

## Question

## Interfaces (suite)

### Programmation Orientée Objet

Jean-Christophe Routier  
Licence mention Informatique  
Université Lille 1



Peut-on obtenir des comportements différents pour un même envoi de message ?

c-à-d

un envoi de message peut-il entrainer différentes invocations de méthode ?

Université Lille 1 - Licence Informatique						Université Lille 1 - Licence Informatique					
Programmation Orientée Objet						Programmation Orientée Objet					
Tris	NON	OUI	OCP	Compteur	Abstraction	Tris	NON	OUI	OCP	Compteur	Abstraction
000000000000	00000000	000000	000000	0000000	00	●000000000000	00000000	000000	000000	0000000	00

## Pourquoi ? (exemples)

- disposer d'une méthode **générique** de tri d'un tableau (ie. sans imposer la nature des éléments). Il faut pouvoir :
  - typer les éléments du tableau
  - comparer deux éléments : le traitement associé à cette comparaison va donc dépendre de la classe des éléments
- disposer d'un outil de manipulation d'images de **différents** formats (jpg, gif, bmp, etc.).  
les traitements dépendent du format de l'image (car de son codage), on veut pourtant manipuler les images d'une manière **identique**, et pouvoir ajouter d'autres formats d'images.
- disposer d'un framework graphique "**extensible**" (pour des « compteurs » par exemple)

### Tri **croissant** d'un tableau d'entiers.

cf. portail, correction TD 1

```
public void ascendingSort(int[] elements) {
    int tmp;
    for (i = elements.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elements[j] > elements[j+1]) {
                tmp = elements[j];
                elements[j] = elements[j+1];
                elements[j+1] = tmp;
            }
        } // for j
    } // for i
}
```

// mal triés ? on échange

Tri **décroissant** d'un tableau d'entiers.

```
public void descendingSort(int[] elements) {
    int tmp;
    for (i = elements.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elements[j] < elements[j+1]) {           // mal triés ?  on échange
                tmp = elements[j];
                elements[j] = elements[j+1];
                elements[j+1] = tmp;
            }
        } // for j
    } // for i
}
```

Tri alphabétique **croissant** d'un tableau de chaînes.

```
public void ascendingSort(String[] elements) {
    String tmp;
    for (i = elements.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elements[j] greater than elements[j+1]) { // mal triés ?  on échange
                tmp = elements[j];
                elements[j] = elements[j+1];
                elements[j+1] = tmp;
            }
        } // for j
    } // for i
}
```

Université Lille 1 - Licence Informatique						5	Université Lille 1 - Licence Informatique						6
Tris	NON	OUI	OCP	Compteur	Abstraction		Tris	NON	OUI	OCP	Compteur	Abstraction	
○○○●○○○○○○○	○○○○○○○○	○○○○○○	○○○○○○	○○○○○○	○○		○○○○●○○○○○○	○○○○○○○○	○○○○○○	○○○○○○	○○○○○○	○○	

Tri alphabétique **croissant** d'un tableau de chaînes.

```
public void ascendingSort(String[] elements) {
    String tmp;
    for (i = elements.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elements[j].compareTo(elements[j+1]) > 0) { // mal triés ?  on échange
                tmp = elements[j];
                elements[j] = elements[j+1];
                elements[j+1] = tmp;
            }
        } // for j
    } // for i
}
```

Tri **croissant** selon le poids d'un tableau de carottes.

```
public void ascendingSort(Carrot[] elements) {
    Carrot tmp;
    for (i = elements.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elements[j] heavier than elements[j+1]) { // mal triés ?  on échange
                tmp = elements[j];
                elements[j] = elements[j+1];
                elements[j+1] = tmp;
            }
        } // for j
    } // for i
}
```

```

public class Carrot {
    private int weight;
    ...
    public int compareTo(Carrot carrot) {
        return this.weight - carrot.weight;
    }
}

public void ascendingSort(Carrot[] elements) {
    Carrot tmp;
    for (i = elements.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elements[j] heavier than elements[j+1] ) { // mal triés ? on échange
                tmp = elements[j];
                elements[j] = elements[j+1];
                elements[j+1] = tmp;
            }
        } // for j
    } // for i
}

```

```

public class Carrot {
    private int weight;
    ...
    public int compareTo(Carrot carrot) {
        return this.weight - carrot.weight;
    }
}

public void ascendingSort(Carrot[] elements) {
    Carrot tmp;
    for (i = elements.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elements[j].compareTo(elements[j+1]) > 0 ) { // mal triés ? on échange
                tmp = elements[j];
                elements[j] = elements[j+1];
                elements[j+1] = tmp;
            }
        } // for j
    } // for i
}

