DESIGN PRINCIPLE-BASED REFACTORING: SRP

SRP: *A class should have one, and only one reason to change*

Operations in the class should be closely related to one subject area → *strong cohesion*

Shift the meaning a little bit, then operations in the class should be closely related to one responsibility.

In the context of the SRP, a responsibility is defined as *"a reason for change"* or *"axis of change"*

If you can think of more than one motive for changing a class, you are violating the SRP.

Employee
doAsEmployee() genReport() saveToDB()

SRP: *A class should have one, and only one reason to change*

When the requirements change, that change will be manifest through a change in responsibility amongst the classes.

A responsibility is a set of code fragments which are changed all together for the same reason

In the context of the SRP, a responsibility is defined as

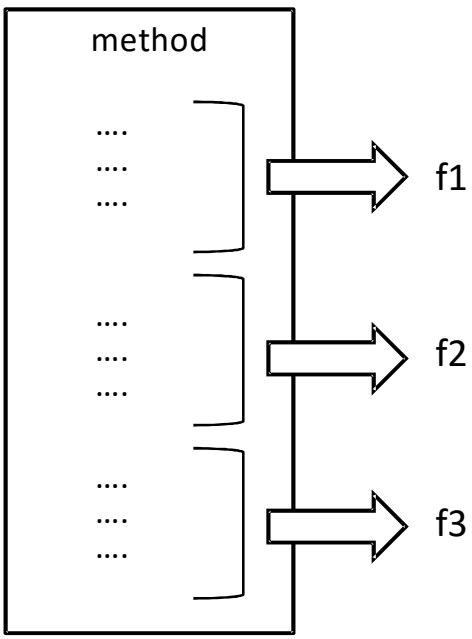
"a reason for change" or *"axis of change"*

If you can think of more than one motive for changing a class, you are violating the SRP.

Employee
doAsEmployee() genReport() saveToDB()

SRP-related Bad Smells

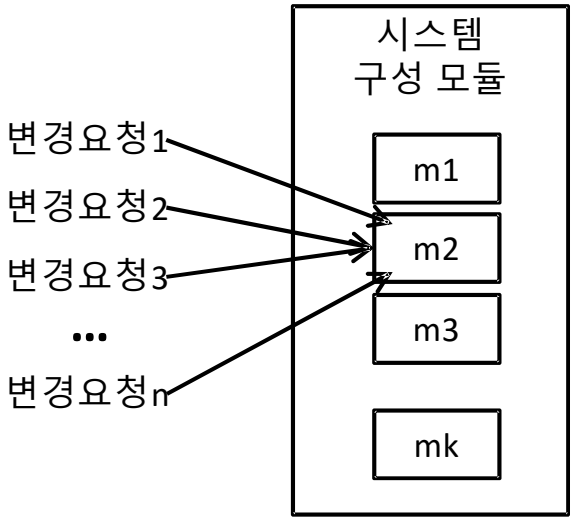
Long method: A long methods implements have more than one functions.



Method can be changed by three independent reasons.

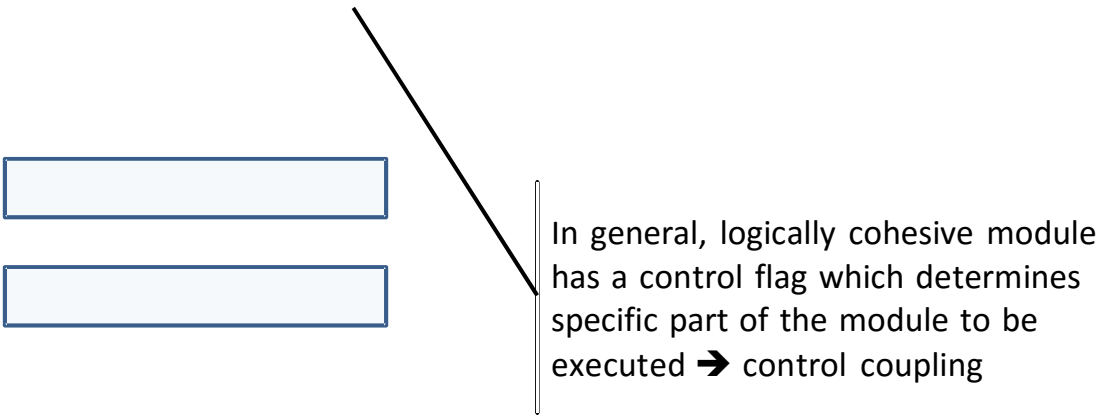
SRP-related Bad Smells

Divergent change: when one class is commonly changed in different ways for different reasons.



