behind the public interface of a class.

This allows you to alter or extend these parts without affecting the parts that don't change.

This principle is the basis for almost every design pattern.

When designing software, look for the portions most likely to change and prepare them for future expansion by shielding the rest of the program from that change.

Hide the potential variation behind an interface.

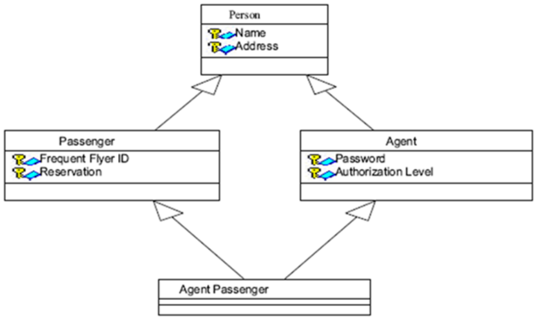
Then, when the implementation changes, software written to the interface doesn't need to change.

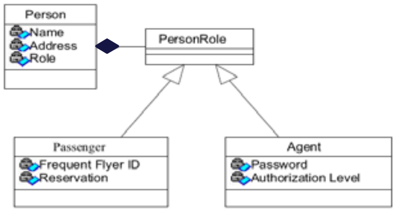
Principle

Favor Composition over Inheritance

There are two common techniques for reusing functionality in object oriented systems:

1. **class inheritance**
2. **object composition**





Pros and Cons of Inheritance (1)

Advantages:

- New implementation is easy, since most of it is inherited.
- Easy to modify or extend the implementation being reused.

Pros and Cons of Inheritance (2)

Disadvantages:

- Breaks encapsulation, since it exposes a subclass to implementation details of its superclass.
- White-box reuse, since internal details of superclasses are often visible to subclasses.
- Subclasses may have to be changed if the implementation of the superclass changes.
- Implementations inherited from superclasses can not be changed at runtime.

Pros and Cons of Composition (1)

Advantages:

- Composing objects are accessed by the composed class solely through their interfaces.
- Black-box reuse, since internal details of composed objects are not visible.
- Good encapsulation.
- Fewer implementation dependencies.
- Each class is focused on just one task.
- The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type.

Pros and Cons of Composition (2)

Disadvantages:

- Resulting systems tend to have more objects and inter-relationships between them than when it is defined in a single class.
- Interfaces must be carefully defined in order to use many different objects as composition blocks.

Inheritance/Composition Summary

- Both composition and inheritance are important methods of reuse.
- Inheritance was overused in the early days of OO development.
- Over time we've learned that designs can be made more reusable and simpler by favoring composition.
- Of course, the available set of composable classes can be enlarged using inheritance:
  - So composition and inheritance work together.
- But our principle is:

Favor Composition Over Inheritance

Coad's Rules

Use inheritance only when all of the following criteria are satisfied:

– A subclass expresses "is a special kind of" and not "is a role played by a".

– An instance of a subclass never needs to become an object of another class.

– A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass.

– A subclass does not extend the capabilities of what is merely a utility class.

– For a class in the actual Problem Domain, the subclass specializes a role, transaction or device.

Inheritance/Composition Example 1

"Is a special kind of" not "is a role played by a":

- **Pass**. Reservation and purchase are a special kind of transaction.

Never needs to transmute:

- **Pass**. A Reservation object stays a Reservation object; the same is true for a Purchase object.

Extends rather than overrides or nullifies:

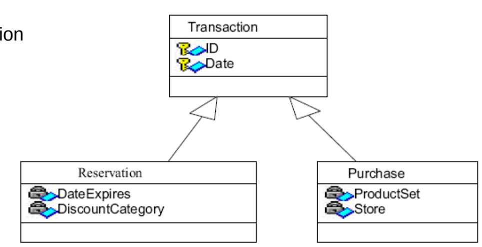
- **Pass**.

Does not extend a utility class:

- **Pass**.

Within the Problem Domain, specializes a role, transaction or device:

- **Pass**. It's a transaction.



