



Département informatique
Module : Conception et design pattern

Partie II: design pattern

Dr. Yassine RHAZALI
Master spécialisé: Génie logiciel pour
le Cloud

Définition

- Un patron de conception (*Design Pattern* : *DP*) est une solution à un problème fréquent dans la conception d'applications orientées objet.
- Un patron de conception décrit alors la solution éprouvée pour résoudre ce problème d'architecture de logiciel.
- **Note :**
 - Les Design Patterns sont indépendants des langages de programmation utilisés.

Définition...

- Les cas les plus généraux sont référencés dans un ouvrage considéré comme majeur : Design Patterns, de Gamma et al. (appelé aussi le Gang of Four - GoF / Bande des Quatre)
- Le premier Design Pattern connu est le MVC Modèle-Vue-Contrôleur et qui permet d'obtenir un couplage faible entre le cœur fonctionnel d'une application et son interface.

Représentation d'un DP

- Les Design Patterns sont représentés par :
 - **Nom** : qui permet de l'identifier clairement
 - **Problématique** : description du problème auquel il répond
 - **Solution** : description de la solution souvent accompagnée d'un schéma UML
 - **Conséquences** : les avantages et les inconvénients de cette solution

Les avantages

- Permet de répondre à un problème de conception grâce à une solution éprouvée et validée par des experts.
→ gain en rapidité et en qualité de conception ce qui diminue également les coûts.
- Les DP étant largement documentés et connus d'un grand nombre de développeurs ils permettent également de faciliter la communication.
 - Si un développeur annonce que sur ce point du projet il va utiliser le Design Pattern Observateur il est compris des informaticiens sans pour autant rentrer dans les détails de la conception (diagramme UML, objectif visé...).

Organisation des patrons de conception

- Les patrons de conception sont classés en trois catégories :
 - **Création** : ils permettent d'instancier et de configurer des classes et des objets.
 - **Structure** : ils permettent d'organiser les classes d'une application.
 - **Comportement** : ils permettent d'organiser les objets pour qu'ils collaborent entre eux.

DP de Cration (construction)

- Les DP de Cration/construction : se préoccupent de régler des problèmes liés à la cration ou l’instanciation des objets :
 - Abstract Factory,
 - Builder,
 - Factory Method,
 - Prototype,
 - Singleton

DP de Structure

- **Les DP de Structure** : s'occupent des questions relatives au découplage des concepts et à l'organisation structurelle des concepts manipulés par un logiciel.
 - **Adapter**,
 - **Bridge**,
 - **Composite**,
 - **Decorator**,
 - **Facade**,
 - **Flyweight**,
 - **Proxy**

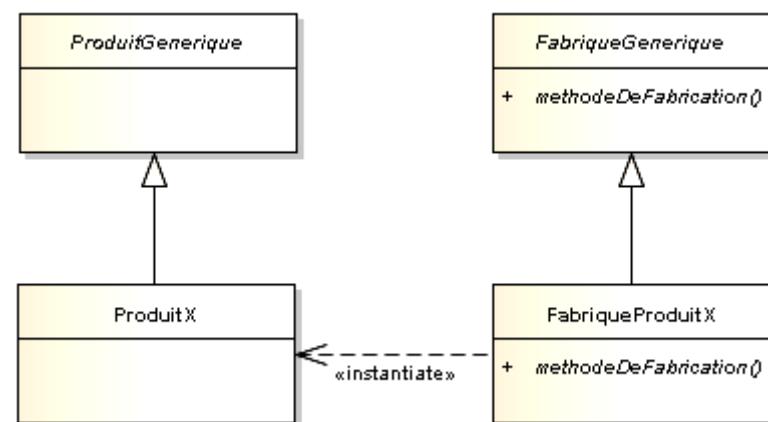
DP de Comportement

- Les DP de Comportement : ont pour sujet principal l'organisation de la collaboration des objets.
 - Observer,
 - Strategy,
 - Callback,
 - Chain of Responsibility,
 - Command,
 - Interpreter,
 - Iterator,
 - Mediator,
 - Memento,
 - State,
 - Template Method,
 - Visitor