Related Concept: Logical Cohesion

Logical cohesion definitely violates SRP



Logically cohesive function has an ambiguous name or represents multiple purposes

Related Concept: Logical Cohesion

Logical cohesion definitely violates SRP

```
int sumAndProduct(int flag, int* values, int size) {
  int result = 0;
  for (unsigned int i = 0; i < size; i++) {
    if (flag == 0)
      result += values[i];
    else
      result *= values[i];
  }
  return result;
}
```

In general, logically cohesive module has a control flag which determines specific part of the module to be executed ➔ control coupling

Logically cohesive function has an ambiguous name or represents multiple purposes

Refactoring Techniques

Smell	Refactoring
A sequence of conditional tests	Consolidate conditional expression
Complicated conditional	Decompose conditional
Long method	Compose method
Long method	Replace temp with query
Poor cohesion	Separate query from modifier
Long method	Move accumulation to collecting parameter
Long method	Replace method with method object
Feature envy	Move method
Poor cohesion	Extract class
Poor cohesion	Extract subclass
Primitive obsession	Replace data value with object
Divergent change	Separate domain with presentation
Primitive obsession	Replace type code with class
embellishment	Decorator pattern
Hard-coded notification	Observer pattern

Consolidate Conditional Expression

You have a sequence of conditional tests with the same result

```
public double disabilityAmount() {
    if (seniority < 2)
        return 0;
    if (monthsDisabled > 12)
        return 0;
    if (isPartTime)
        return 0;

    // compute the disability amount …
}
```


Consolidate Conditional Expression

Combine them into a single conditional expression

It makes the check clearer

```
public double disabilityAmount() {
  if ( (seniority < 2) || (monthsDisabled > 12) || (isPartTime))
    return 0;

  // compute the disability amount
  ...
```

Continue with extracting method

```
public double disabilityAmount() {
  if ( isNotEligibleForDisability() ) return 0;
  // compute the disability amount
  ...
}
private boolean isNotEligibleForDisability() {
  return ((seniority < 2) || (monthsDisabled > 12) || (isPartTime));
}
```

Decompose Conditional

You have a complicated conditional statement

```
if ( date.before (SUMMER_START) || date.after(SUMMER_END) )
  charge = quantity * winterRate + winterServiceCharge;
else
  charge = quantity * summerRate;
```


Decompose Conditional

Extract methods from the condition, then part, and else parts

Make code clearer by decomposing it and replacing chunks of code with a method with meaningful name

```
if ( notSummer(date) )
    charge = winterCharge(quantity);
else
    charge = summerCharge(quantity);

private boolean notSummer(final Date date) {
  return date.before (SUMMER_START) || date.after(SUMMER_END);
}
private double summerCharge(final int quantity) {
  return quantity * summerRate;
}
private double winterCharge(final int quantity) {
  return quantity * winterRate + winterServiceCharge;
}
```

Compose Method

You can't rapidly understand a method's logic.

It consists of a number of concrete logics.

```
public void add(Object element) {
  if ( !readOnly) {
    int newSize = size + 1;
    if ( newSize > elements.length ) {
      Object[] newElements = new Object[elements.length+10];
      for ( int i=0; i < size; i++)
        newElements[i] = elements[i];
      elements = newElements ;
    }
    elements[size++] = element;
  }
}
```


Compose Method

Transform the logic into a small number of intention-revealing steps at the same level of detail

```
public void add(Object element) {
  if ( readOnly ) return; // guard clause
  if ( atCapacity() )
    grow();
  addElement(element);
}
private boolean atCapacity() {
  return (size + 1) > elements.length;
}
private void grow() {
  Object[] newElements =
    new Object[elements.length + GROWTH_INCREMENT]; //Constant
  for (int i = 0; i < size; i++)
    newElements[i] = elements[i];
  elements = newElements;
}
private void addElement(Object element) {
  elements[size++] = element;
}
```

Replace Temp with Query

You have a temporary variable to hold the result of expression.

