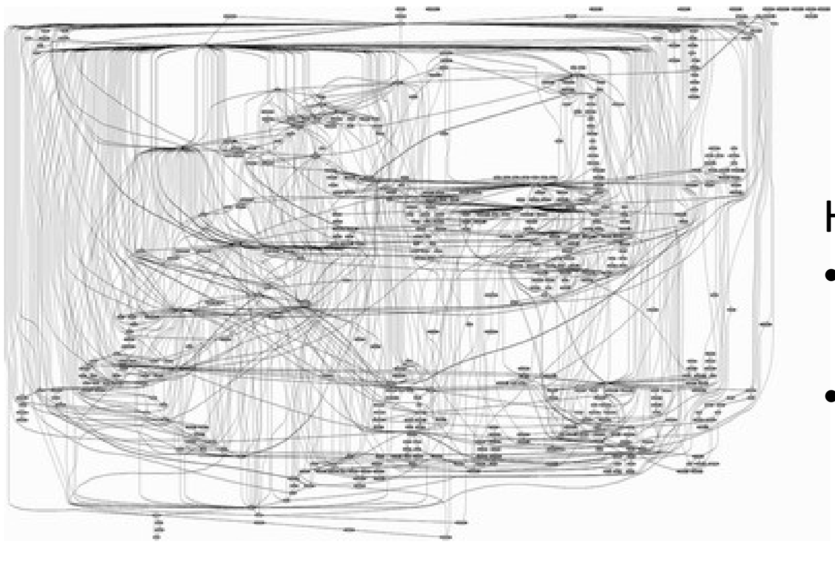
DESIGN PRINCIPLE-BASED REFACTORING: LOW COUPLING

Coupling

- Degree of interdependence between two modules.
- Highly coupled systems are harder to understand and maintain.



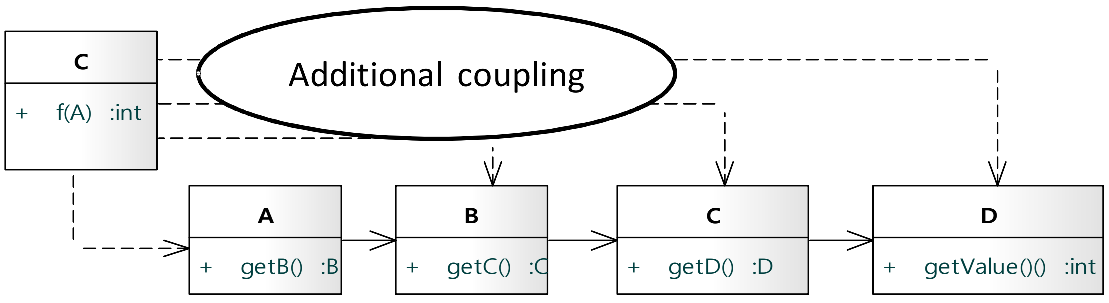
How to achieve low coupling

- Eliminate unnecessary relationships
- Reduce the number of necessary relationships

Coupling-related Smells

- **Message chains**: You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, and so on

```
class C {
  public int f(A a) {
    return a.getB().getC().getD().getValue() ;
  }
}
```

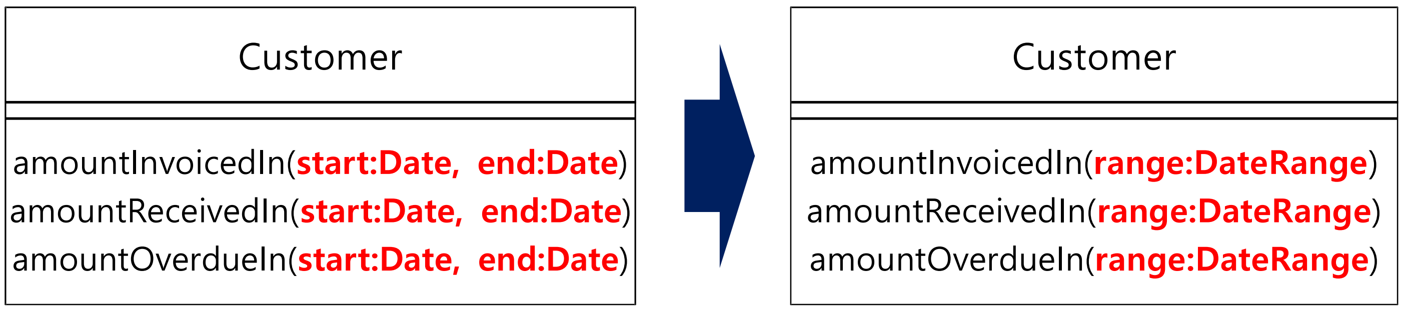


Refactoring Techniques

Smell	Refactoring
Data clump	Introduce parameter object
Long parameter list	Preserve whole object Replace parameter with method call
control couple	Replace parameter with explicit method
Law of Demeter TDA Message chains	Hide delegate
Fat interface	Extract interface, ISP
High fan out	Façade Pattern