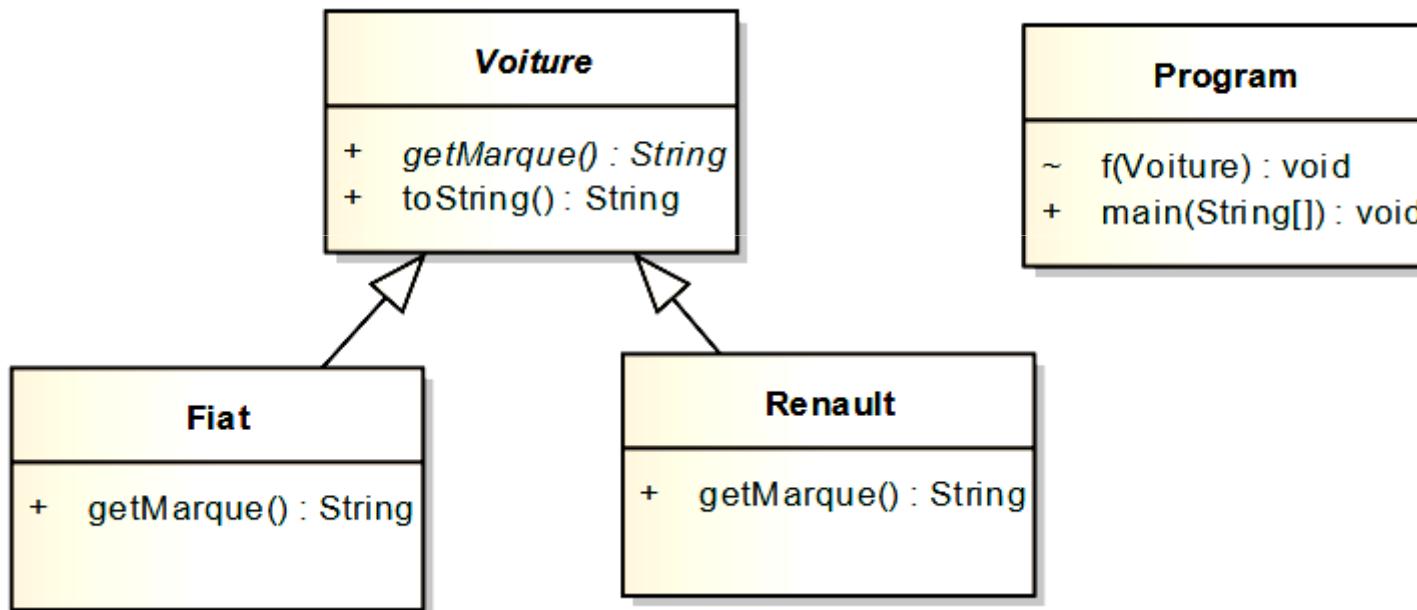
CHAPITRE I: PATRON DE CONCEPTION DE CRÉATION (*CREATIONAL DP*)

FACTORY METHOD

- La **fabrique** (*factory method*) est un patron de conception créational.
- Elle permet d'instancier des objets dont le type est dérivé d'un type abstrait.
- La classe exacte de l'objet n'est donc pas connue par l'appelant.



FACTORY METHOD : DC



FACTORY METHOD (1) : Java

```
abstract class Voiture {
    public abstract String getMarque();
    public String toString() {
        return getMarque();
    }
}

public class Renault extends Voiture {
    @Override
    public String getMarque() {
        return "Renault";
    }
}

public class Fiat extends Voiture {
    @Override
    public String getMarque() {
        return "Fiat";
    }
}
```

FACTORY METHOD (1) : Java

```
public class Program {  
    static void f(Voiture v) {  
        System.out.println("Voiture " + v);  
    }  
  
    public static void main(String[] args) {  
        Voiture v = new Renault();  
        f(v);  
    }  
}
```

Voiture Renault

FACTORY METHOD (1) : C++

```
class Voiture {  
public:  
    virtual string getMarque() const = 0;  
    friend ostream &operator<<(ostream &o,const Voiture *v) {  
        return o << v->getMarque();  
    }  
};  
  
class Renault : public Voiture {  
public:  
    string getMarque() const { return "Renault"; }  
};  
  
class Fiat : public Voiture {  
public:  
    string getMarque() const { return "Fiat"; }  
};
```

FACTORY METHOD (1) : C++

```
// cette fonction est indépendante du type concret
void f(const Voiture *v) {
    cout << "Voiture " << v << endl;
}
// cette partie du code est dépendante du type concret!
int main() {
    Voiture *v = new Renault();
    f(v);
    return 0;
}
```