Especially, the temporary variable is also defined in other methods ➔ **shotgun surgery**

```
public double getDiscountPrice() {
  double basePrice = quantity * itemPrice;
  if ( basePrice > 1000 )
    return basePrice * 0.95;
  else
    return basePrice * 0.98;
}

public double getPriceforVIP() {
  double basePrice = quantity * itemPrice;
  return basePrice * 0.7;
}
```


Replace Temp with Query

Extract the expression into a method so that it can be used in other methods

```
public double getDiscountPrice() {
  if ( getBasePrice() > 1000 )
    return getBasePrice() * 0.95;
  else
    return getBasePrice() * 0.98;
}

public double getPriceforVIP() {
  return getBasePrice() * 0.7;
}

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

Separate Query from Modifier

You have a method that returns a value but also changes the state of the object

```
private List<int> scores = new ArrayList<int>();

public long addAndGetSum(int score) {
  scores.Add(score);
  long sum = 0;
 for ( int i=0; i < scores.size(); i ++)
   sum += scores.get(i);
  return sum;
}
```


Separate Query from Modifier

Create two methods, one for the query and one for the modification

The query method can be reused independently.

```
public long getSum() {
  long sum = 0;
  for ( int i=0; i < scores.size(); i ++)
    sum += scores.get(i);
  return sum;
}

public void addScor (int score) {
  scores.Add(score);
}
```

Move Accumulation to Collecting Parameter

You have a single bulky method that accumulates information to a local variable

```
class TagNode...
  public String toString() {
    String result = new String();
    result += "<" + tagName + " " + attributes + ">";
    Iterator it = children.iterator();
    while (it.hasNext()) {
      TagNode node = (TagNode)it.next();
      result += node.toString();
    }
    if (!value.equals(""))
      result += value;
    result += "</" + tagName + ">";
    return result;
}
```


Move Accumulation to Collecting Parameter

Accumulate results to a Collecting Parameter that gets passed to extracted methods

```
public String toString() {
  StringBuffer result = new StringBuffer("");
  appendContentsTo(result);
  return result.toString();
}

private void appendContentsTo(StringBuffer result) {
  writeOpenTagTo(result); writeChildrenTo(result);
  writeValueTo(result); writeEndTagTo(result);
}
```

```
private void writeOpenTagTo(StringBuffer result) {
  result.append("<"); result.append(name);
  result.append(attributes.toString());
  result.append(">");
}

private void writeChildrenTo(StringBuffer result) {
  Iterator it = children.iterator();
  while (it.hasNext()) {
    TagNode node = (TagNode)it.next();
    node.appendContentsTo(result);
  }
}

private void writeValueTo(StringBuffer result) {
  if (!value.equals("")) result.append(value);
}

private void writeEndTagTo(StringBuffer result) {
  result.append("</"); result.append(name); result.append(">");
}
```


Replace Method with Method Object

You have a long method to which **Extract Method** cannot be applied

```
public class Account {
  gamma(int inputVal, int quantity, int yearToDate) {
    int importantValue1 = (inputVal * quantity) + delta();
    int importantValue2 = (inputVal * yearToDate) + 100;
    if ((yearToDate - importantValue1) > 100) { // Important thing
      importantValue2 -= 20;
    }
    int importantValue3 = importantValue2 * 7;
    // and so on…
    return importantValue3 - 2 * importantValue1;
  }
  // ...
```

Replace Method with Method Object

Turn the method into its own object so that all the local variables become fields on that object.

- You can then decompose the method into other methods on the same object

```
public class Account {
  public int gamma(int inputVal, int quantity, int yearToDate)
  {
    return new Gamma(this, inputVal, quantity, yearToDate).compute();
  }
  // ...
```



```
public class Gamma {
  private final Account account;

  private int importantValue1;
  private int importantValue2;
  private int importantValue3;

  private int inputVal; private int quantity; private int yearToDate;

  public Gamma(Account account, int inputVal,
               int quantity, int yearToDate) {
    this.account = account; this.inputVal = inputVal;
    this.quantity = quantity; this.yearToDate = yearToDate;
  }

  int compute() {
    importantValue1();
    importantValue2();
    importantValue3();
    // and so on…
    return importantValue3 - 2 * importantValue1;
  }
```

Move Method

Feature envy:
a method is, or will be, using, or used by more features of another class