Introduce Parameter Object

Data Clump: you have a group of parameters naturally go together


Customer
amountInvoicedIn(start:Date, end:Date)
amountReceivedIn(start:Date, end:Date)
amountOverdueIn(start:Date, end:Date)
Customer
amountInvoicedIn(range:DateRange)
amountReceivedIn(range:DateRange)
amountOverdueIn(range:DateRange)

Data Clump: data items that enjoy hanging around in groups together
Also, long parameter list can be resolved.

Preserve Whole Object

You are getting several values from an object and passing these values as parameters in a method call

Send the whole object instead

```
Weather today = getWeather(new Date())
int low = today.getLow();
int high = today.getHigh();
boolean comfortable = isComfortable(low,high);
```

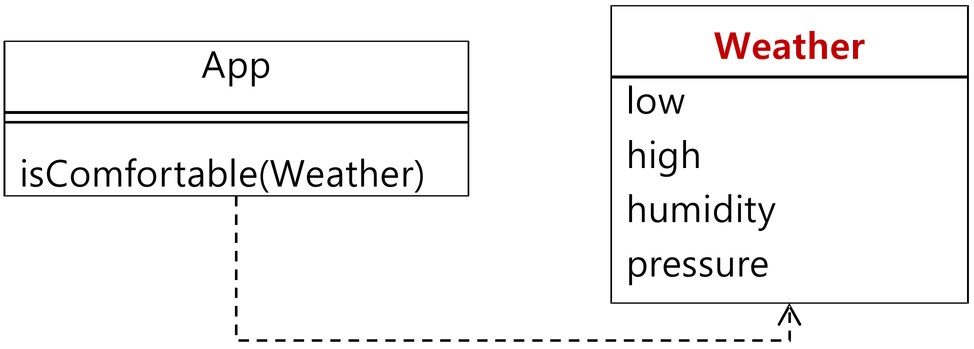
```
Weather today = getWeather(new Date())
boolean comfortable = isComfortable(today);
```

Later, **isComfortable** needs another values such as (humidity) from **Weather**

Preserve Whole Object: CABEAT

Avoid **Stamp Coupling**

Module is passed a composite data structure, but use only a part of it



`isComfortable()` should use low, high, humidity, pressure of **Weather**
Otherwise, it is stamp coupled with **Weather**

Replace Parameter with Method Call

Long Parameter List: Before a method call, a second method is run and its result is sent back to the first method as an arg. But the parameter value could have been obtained inside the first method

```
double getPrice() {
  int basePrice = quatity * itemPrice;
  int discountLevel =  getDiscountLevel();
  double finalPrice = discountedPrice(basePrice, discountLevel);
}
double discountedPrice(int basePrice, int discountLevel) {
  return (discountLevel == 2) ?
    basePrice * 0.1 : basePrice * 0.1;
}
private int getDiscountLevel() {
   return (quantity > 100) ? 2 : 1;
}
```

Replace Parameter with Method Call

```
public double getPrice() {
   return discountedPrice() ;
}

private double discountedPrice()  {
  return (getDiscountLevel() == 2) ?
    getBasePrice() * 0.1 : getBasePrice() * 0.05;
}

private int getDiscountLevel()  {
   return (quantity > 100) ? 2 : 1;
}

private double getBasePrice()  {
   return quantity * itemPrice;
}
```

The method for getting the discount (**discountedPrice**) is now possible to use separately from the method for getting the price (**getPrice**).