```

```

public class Carrot implements Comparable<Carrot> {
    private int weight;
    ...
    public int compareTo(Carrot carrot) {
        return this.weight - carrot.weight;
    }
}

public void ascendingSort(Carrot[] elements) {
    Carrot tmp;
    for (i = elements.length-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elements[j].compareTo(elements[j+1]) > 0 ) { // mal triés ? on échange
                tmp = elements[j];
                elements[j] = elements[j+1];
                elements[j+1] = tmp;
            }
        } // for j
    } // for i
}

```

- Les objets du tableau doivent être comparables 2 à 2  
C'est la seule condition pour le "typage" : type **Comparable**

```

public interface Comparable<T> {
    public int compareTo(T o);
}

public class BubbleSort {
    public void sort(Comparable<?>[] elements) {
        Comparable tmp;
        for(int i = elements.length-1; i > 0; i--) {
            for(int j = 0; j < i; j++) {
                if (elements[j].compareTo(elements[j+1]) < 0) {
                    tmp = elements[j];
                    elements[j] = elements[j+1];
                    elements[j+1] = tmp;
                }
            }
        }
    }
    public void display(Comparable<?>[] t) { ... // TO DO }
}

```

```
// ===== utilisation =====
BubbleSort bubble = new BubbleSort();
String[] elts = new String[]{"bacd", "abcd", "aabc", "abda"};
bubble.display(elts); // (1)
bubble.sort(elts);
bubble.display(eltds); // (2)

Carrot[] karrots = new Carrot[] { ... };
bubble.sort(karrots);
bubble.display(karrots);
```

- L'utilisation d'interface permet la définition d'un algorithme générique
- On identifie l'abstraction pertinente  
ici : *être comparables 2 à 2*
- On l'exprime par une interface qu'il "suffit" d'implémenter

```
+---trace-----
| bacd abcd aabc abda // (1)
|
| aabc abcd abda bacd // (2)
```

## Manipulation d'images

## Une seule classe

- application "ImageManipulator" de manipulation d'images de **différents** formats (jpg, gif, bmp, etc.).
- pour une image on veut pouvoir connaître :
  - sa dimension (largeur× hauteur)
  - la valeur du pixel à des coordonnées données
  - sauvegarder/charger l'image vers/depuis un fichier
- les traitements **différent** selon le format de l'image (car codages différents)
- on veut pourtant manipuler les images d'une manière **identique**, et pouvoir ajouter d'autres formats d'images.

### Proposition 1

avoir une seule classe `Image` et distinguer par un attribut "type" les différents formats d'image.

Ce qu'il NE FAUT PAS écrire

Commentaires

```
public class Image {
    private String type; // ou avec une énumération ImageType
    public Image(String type) { // constructeur
        this.type = type;
        ...
    }
    public void save() {
        if (this.type.equals("jpg")) {
            ...CODE A // traitement pour sauvegarder une image au format jpg
        }
        else if (this.type.equals("gif")) {
            ...CODE B // traitement pour sauvegarder une image au format gif
        }
        else if (this.type.equals("bitmap")) {
            ...CODE C // traitement pour sauvegarder image au format bitmap
        }
    }
    ... } // fin Image
}

public class ImageManipulator {
    ...
    public void saveImage(Image img) { img.save(); }
}

// utilisation
ImageManipulator manipulator = new ImageManipulator();
Image img1 = new Image("gif");
Image img2 = new Image("jpg");
manipulator.saveImage(img1); // "ça marche" :
manipulator.saveImage(img2); // même manipulation
                             // pour les 2 formats
```

NON !

Question

Quelles sont les conséquences de ce choix ?

- au niveau du code à écrire
- au niveau de la conception

- nécessité de séparer les traitements particuliers à un format par des tests sur type  
 ↳ pour chaque méthode “spécialisée” du format
  - getWidth(), getHeight(), load(), save(), etc.
- problème lors de l’ajout d’un nouveau type : chaque méthode doit être modifiée  
 ↳ **difficulté d’extension**
- + problème de l’attribut représentant les données de l’image : il faut choisir une représentation commune pour tous les formats.