```
public class Account {
   private int daysOverdrawn;
   private AccountType type;
   double overdraftCharge() {
      if ( type.isPremium() ) {
         double result = 10;
         if (daysOverdrawn > 7)
            result += (daysOverdrawn-7)*0.85;
         return result;
      }
      else return daysOverdrawn * 1.75;
   }
   double bankCharge() {
      double result = 4.5;
      if (daysOverdrawn > 0)
         result += overdraftCharge();
      return result;
   }
   ...
}
```


Move Method

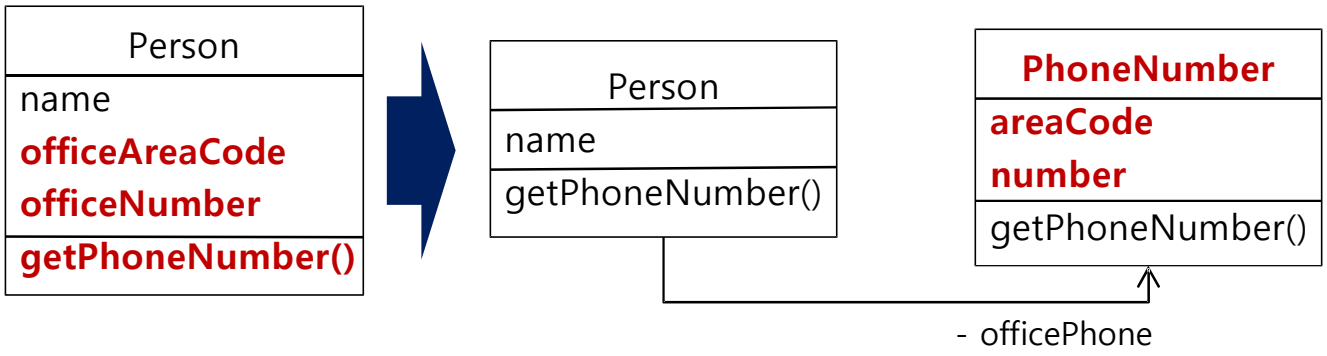
```
public class Account {
  private int daysOverdrawn;
  private  AccountType type;
  double bankCharge() {
  double result = 4.5;
  if (daysOverdrawn > 0)
    result += type.overdraftCharge(daysOverdrawn);
    return result;
  }
  ...
}
```

```
public class AccountType {
  double overdraftCharge(int daysOverdrawn) {
    if (isPremium()) {
      double result = 10;
      if(daysOverdrawn > 7)
        result += (daysOverdrawn - 7) * 0.85;
        return result;
      } else return daysOverdrawn * 1.75;
    }
  }
```

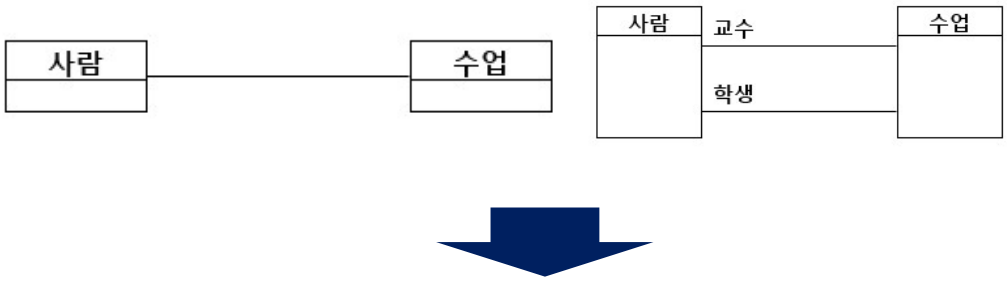
Extract Class

Large class: you have one class doing work that should be done by two

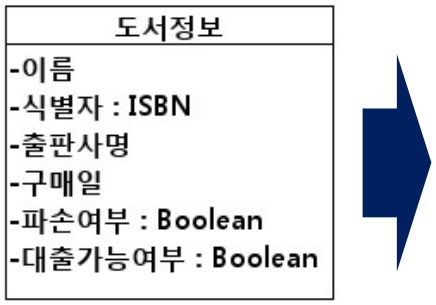
Create a new class and move the relevant fields and methods from the old class into the new class



More Examples for Extract Class



More Examples for Extract Class



Extract Subclass

A class has features that are used only in some instances

Create a subclass for that subset of features