Replace Parameter with Explicit Methods

Control Couple: you have a method that runs different code depending on the values of an enumerated parameter

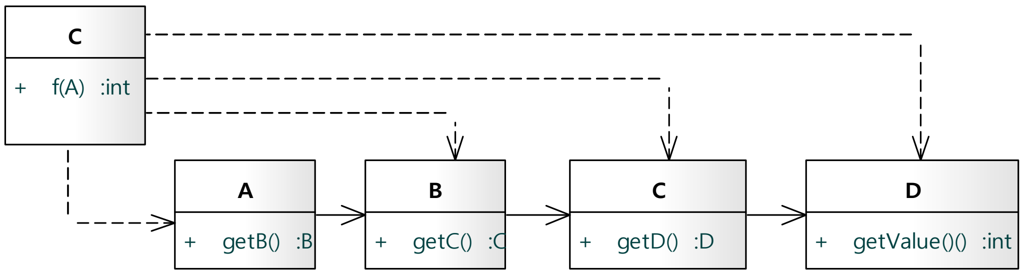
Create a separate method for each value of the parameter

```
void applyDiscount(int type, double discount) {
   if (type == FIXED_DISCOUNT)  {
      price -= discount;
   else //  PERCENT_DISCOUNT:
      price -= price*discount;
   }
}
```

```
void applyFixedDiscount(double discount) {
   price -= discount;
}
void applyPercentDiscount(double percent) {
   price -= price*percent;
}
```

Hide Delegate

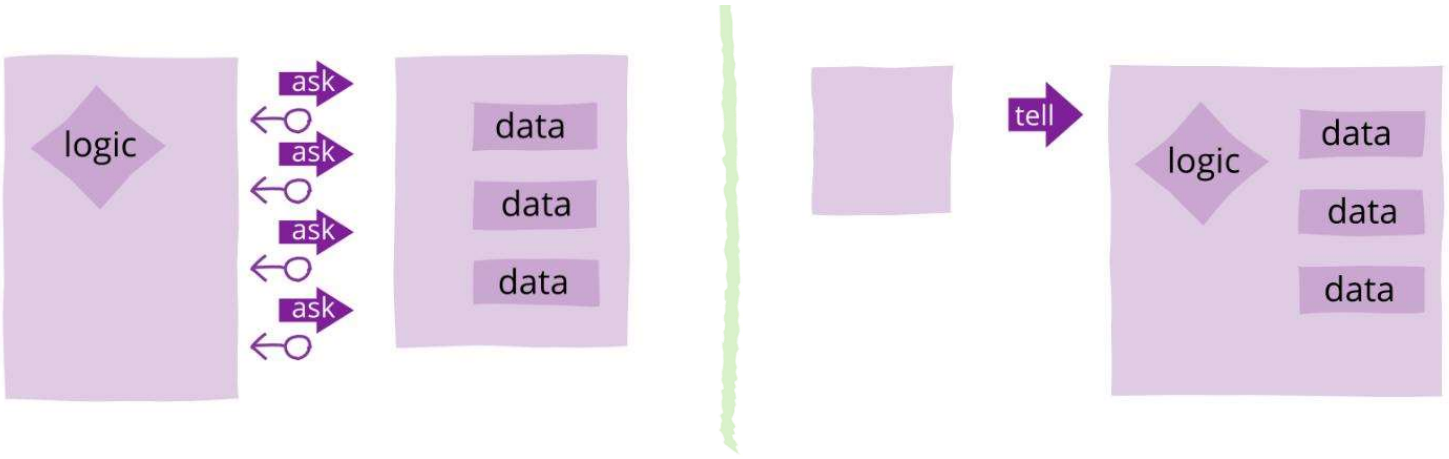
Message Chain: in code you see a series of calls resembling `a.getB().getC().getD.getValue()`. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.