Voiture Renault

FACTORY METHOD (1) : C#

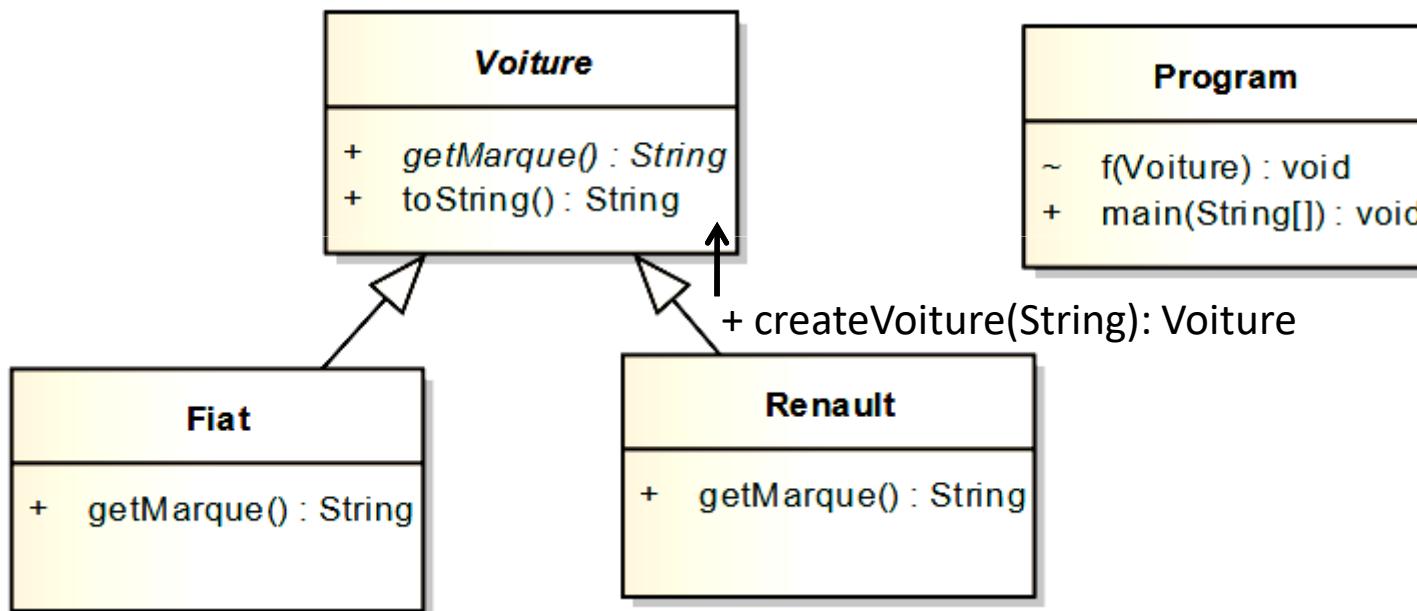
```
abstract class Voiture {  
    public abstract string getMarque();  
    public override string ToString(){ return getMarque(); }  
}  
class Renault : Voiture {  
    override public string getMarque() { return "Renault"; }  
}  
class Fiat : Voiture {  
    override public string getMarque() { return "Fiat"; }  
}
```

FACTORY METHOD (1) : C#

```
class Program{
    static void f(Voiture v) {
        Console.WriteLine("Voiture " + v);
    }
    static void Main(string[] args) {
        Voiture v = new Renault();
        f(v);
    }
}
```

Voiture Renault

FACTORY METHOD : DC



FACTORY METHOD

- La méthode **createVoiture** est une méthode de fabrication
 - Elle reçoit en paramètre les critères qui permettront au concepteur des objets Voiture d'opérer le choix du type concret.
 - Dans le code client, les types concrets n'apparaissent plus.
 - Le code client est devenu entièrement indépendant des types concrets.