```
class JobItem {
  private int quantity;  private int unitPrice;
  private  Employee employee; private boolean isLabor;

  JobItem(int quantity,int unitPrice,boolean isLabor,Employee employee)  {
    this.quantity =  quantity;  this.unitPrice =  unitPrice;
    this.isLabor =  isLabor;  this.employee =  employee;
  }

  int getTotalPrice()  { return quantity * getUnitPrice(); }
  int getQuantity()  { return quantity;  }
  int getUnitPrice()  {
    return  (isLabor) ? employee.getRate() : unitPrice; }
  Employee getEmployee() { return employee;  }
}
```

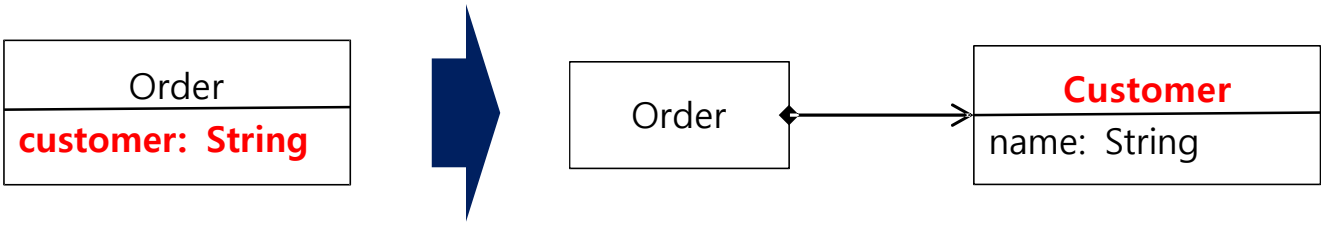
Extract Subclass