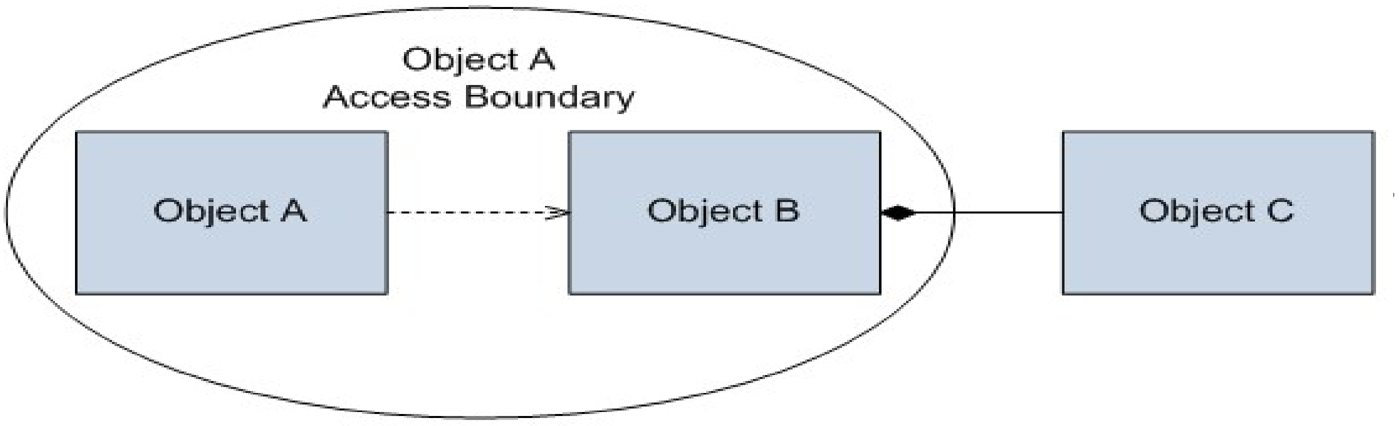
Tell Don't Ask

Object-orientation is about bundling data with the functions that operate on that data.

Rather than asking an object for data and acting on that data, we should instead tell an object what to do.



Don’t Talk to Strangers



Assign the responsibility to a client’s direct object (i.e., *familiar*) to collaborate with an indirect object (i.e., *stranger*), so that the client does not know about the indirect object.

Law of Demeter

(Principle of Least Knowledge)

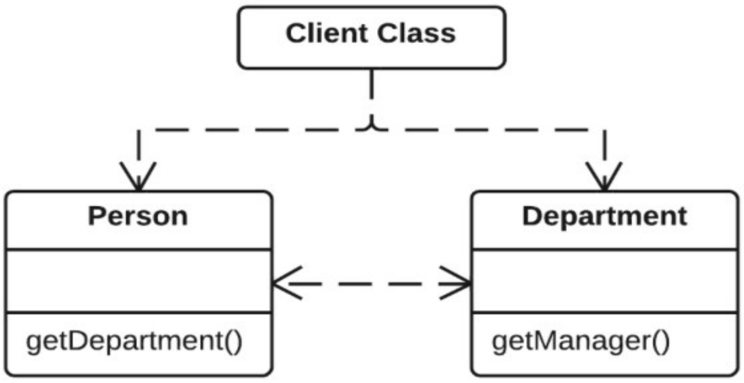
Constrains on what objects you should send a message to within a method.

- The *this* object (or *self*).
- The parameter of the method.
- An attribute of *self*.
- An element of a collection which is an attribute of *self*.
- An object created within the method.

Hide Delegate

delegate — the end object that contains the functionality needed by the client

server — the object to which the client has direct access



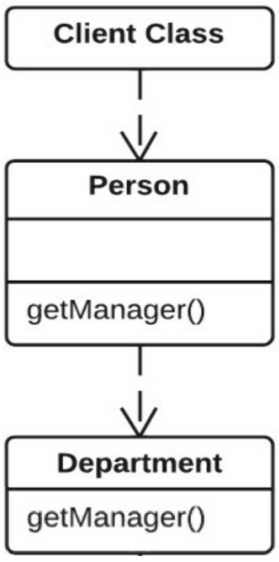
```
class Client {
  public void print(Person person) {
    System.out.println(person.getDepartment().getManager().getName())
}
```

Hide Delegate

Create methods on the *server* to hide the *delegate*

Method should not return its friend object

```
class Client {
  public void print(Person person) {
    System.out.println(person.getManager().getName());
}
```



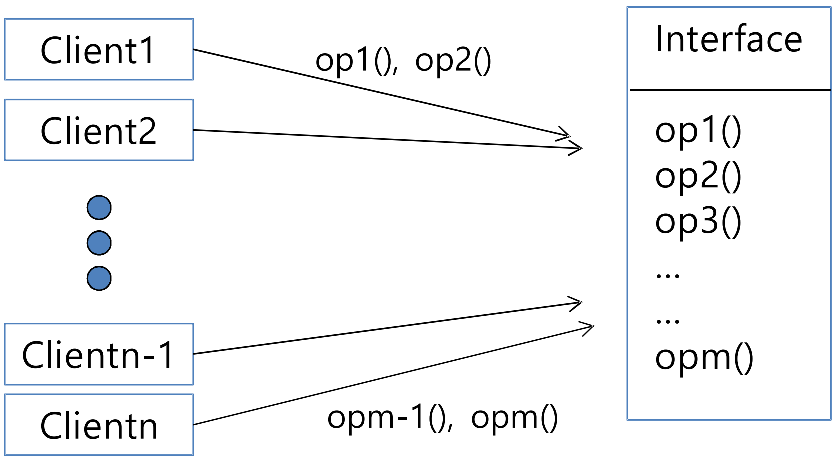
Excessive use may incur smell of **Middle Man**