Pour des comportements différents il faut des classes différentes

Proposition 2

avoir 3 classes

JpgImage

GifImage

BitmapImage

avec chacune leur “version” des méthodes

getWidth(), getHeight(), load(), save(), etc.

chaque classe d’image définit la méthode public void save()

```
public class JpgImage implements Image {
    public void save() {
        ...CODE A // traitement pour sauvegarder l'une image au format jpg
    }
    ...
}

public class GifImage implements Image {
    public void save() {
        ...CODE B // traitement pour sauvegarder l'image au format gif
    }
    ...
}
```

Question

Que devient la méthode saveImage de ImageManipulator ?

Ce qu'il NE FAUT PAS écrire NON PLUS

Constat

```
public class ImageManipulator {
    ...
    public void saveImage(Object img) throws NotAnImageExpection {
        if (img instanceof JpgImage) {
            ((JpgImage) img).save();
        }
        else if (img instanceof GifImage) {
            ((GifImage) img).save();
        }
        else if (img instanceof BitmapImage) {
            ((BitmapImage) img).save();
        }
        else throw new NotAnImageExpection();
    }
    ...
}

// utilisation
ImageManipulator manipulator = new ImageManipulator();
JpgImage img1 = new JpgImage(...);
GifImage img2 = new GifImage(...);
manipulator.saveImage(img1);
manipulator.saveImage(img2);
```

NON !

Question

Quelles sont les conséquences de ce choix ?

- au niveau du code à écrire
- au niveau de la conception

- difficulté de l'ajout d'une nouvelle classe d'images  
 ↪ nécessite modification de chaque méthode de ImageManipulator
  - l'argument de saveImage est de type Object  
 ↪ pas de détection à la compilation si l'argument de saveImage n'est pas une instance d'une "classe image"
- ```
ImageManipulator manipulator = new ImageManipulator();
// accepté à la compilation :
manipulator.saveImage(new String("timoleon"));
```
- ⇒ on perd le typage

| Université Lille 1 - Licence Informatique |              |           |        |         |                   | Université Lille 1 - Licence Informatique |              |           |         |         |                   |
|-------------------------------------------|--------------|-----------|--------|---------|-------------------|-------------------------------------------|--------------|-----------|---------|---------|-------------------|
| Programmation Orientée Objet              |              |           |        |         |                   | Programmation Orientée Objet              |              |           |         |         |                   |
| 21                                        | Tris         | NON       | OUI    | OCP     | Compteur          | 22                                        | Tris         | NON       | OUI     | OCP     | Compteur          |
|                                           | ○○○○○○○○○○○○ | ○○○○○○○○○ | ●○○○○○ | ○○○○○○○ | ○○○○○○○           |                                           | ○○○○○○○○○○○○ | ○○○○○○○○○ | ○●○○○○○ | ○○○○○○○ | ○○○○○○○           |
|                                           |              |           |        |         | Abstraction<br>○○ |                                           |              |           |         |         | Abstraction<br>○○ |

Conclusion ?

La solution

Question

Quelle(s) conclusion(s) tirer de ces deux (mauvaises) approches ?

Il faut **mixer** les deux approches :

- Il faut un type Image **commun**
- Il faut des classes **différentes** pour chaque format d'image

Pour obtenir des traitements différents avec une même invocation de méthode, il faut avoir des objets :

- de **même** type
- de classes **différentes**

La solution ?

les **INTERFACES**

- fixent les messages acceptés/autorisés
- le comportement doit être implémenté **séparément** par des classes

## Ce qu'il FAUT écrire

```
public interface Image {
    public void save();
    ... // et d'autres comme public int getWidth();
}
public class JpgImage implements Image {
    public void save() {
        ...CODE A // traitement pour sauvegarder l'une image au format jpg
    }
    ... // dont les autres méthodes de l'interface s'il y en a
}
public class GifImage implements Image {
    public void save() {
        ...CODE B // traitement pour sauvegarder l'image au format gif
    }
}

public class ImageManipulator {
    public void saveImage(Image img) { img.save(); }
}

// utilisation
ImageManipulator manipulator = new ImageManipulator();
Image img1 = new JpgImage();
Image img2 = new GifImage();
manipulator.saveImage(img1);
manipulator.saveImage(img2);
```

OUI !

## Ajout d'une nouvelle classe d'images ?

Que faut-il faire ?