Inheritance ok here!

Inheritance/Composition Example 2 (1)

"Is a special kind of" not "is a role played by a“:

- **Fail**. A passenger is a role a person plays. So is an agent.

Never needs to transmute:

- **Fail**. A instance of a subclass of Person could change from Passenger to Agent to Agent Passenger over time.

Extends rather than overrides or nullifies:

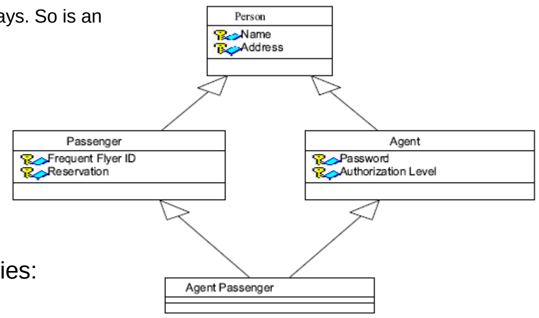
- **Pass**.

Does not extend a utility class:

- **Pass**.

Within the Problem Domain, specializes a role, transaction or device:

- **Fail**. A Person is not a role, transaction or device.



Inheritance does not fit here!

Inheritance/Composition Example 2 (2)

"Is a special kind of" not "is a role played by a“:

- **Pass**. Passenger and agent are special kinds of person roles.

Never needs to transmute:

- **Pass**. A Passenger object stays a Passenger object; the same is true for an Agent object.

Extends rather than overrides or nullifies:

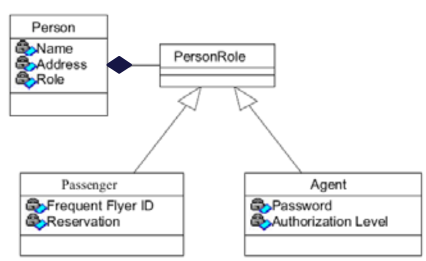
- **Pass**.

Does not extend a utility class:

- **Pass**.

Within the Problem Domain, specializes a role, transaction or device:

- **Pass**. A PersonRole is a type of role.



Inheritance ok here!

Principle

Program To an Interface (Supertype), Not an Implementation

Interface and implementation

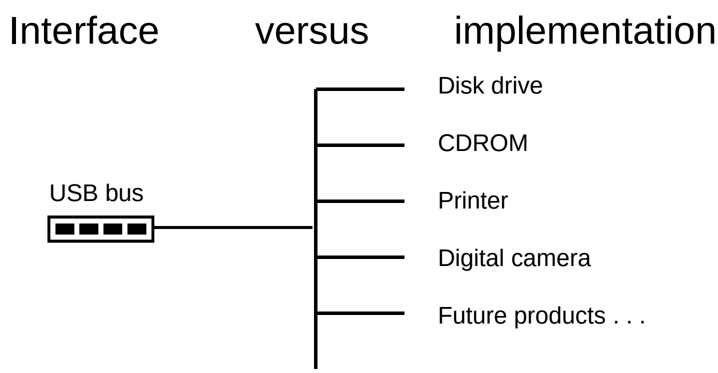
Interface is a subset of all the methods that an object implements.

Implementation is the code.

Interface is MORE IMPORTANT than implementation:

- Interface, once decided, is hard to change,
- Implementation can be easily changed.