```
abstract class JobItem {
  private int quantity;
  public JobItem(int quantity) { this.quantity = quantity; }
  public int getTotalPrice() { return quantity * getUnitPrice(); }
  public int getQuantity() { return quantity; }
  abstract public int getUnitPrice();
}
class LaborItem extends JobItem {
  private Employee employee;
  public LaborItem(int quantity, Employee employee) {
    super(quantity); this.employee = employee;
  }
  public Employee getEmployee() { return employee; }
  public int getUnitPrice() { return employee.getRate(); }
}
class PartsItem extends JobItem {
  private int unitPrice;
  public PartsItem(int quantity, int unitPrice) {
    super(quantity); this.unitPrice = unitPrice;
  }
  public int getUnitPrice() { return unitPrice; }
}
```

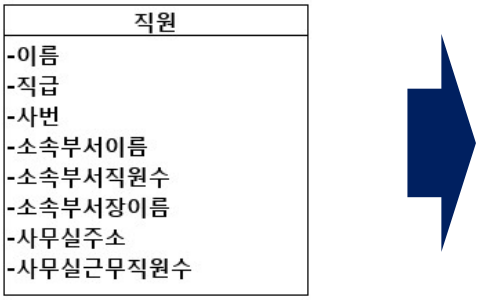

Replace Data Value with Object

Primitive obsession: you have a data item that needs additional data or behavior

Turn the data item into an object



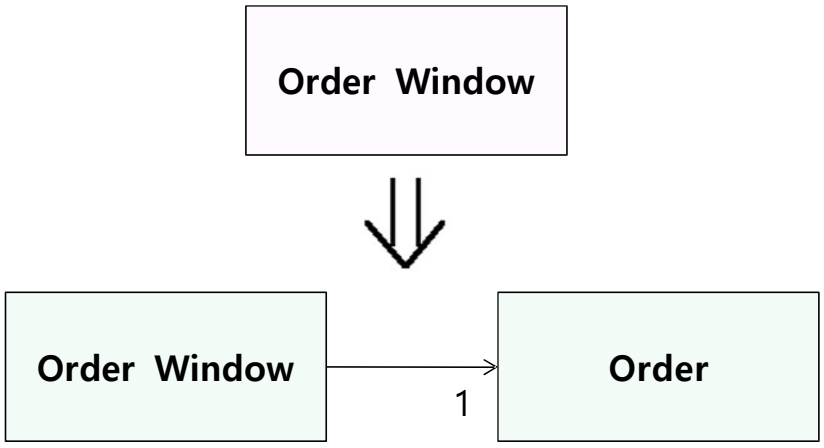
Replace Data Value with Object



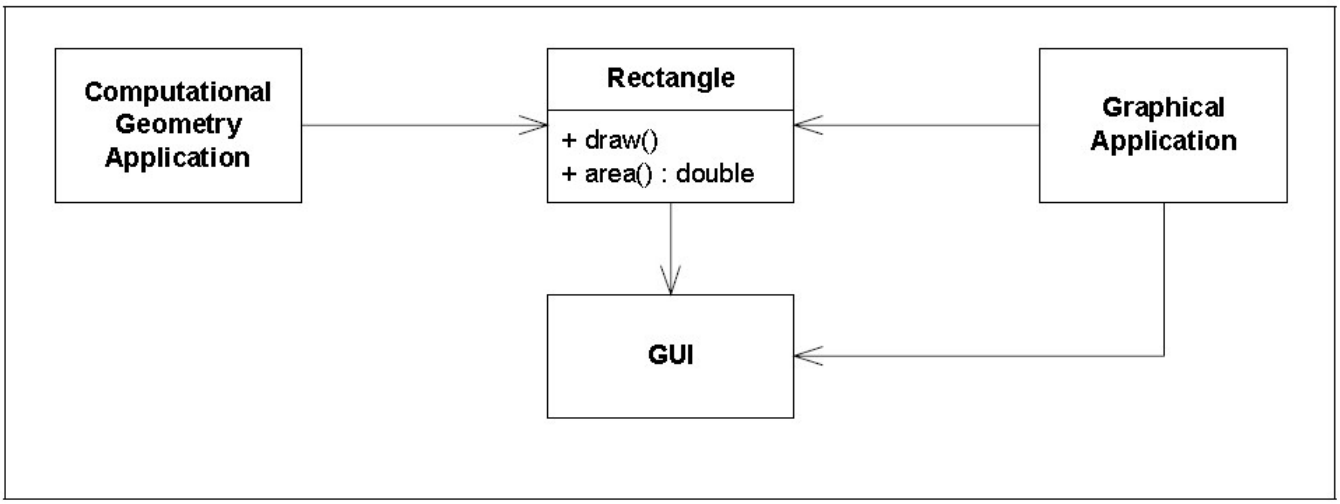
Separate Domain From Presentation

Divergent Change: you have GUI classes that contain domain logic

Separate the domain logic into separate domain classes



Separate Domain From Presentation

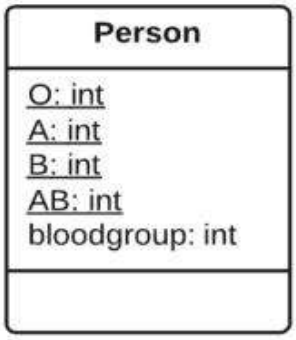


Separate Domain From Presentation

Replace Type Code with Class

Primitive Obsession: a class has a numeric type code that does not affect its behavior

Replace the number with a new class

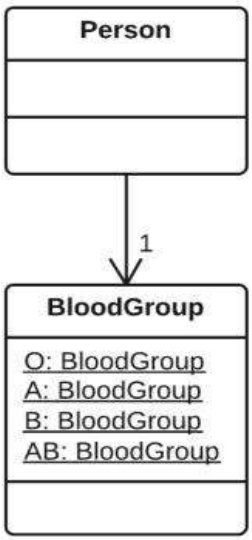



```
class Person{
   public static final int O = 0;
   public static final int A = 1;
   public static final int B = 2;
   public static final int AB = 3;
   private int bloodGroup;
   public Person(int bloodGroup) { bloodGroup = bloodGroup; }
   public void setBloodGroup(int arg) { bloodGroup=arg; }
}
```


Replace Type Code with Class

```
class Person{
   private BloodGroup bloodGroup;
   public Person(BloodGroup arg){
      bloodGroup = arg;
   }
   public void setBloodGroup(BloodGroup arg) {
      bloodGroup=arg;
   }
   ...
}

class BloodGroup {
   public static final BloodGroup O = new BloodGroup(0);
   public static final BloodGroup A = new BloodGroup(1);
   public static final BloodGroup B = new BloodGroup(2);
   public static final BloodGroup AB = new BloodGroup(3);
   private final int code;
   private BloodGroup(int code) { this.code = code; }
}
```



SRP-Related Patterns

- Observer Pattern
- Decorator Pattern