Il suffit de définir une classe implémentant l'interface Image

```
public class BitmapImage implements Image {
    public void save() {
        ... // traitement pour sauvegarder une image au format bitmap
    }
}

... utilisation
ImageManipulator manipulator = new ImageManipulator();
manipulator.saveImage(new BitmapImage()); // même manipulation sans autre changement
```

| Université Lille 1 - Licence Informatique |                  |                |                |                     |                   | 25 | Université Lille 1 - Licence Informatique |                  |                |                |                     |                   | 26 |
|-------------------------------------------|------------------|----------------|----------------|---------------------|-------------------|----|-------------------------------------------|------------------|----------------|----------------|---------------------|-------------------|----|
| Tris<br>○○○○○○○○○○○○                      | NON<br>○○○○○○○○○ | OUI<br>○○○●○○○ | OCP<br>○○○○○○○ | Compteur<br>○○○○○○○ | Abstraction<br>○○ |    | Tris<br>○○○○○○○○○○○○                      | NON<br>○○○○○○○○○ | OUI<br>○○○●○○○ | OCP<br>○○○○○○○ | Compteur<br>○○○○○○○ | Abstraction<br>○○ |    |

## Polymorphisme

Un objet est toujours instance d'une classe

- Si une classe implémente une interface une instance de cette classe peut être vue comme du type de l'interface et manipulée comme telle.

### Polymorphisme

un objet est du type :

- de sa classe
- des interfaces implémentées par sa classe

- On peut déclarer une référence comme étant du type d'une interface ⇒ on n'autorise sur cette référence que les envois de messages définis par l'interface

MAIS

- on initialise la référence par un objet qui est **instance d'une classe**
- cette classe doit implémenter l'interface.
- on **ne peut pas** invoquer sur la référence les méthodes définies par la classe mais **pas** par l'interface.

La référence ne donne accès qu'à la partie de l'objet **restreinte** à ce qui est défini par l'interface.

Open Closed Principle

Remarque méthodologique

Open Closed Principle

Un module doit être ouvert aux extensions mais fermé aux modifications

- À l'extrême toujours commencer par définir des interfaces et seulement ensuite les classes les implémentant
- “manipuler des abstractions et concrétiser le plus tard possible”

- se méfier lorsque l'on voit apparaître des traitements de méthodes avec des cascades de “si *condition* alors ... sinon ...” avec *condition* qui porte sur une “**condition de typage**”
- dans ce cas :
  - identifier le(s) point(s) commun(s) entre les objets a priori toutes les méthodes concernées
  - créer une interface regroupant ces méthodes
  - créer une classe pour chacun des “types d'objet” et lui faire implémenter l'interface et fixer le code des méthodes

| Université Lille 1 - Licence Informatique |                 |               |               |                     |                   | Université Lille 1 - Licence Informatique |                 |               |               |                     |                   |
|-------------------------------------------|-----------------|---------------|---------------|---------------------|-------------------|-------------------------------------------|-----------------|---------------|---------------|---------------------|-------------------|
| Programmation Orientée Objet              |                 |               | 29            |                     |                   | Programmation Orientée Objet              |                 |               | 30            |                     |                   |
| Tris<br>○○○○○○○○○○○○                      | NON<br>○○○○○○○○ | OUI<br>○○○○○○ | OCP<br>○○●○○○ | Compteur<br>○○○○○○○ | Abstraction<br>○○ | Tris<br>○○○○○○○○○○○○                      | NON<br>○○○○○○○○ | OUI<br>○○○○○○ | OCP<br>○○○●○○ | Compteur<br>○○○○○○○ | Abstraction<br>○○ |

| ClasseUn                     |
|------------------------------|
| +fooUn()<br>+barUn()<br>+f() |

| ClasseDeux                       |
|----------------------------------|
| +fooDeux()<br>+barDeux()<br>+g() |

| ClasseTrois                        |
|------------------------------------|
| +fooTrois()<br>+barTrois()<br>+h() |

```
public class Utilisatrice {
    public void appelFoo(Object o) {
        if (o instanceof ClasseUn) {
            ((ClasseUn) o).fooUn();
        } else if (o instanceof ClasseDeux) {
            ((ClasseDeux) o).fooDeux();
        } else if (o instanceof ClasseTrois) {
            ((ClasseTrois) o).fooTrois();
        }
    }
    public void appelBar(Object o) {
        if (o instanceof ClasseUn) {
            ((ClasseUn) o).barUn();
        } else if (o instanceof ClasseDeux) {
            ((ClasseDeux) o).barDeux();
        } else if (o instanceof ClasseTrois) {
            ((ClasseTrois) o).barTrois();
        }
    }
    public void appelF(ClasseUn c1) { c1.f(); }
    public void appelG(ClasseDeux c2) { c2.g(); }
}
```