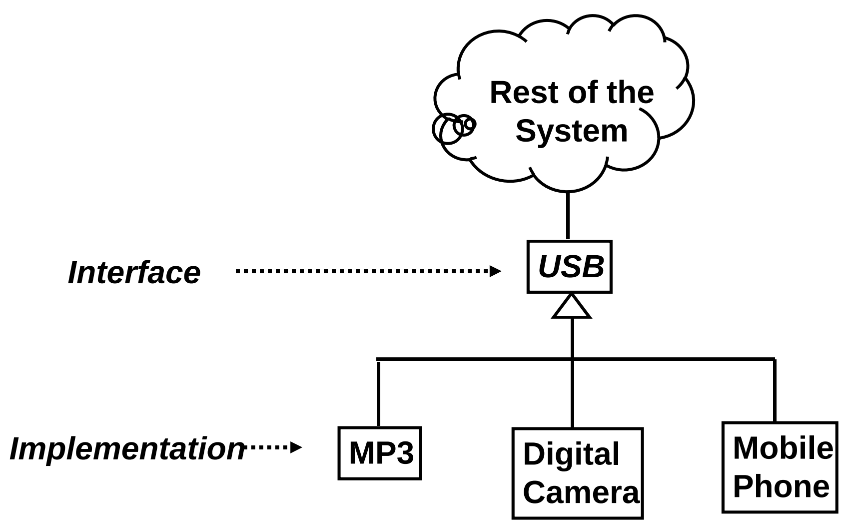
Interface and implementation



USB is an *abstraction* of the hardware

- Design the PC system around the USB interface, not to a particular device

Interface and implementation



Interfaces

- Interface is a subset of all the methods that an object implements.
- An object can have many interfaces
- A *type* is a specific interface of an object.
- Different objects can have the same type and the same object can have many different types.
- An object is known by other objects only through its interface.

Interface Inheritance

- Interface Inheritance (Subtyping) describes when one object can be used in place of another object.

Benefits of Interfaces

Advantages:

- Clients are unaware of the specific class of the object they are using.
- One object can be easily replaced by another.
- Object connections need not be hardwired to an object of a specific class, thereby increasing flexibility.
- Loosens coupling.
- Increases likelihood of reuse.
- Improves opportunities for composition since contained objects can be of any class that implements a specific interface.

Disadvantages:

- Modest increase in design complexity.

Interface Example (1)

```
/*
* Interface IManeuverable provides the specification for a
* maneuverable vehicle.
*/
public interface IManeuverable {
public void left();
public void right();
public void forward();
public void reverse();
public void climb();
public void dive();
public void setSpeed(double speed);
public double getSpeed();
}

public class Car
   implements IManeuverable { // Code here. }

public class Boat
   implements IManeuverable { // Code here. }

public class Submarine
  implements IManeuverable { // Code here. }
```

Interface Example (2)

This method in some other class can maneuver the vehicle without being concerned about what the actual class is (car, boat, submarine) or what inheritance hierarchy it is in:

```
public void travel(IManeuverable vehicle) {
    vehicle.setSpeed(35.0);
    vehicle.forward();
    vehicle.left();
    vehicle.climb();
}
```

The Open-Closed Principle (OCP)

Software Entities (Classes, Modules, Functions) should be Open for Extension, but Closed for Modification

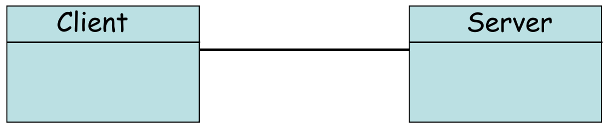
OCP

- OCP states:
  - We should attempt to design modules that never need to be changed: we do not modify old code.
  - We extend the behavior of the system by adding new code.
- OCP attacks software rigidity and fragility:
  - When one change causes a cascade of changes!

OCP

- It is not possible to have all the modules of a software system satisfy the OCP, but we should attempt to minimize the number of modules that do not satisfy it.
- The OCP is really the heart of OO design.
- Conformance to OCP yields the greatest level of reusability and maintainability.

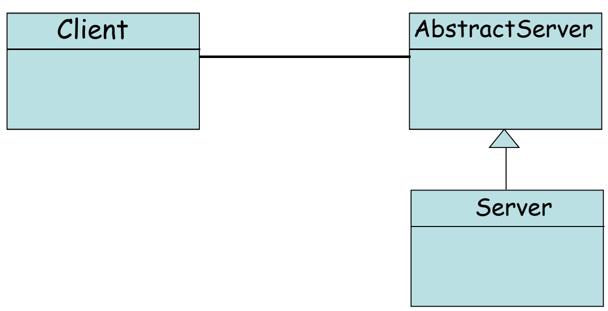
Example: "*Closed Client*"



- Client and Server are concrete classes
- Client class uses Server class
- If Client object wants to switch to a different Server object, what would need to happen?