Fat Interface

Client depends on an interface, but only part of the interface are used.

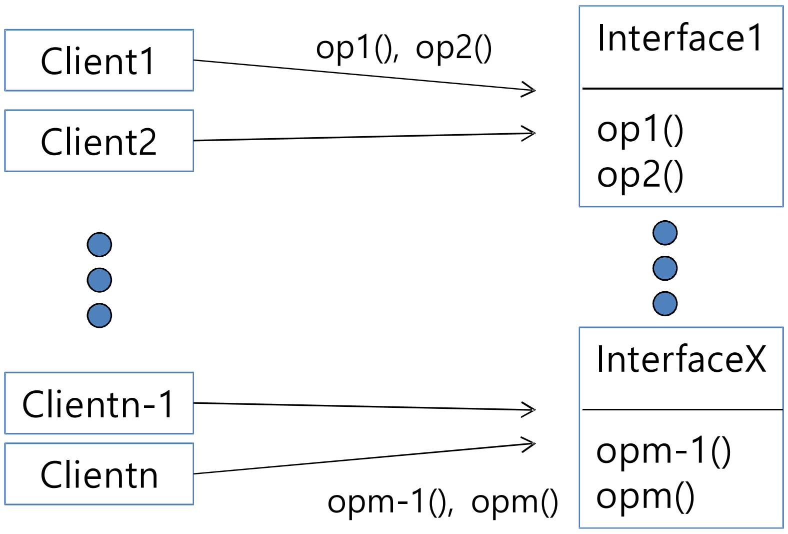
Create unnecessary dependency

Fat interface causes inadvertent changes to unrelated clients



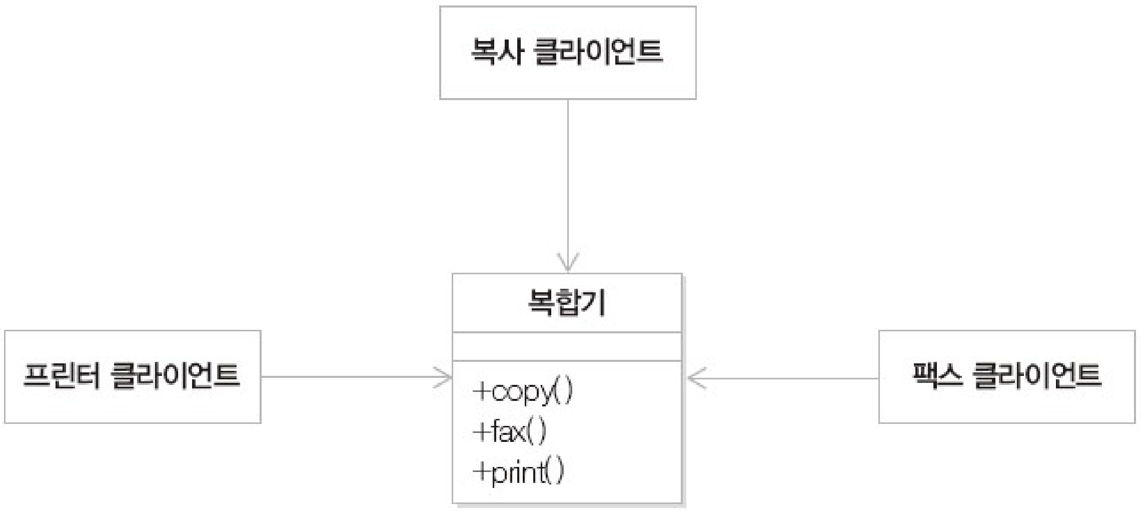
ISP: The Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use.



Violation of the ISP violates the SRP.

ISP



Improved Design

Coupling-Related Patterns

- Façade Pattern