## Factoriser les types : créer une interface

```
public interface MonInterface {
    public void foo();
    public void bar();
}

public class ClasseUn implements MonInterface {
    public void foo() { même code que fooUn } // remplace fooUn
    public void bar() { même code que barUn } // remplace barUn
    public void f() { ... } // inchangée
}
public class ClasseDeux implements MonInterface { ... } // similaire
public class ClasseTrois implements MonInterface { ... } // similaire

public class Utilisatrice {
    public void appelFoo(MonInterface ref) {
        ref.foo();
    }
    public void appelBar(MonInterface ref) {
        ref.bar();
    }
    public void appelF(ClasseUn c1) { c1.f(); }
    public void appelG(ClasseDeux c2) { c2.g(); }
}
```



3 “types” et 3 comportements ? (bid = 1/2/3)  
alors 3 classes et créer une interface pour **factoriser** le type

```
public class Timoleon {
    private int bidule; // ou un String, un char, un type énuméré ...
    public Timoleon(int b) { this.bidule = b; }
    public int getBidule() { return this.bidule; }
}

public class Utilisatrice {
    public void appelFoo(Timoleon t) {
        int bid = t.getBidule();
        if (bid == 1) {
            bloc de code 1 pour foo;
        } else if (bid == 2) {
            bloc de code 2 pour foo;
        } else if (bid == 3) {
            bloc de code 3 pour foo;
        }
    }
    public void appelBar(Timoleon t) {
        switch(t.getBidule()) {
            case 1:
                bloc de code 1 pour bar;
                break;
            case 2:
                bloc de code 2 pour bar;
                break;
            case 3:
                bloc de code 3 pour bar;
                break;
        }
    }
}
```

```
public interface TimoleonInterface {
    public void foo();
    public void bar();
}

public class TimoleonUn implements TimoleonInterface {
    public void foo() { bloc de code 1 pour foo }
    public void bar() { bloc de code 1 pour bar }
}

public class TimoleonDeux implements TimoleonInterface {
    public void foo() { bloc de code 2 pour foo }
    public void bar() { bloc de code 2 pour bar }
}

public class TimoleonTrois implements TimoleonInterface {
    public void foo() { bloc de code 3 pour foo }
    public void bar() { bloc de code 3 pour bar }
}

public class Utilisatrice {
    public void appelFoo(TimoleonInterface tim) {
        tim.foo();
    }
    public void appelBar(TimoleonInterface tim) {
        tim.bar();
    }
}
```

| Université Lille 1 - Licence Informatique |           |         |         |          |             | 33 | Université Lille 1 - Licence Informatique |           |         |         |          |             | 34 |
|-------------------------------------------|-----------|---------|---------|----------|-------------|----|-------------------------------------------|-----------|---------|---------|----------|-------------|----|
| Tris                                      | NON       | OUI     | OCP     | Compteur | Abstraction |    | Tris                                      | NON       | OUI     | OCP     | Compteur | Abstraction |    |
| ○○○○○○○○○○○○                              | ○○○○○○○○○ | ○○○○○○○ | ○○○○○○○ | ●○○○○○○○ | ○○          |    | ○○○○○○○○○○○○                              | ○○○○○○○○○ | ○○○○○○○ | ○○○○○○○ | ●○○○○○○○ | ○○          |    |

## Compteur

Un compteur est “quelque chose” :

- qui a une valeur et qui peut passer à la valeur suivante (“incrémenter”)

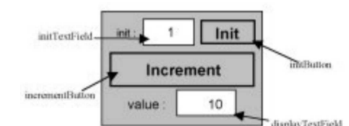
il existe **différents** types de compteur

- cyclique, borné, à “incrément” variable, etc.

pour chacun de ces types de compteurs le traitement est différent, mais la manipulation est identique

- on souhaite disposer d'une classe pour gérer l'interface graphique de tels compteurs
- l'interface ne dépend a priori pas du type du compteur

```
public class CounterGUI { // très approximatif !!!
    private Counter counter;
    ...
    public CounterGUI(Counter counter) {
        ... // initialisation de la partie graphique
        this.counter = counter;
    }
    public void initButtonAction() {
        int value = Integer.parseInt(initTextField.getText());
        counter.initValue(value);
        displayCounter();
    }
    public void incrementButtonAction() {
        counter.increment();
        displayCounter();
    }
    public void displayCounter() {
        displayTextField.setText(""+counter.getCurrentValue());
    }
}
```