Client code needs to be modified to name the new Server class !

Example: "*Open Client*"



- How is this "open" ?

Since the Client depends on the AbstractServer, we can simply switch the Client to using a different Server, by providing a new Server implementation. Client code is unaffected!

Another OCP Example (1)

- Consider the following method:

```
public double totalPrice(Part[] parts) {
  double total = 0.0;
  for (int i=0; i<parts.length; i++) {
    total += parts[i].getPrice();
  }
  return total;
}
```

- The job of the above method is to total the price of all parts in the specified array of parts.
- Does this conform to OCP?
  - ***YES***! *If Part is a base class or an interface and polymorphism is being used, then this class can easily accommodate new types of parts* ***without*** *having to be modified!*

OCP Example (2)

- But what if the Accounting Department now decreed that *motherboard parts* and *memory parts* have a premium applied when figuring the total price?
- Would the following be a ***suitable*** modification? Does it ***conform to OCP***?

```
public double totalPrice(Part[] parts) {
  double total = 0.0;
  for (int i=0; i<parts.length; i++) {
    if (parts[i] instanceof Motherboard)
      total += (1.45 * parts[i].getPrice());
    else if (parts[i] instanceof Memory)
      total += (1.27 * parts[i].getPrice());
    else
      total += parts[i].getPrice();
  }
  return total;
}
```

OCP Example (3)

No! Every time the Accounting Department comes out with a new pricing policy, we have to modify totalPrice ( ) method. This is **not “*Closed for modification*”**

These policy changes have to be implemented some place, so what is a solution?

Version 1. Could incorporate the pricing policy in getPrice ( ) method of Part.

OCP Example (4)

- Here are example Part and Concrete Part classes:

```
// Class Part is the superclass for all parts.
public class Part {
  private double price;
  public Part(double price) (this.price = price;}
  public void setPrice(double price) {this.price = price;}
  public double getPrice() {return price;}        Add method
}

// Class ConcretePart implements a part for sale.
// Pricing policy explicit here!
public class ConcretePart extends Part {
  public double getPrice() {
    // return (1.45 * price);    //Premium
    return (0.90 * price);       //Labor Day Sale
  }
```

Code in concrete classes

•Does this work? Is it “***closed for modification***”?

–**No**. *We must now modify each subclass of Part whenever the pricing policy changes!*

How to make it "*Closed for Modification*"

- Version 2. Better idea: have a PricePolicy class which can be used to provide different pricing policies:

```
// The Part class now has a contained PricePolicy object.
public class Part {
  private double price;
  private PricePolicy pricePolicy;

  public void setPricePolicy(PricePolicy pricePolicy) {
    this.pricePolicy = pricePolicy;}
  public void setPrice(double price) {this.price = price;}
  public double getPrice() {return pricePolicy.getPrice(price);}
}
```

OCP Example (5)

```
/**
 * Class PricePolicy implements a given price policy.
 */
public class PricePolicy {
  private double factor;

  public PricePolicy (double factor) {
    this.factor = factor;
  }

  public double getPrice(double price) {return price * factor;}

}
```

With this solution we can dynamically set pricing policies at run time by changing the PricePolicy object that an existing Part object refers to.

Corollary to OCP: Single Choice Principle

Whenever a software system must support a set of alternatives, ideally only one class in the system knows the entire set of alternatives

The Liskov Substitution Principle (LSP)

Functions that use references to base classes (super classes) must be able to use objects of derived subclasses without knowing it.

LSP

If a function does not satisfy the LSP, then it probably makes explicit reference to some or all of the subclasses of its superclass.

Such a function also violates the OCP, since it may have to be modified whenever a new subclass is created.

LSP Example

The Liskov Substitution Principle seems obvious given polymorphism.