FACTORY METHOD (2) : Java

```
abstract class Voiture {
    public abstract String getMarque();
    public String toString() {
        return getMarque();
    }
    // méthode statique pour créer un objet
    public static Voiture createVoiture(String origine) {
        if (origine.equals("fr"))    return new Renault();
        if (origine.equals("it"))    return new Fiat();
        return null;
    }
}
public class Renault extends Voiture { ... }
public class Fiat extends Voiture { ... }
```

FACTORY METHOD (2) : Java

```
public class Program {  
    static void f(Voiture v) {  
        System.out.println("Voiture " + v);  
    }  
  
    public static void main(String[] args) {  
        Voiture v = Voiture.createVoiture("fr");  
        f(v);  
    }  
}
```

Voiture Renault

FACTORY METHOD (2) : C++

```
class Voiture {
public:
    virtual string getMarque() const = 0;
    friend ostream &operator<<(ostream &o,const Voiture *v) {
        return o << v->getMarque();
    }
    // méthode statique pour créer un objet
    static Voiture *createVoiture(string o);
};

class Renault : public Voiture {
    // ...
};

class Fiat : public Voiture {
    // ...
};
```

FACTORY METHOD (2) : C++

```
Voiture *Voiture::createVoiture(string origine) {
    if (origine=="fr") return new Renault;
    if (origine=="it") return new Fiat;
    return NULL;
}
// cette fonction est indépendante du type concret
void f(const Voiture *v) { ... }
// cette fois le type concret n'apparaît pas!
int main() {
    Voiture *v = Voiture::createVoiture("fr");
    f(v);
    return 0;
}
```

Voiture Renault

FACTORY METHOD (2) : C#

```
abstract class Voiture {  
    abstract public string getMarque();  
    public override string ToString() { return getMarque(); }  
    // méthode statique pour créer un objet  
    public static Voiture createVoiture(string origine) {  
        if (origine=="fr") return new Renault();  
        if (origine=="it") return new Fiat();  
        return null;  
    }  
}  
class Renault : Voiture { ... }  
class Fiat : Voiture { ... }
```

FACTORY METHOD (2) : C#

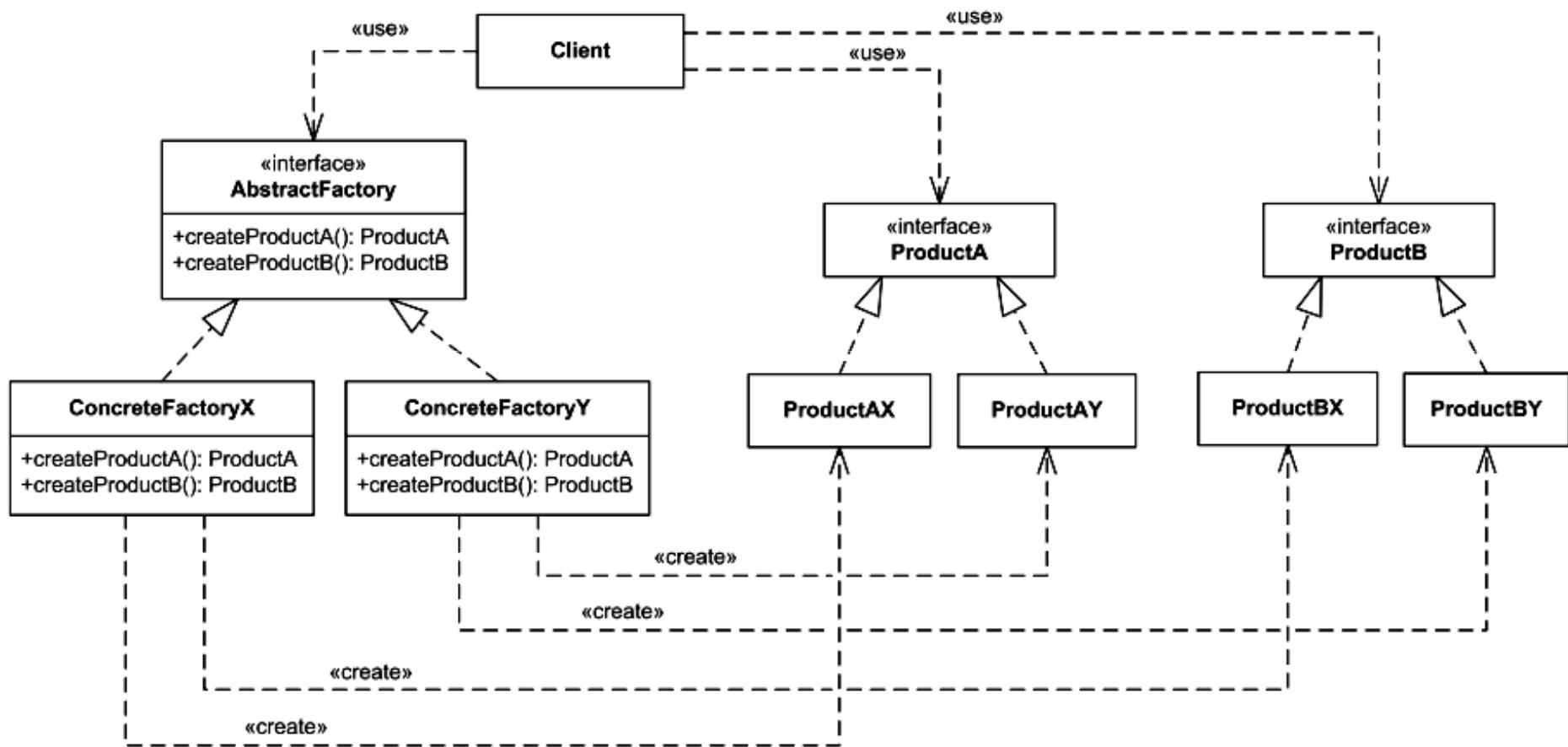
```
class Program {
    // cette fonction est indépendante du type concret
    static void f(Voiture v) {
        Console.WriteLine("Voiture " + v);
    }
    static void Main(string[] args) {
        Voiture v = Voiture.createVoiture("fr");
        f(v);
        Console.ReadLine();
    }
}
```