Il faut définir le type Counter

# Abstraction de la notion de Compteur

- il faut un type Counter
- les objets de ce type doivent accepter les messages :
  - public int `getCurrentValue()`
  - public void `increment()`
  - public void `initValue(int init)`

d'où l'interface

```
public interface Counter {
    public int getCurrentValue();
    public void increment();
    public void initValue(int init);
}
```

on peut alors créer les classes :

```
public class SimpleCounter implements Counter {
    private int value;
    public SimpleCounter(int value) { this.value = value; }
    public int getCurrentValue() { return this.value; }
    public void increment() { this.value++; }
    public void initValue(int init) { this.value = init; }
}

public class ModularCounter implements Counter {
    private int value;
    private int modulo;
    public SimpleCounter(int value, int modulo) {
        this.value = value; this.modulo = modulo;
    }
    public int getCurrentValue() { return this.value; }
    public void increment() { this.value = (this.value+1) % modulo; }
    public void initValue(int init) { this.value = init; }
}

// ... utilisation

Counter simpleCounter = new SimpleCounter(0);           // upcast
Counter modularCounter = new ModularCounter(0,7);       // vers Counter

new CounterGUI(simpleCounter);
new CounterGUI(modularCounter);
```

l'ajout d'une nouvelle classe de compteur est immédiat et naturel

respect de l'Open Closed Principle

```
public class AnotherCounter implements Counter {
    private int value;
    public SimpleCounter(int value) { this.value = value; }
    public int getCurrentValue() { return this.value; }
    public void increment() { this.value = 2*this.value + 1; }
    public void initValue(int init) { this.value = init; }
}

// ... utilisation

Counter anotherCounter = new AnotherCounter(0);           // upcast vers Counter

new CounterGUI(anotherCounter);
```

## Autre version : abstraction de la notion d'incrément

- Répétition du code des méthodes `getCurrentValue` et `initValue`
- car en fait l'abstraction se situe au niveau de la manière d'incrémenter...

```
public interface IncrementFunction {
    public int increment(int value);
}

public class SimpleIncrement implements IncrementFunction {
    public int increment(int value) { return value++; }
}

public class ModularIncrement implements IncrementFunction {
    private int modulo;
    public ModularIncrement (int modulo) { this.modulo = modulo; }
    public int increment(int value) { return (value+1) % modulo; }
}

public class AnotherIncrement implements IncrementFunction {
    public int increment(int value) { return 2*value+1; }
}
```

## Interface = Abstraction

Les interfaces sont des **types**

- **fixent** des signatures des méthodes **sans imposer** le comportement associé,
- permettent une vision **polymorphe** sur les objets et facilitent la **généricité** (notion de “*template*”),
- permettent d'**offrir** aux autres un cadre de programmation  
→ concepteur de *framework*
- permettent de **réutiliser** des classes et de les **adapter** à un contexte  
→ utilisateur de *framework*
- facilitent **l'extension** d'un programme (l'ajout de comportements) **sans modification** de l'existant.

```
public class Counter {
    private int value;
    private IncrementFunction incrementF;
    public SimpleCounter(int value, IncrementFunction incrementF) {
        this.value = value;
        this.incrementF = incrementF;
    }
    public int getCurrentValue() { return this.value; }
    public void increment() { this.value = incrementF.increment(this.value); }
    public void initValue(int init) { this.value = init; }
}

// ... utilisation

Counter simpleCounter = new Counter(0, new SimpleIncrement()); // upcast
Counter modularCounter = new Counter(0, new ModularIncrement(7)); // vers
Counter anotherCounter = new Counter(0, new AnotherIncrement()); //IncrementFunction

new CounterGUI(simpleCounter);
new CounterGUI(modularCounter);
new CounterGUI(anotherCounter);
```

## Faire varier le comportement

- 1 identifier l'abstraction à manipuler,
- 2 définir une interface caractérisant cette abstraction, c-à-d définir les signatures des méthodes qui y sont liées,
- 3 effectuer les manipulations de l'abstraction sur des références ayant pour type cette interface,
- 4 concrétiser cette interface par différentes classes définissant les différents comportements souhaités,
- 5 utiliser le polymorphisme pour initialiser les références du type de l'interface par des instances des classes l'implémentant