For example:

```
public void drawShape (Shape s) {
    // code here
}
```

The drawShape method should work with any subclass of the Shape superclass (or, if Shape is a Java interface, it should work with any class that implements the Shape interface).

So what is the big deal with LSP?

LSP Example (1)

Consider the following **Rectangle** class:

```
// A very nice Rectangle class
public class Rectangle {
    private double width;
    private double height;

    public Rectangle (double w, double h) {
        width = w;
        height = h;
    }
    public double getWidth ( ) { return width;}
    public double getHeight ( ) { return height;}
    public void setWidth (double w) { width = w; }
    public void setHeight (double h) {height = h;}
    public double area ( ) { return (width * height);}
}
```

LSP Example (2)

Assume we need a Square class. Clearly a square is a rectangle, so the Square class should be derived from the Rectangle class.

Observations:

- A square does not need both a width and a height as attributes, but it will inherit them from Rectangle anyway. So each Square object wastes a little memory -- but this is not a major concern.
- The inherited setWidth ( ) and setHeight ( ) methods are not really appropriate for a Square, since the width and height of a square are identical. So we'll need to override the methods setWidth ( ) and setHeight ( ).

LSP Example (3)

Here's the Square class:

```
// A Square class
public class Square extends Rectangle {
    public Square (double s) { super (s, s);}

    public void setWidth (double w) {
        super.setWidth (w);
        super.setHeight(w);
    }

    public void setHeight (double h) {
        super.setHeight (h);
        super.setWidth (h);
    }
}
```

setWidth ( ) and setHeight ( ) overridden to reflect Square semantics

LSP Example (4)

- Everything looks good. But consider this function!

```
public class TestRectangle {
  // Define a method that takes a Rectangle reference.
   public static void testLSP (Rectangle r) {
       r.setWidth (4.0);
       r.setHeight (5.0);
       System.out.println ("Width is 4.0 and Height is 5.0" + ", Area" + r.area () );
        if (r.area ( ) == 20.0 ){
              System.out.println ("Looking good \n");
        else
              System.out.println("Huh?? What kind of rectangle is this?? \n");
   }
```

LSP Example (5)

```
public static void main (String args[] ) {

  // Create a Rectangle and a Square
  Rectangle r = new Rectangle (1.0, 1.0);
  Square s = new Square (1.0);

  // Now call the testLSP method. According to LSP it should work for either
  // Rectangles or Squares. Does it?
   testLSP ( r );
   testLSP (s);
 }
}
```

LSP Example (6)

Test program output:

Width is 4.0 and Height is 5.0, so Area os 20.0
Looking good!

Width is 4.0 and Height is 5.0, so Area is 25.0
Huh?? What kind of rectangle is this??

Looks like we violated LSP!

LSP Example (7)

- The programmer of the testLSP ( ) method made the reasonable assumption that changing the width of a Rectangle leaves its height unchanged.
- Passing a Square object to such a method results in problems, exposing a violation of LSP.
- The Square and Rectangle classes look self consistent and valid. Yet a programmer, making reasonable assumptions about the base class, can write a method that causes the design model to break down.
- Solutions cannot be viewed in isolation, they must also be viewed in terms of reasonable assumptions that might be made by the users of the design.

LSP Example (8)

- A mathematical square might be a rectangle, but a Square object is a not a Rectangle object, because the behavior of a Square object is not consistent with the behavior of a Rectangle object!
- Behaviorally, a Square is *not* a Rectangle! A Square object is hence not polymorphic with a Rectangle object.
- Hint on LSP violation: when simple methods such as the setWidth and setHeight have to be overridden, inheritance needs to be re-examined!

LSP: Conclusions

- The Liskov Substitution Principle (LSP) makes it clear that the ISA relationship is all about behavior.
- In order for the LSP to hold (and the OCP) all subclasses must conform to behavior that the clients expect of the base classes they use.
- A subtype must have no more constraints than its base type, since the subtype must be usable anywhere the base type is usable.
- If the subtype has more constraints than its base type, there would be uses that would be valid for the base type, but that would violate one of the extra constraints of the subtype and thus violate the LSP!
- The guarantee of the LSP is that a subclass can always be used wherever its base class is used!