Voiture Renault

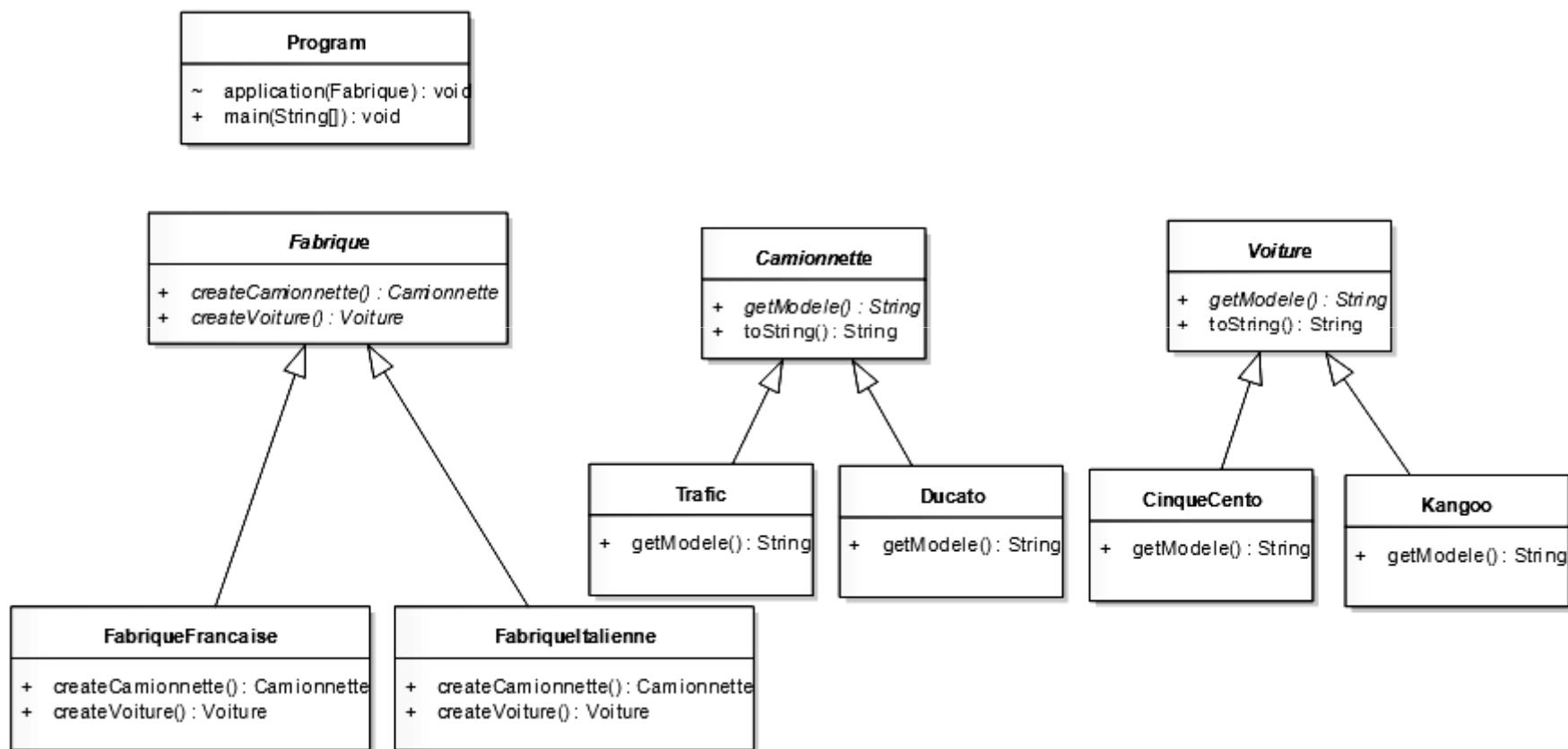
FABRIQUE ABSTRAITE (ABSTRACT FACTORY)