Corollary to LSP: Design by contract Principle

A subtype can only have weaker preconditions and stronger postconditions than its base class

Dependency Inversion Principle (DIP)

Depend On Abstractions,

Not on Concrete Classes.

DIP

- DIP is naïve but powerful principle
  - High-level components should not depend on low-level components. Both should depend upon abstractions.
  - Abstractions should not depend upon details. Details should depend upon abstractions.
  - All relationships in a program must terminate at an abstract class or interface.
- According to this heuristic:
  - No variable should hold a pointer or reference to a concrete class.
  - No class should derive from a concrete class.
  - No method should override an implemented method of any of its base classes.
- This heuristic is violated at least once in every program
  - Classic example: use of String class.

DIP

Dependency is transitive: Changes in the lower level modules can have direct effects on the higher level modules, forcing them to change in turn.

- Absurd! High level modules should be driving change not the other way round.

Reuse: We should be able to reuse the high level, policy setting modules.

- Pretty good already at reusing the lower level implementations libraries.
- Difficult to reuse higher level modules when they depend on lower level details.

Conclusions: Strive for having higher level modules be independent of the lower level modules.

DIP is at the heart of framework design!

DIP

Class X depends on class Y if any of the following applies:

- X has a Y and calls it
- X is a Y
- X depends on some class Z that depends on Y (transitivity)

X depends on Y does not imply Y depends on X. If both happen to be true this is called a cyclic dependency: X can't then be used without Y, and viceversa.

The existence of a large number of cyclic dependencies in an object oriented program might be an indicator for suboptimal program design.

DIP

Breaking up a dependency with Inversion of Control

If an object x of class X calls methods of an object y of class Y, then class X depends on Y.

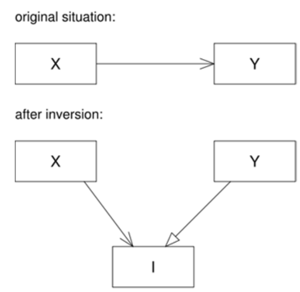
The dependency can now be inverted by introducing a third class, namely an interface class I that must contain all methods that x might call on y.

Furthermore, Y must be changed such that it implements interface I.

X and Y are now both dependent on interface I and class X no longer depends on class Y (presuming that X does not instantiate Y).

The newly introduced interface I depends on nothing.

This elimination of the dependency of class X on Y by introducing an interface I is said to be an inversion of control or a dependency inversion.



DIP helps to make design OCP compliant !

DIP

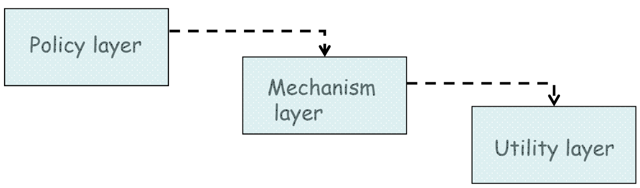
Inversion of Control:

- Each high level module declares an abstract interface for the services it needs
- Lower level layers are realized using through the abstract interface
- Here:
  - Upper level layers do not depend on the lower level modules
  - Lower layers depend on the abstract service layers declared in the upper layers!

DIP

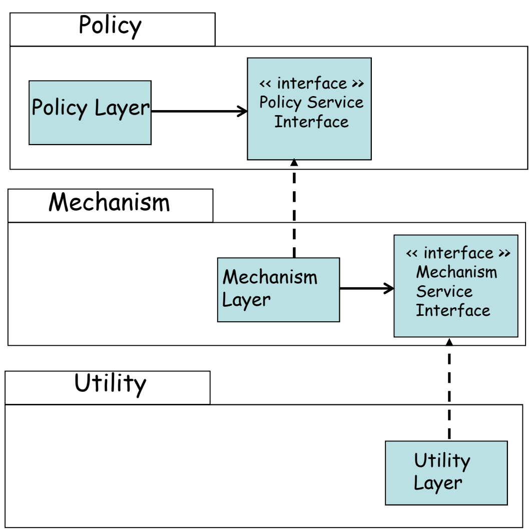
Traditional functional programming:

- High level components: business/application rules
- Low level components: implementation of the business rules
- High level components complete their functionality by calling/invoking the low level implementation provided by the low level components
  - High level depends on the lower level



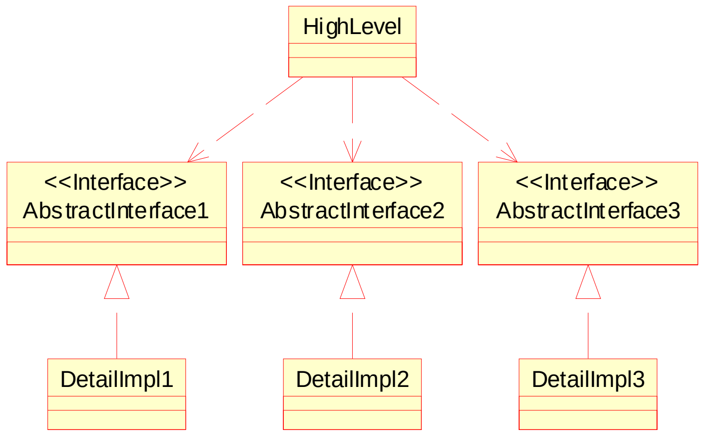
DIP

Better "inverted" Design



- Design applies DIP.
- Inversion of Ownership!
  - The client "owns" the interface
  - The utility libraries don't own the interfaces they implement.
- Sometimes called the *Hollywood principle*: "Don't call us, we'll call you!"
- Advantages:
  - Policy Layer is now unaffected by changes in the Utility Layer
  - Policy Layer is now reusable!
  - Inverting Dependencies breaks:
    - Transitive dependency
    - Direct dependency in most cases
  - Provides design that is more flexible, durable and mobile!

DIP



DIP Example (1)

- Dependency Inversion Principle can be applied whenever one class sends a message to another.

- Consider the case of a Button and Lamp object.
  - Button:
    - senses the external environment
    - receives poll message
    - Determines whether or not user has "***pressed***" it.
  - Lamp:
    - Affects the external environment
    - On receiving turnOn message, illuminates the light
    - On receiving turnOff message, extingishes the light.

- Actual physical mechanism for the Lamp and the Button is irrelevant.

DIP Example (2)

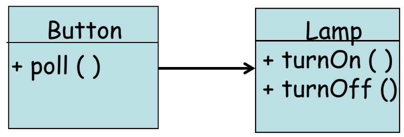
Naïve Design

```
public class Button {
    private Lamp itsLamp;
    public Button (Lamp l) { itsLamp = l;}

    public void poll ( ){
        if (/* some condition*/)
            itsLamp.turnOn ( );
        else
            itsLamp.turnOff ( );
    }

}

public class Lamp {
    public void turnOn ( );
    public void turnOff ( );
}
```



Why is this a naïve design?

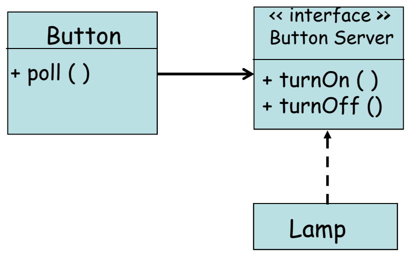
DIP Example (3)

Why is the design naïve?

- The dependency between Lamp and Button implies that Lamp cannot be modified without changing (at least recompiling).
- Also - not possible to reuse the Button class to control a Motor/Portal object.
- The Button and Lamp code violates the DIP.

DIP Example (4)

Applying DIP:



DIP

Final Comments:

- 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.
  - Thus to create instances you must depend on concrete classes!
  - An elegant solution to this problem - *Abstract Factory* Pattern
- If a class/module is concrete but extremely stable DIP can be relaxed.