- La **fabrique abstraite** est un patron de conception (*design pattern*) créational utilisé en génie logiciel orienté objet.
- Elle fournit une interface pour créer des familles d'objets liés ou inter-dépendants sans avoir à préciser au moment de leur création la classe concrète à utiliser.
- Une fabrique abstraite encapsule un ensemble de **fabriques** ayant une thématique commune.
- Le code client crée une instance concrète de la fabrique abstraite, puis utilise son interface générique pour créer des objets concrets de la thématique.

ABSTRACT FACTORY: DC



ABSTRACT FACTORY : DC



ABSTRACT FACTORY : Java

```
abstract class Voiture {
    abstract public String getModele();
    public String toString() {
        return getModele();
    }
}
public class Kangoo extends Voiture {
    @Override
    public String getModele() {
        return "Kangoo";
    }
}
public class CinqueCento extends Voiture {
    @Override
    public String getModele() {
        return "500";
    }
}
```

ABSTRACT FACTORY : Java

```
abstract class Camionnette {  
    abstract public String getModele() ;  
    public String toString(){  
        return getModele();  
    }  
}  
  
public class Trafic extends Camionnette {  
    @Override  
    public String getModele() {  
        return "Trafic";  
    }  
}  
  
public class Ducato extends Camionnette {  
    @Override  
    public String getModele() {  
        return "Ducato";  
    }  
}
```

ABSTRACT FACTORY : Java

```
abstract class Fabrique {
    abstract public Voiture createVoiture();
    abstract public Camionnette createCamionnette();
}

public class FabriqueFrancaise extends Fabrique {
    @Override
    public Voiture createVoiture() {      return new Kangoo();      }
    @Override
    public Camionnette createCamionnette() {      return new Trafic();      }
}

public class FabriqueItalienne extends Fabrique {
    @Override
    public Voiture createVoiture() {      return new CinqueCento();      }
    @Override
    public Camionnette createCamionnette() {      return new Ducato();      }
}
```

ABSTRACT FACTORY : Java

```
public class Program {  
    static void application(Fabrique f) {  
        Voiture v = f.createVoiture();  
        Camionnette c = f.createCamionnette();  
        System.out.println(v);  
        System.out.println(c);  
    }  
    public static void main(String[] args) {  
        FabriqueItalienne fi = new FabriqueItalienne();  
        application(fi);  
        FabriqueFrancaise fr = new FabriqueFrancaise();  
        application(fr);  
    }  
}
```

500
Ducato
Kangoo
Trafic

ABSTRACT FACTORY : C++

```
class Voiture {
public:
    virtual string getModele() const = 0;
    friend ostream &operator<<(ostream &o,const Voiture *v){
        return o << v->getModele();
    }
};

class Kangoo : public Voiture {
public:
    string getModele() const { return "Kangoo"; }
};

class CinqueCento : public Voiture {
public:
    string getModele() const { return "500"; }
};
```

ABSTRACT FACTORY : C++

```
class Camionnette {  
public:  
    virtual string getModele() const = 0;  
    friend ostream &operator<<(ostream &o,const Camionnette *v) {  
        return o << v->getModele();  
    }  
};  
class Trafic : public Camionnette {  
public:  
    string getModele() const { return "Trafic"; }  
};  
class Ducato : public Camionnette {  
public:  
    string getModele() const { return "Ducato"; }  
};
```

ABSTRACT FACTORY : C++

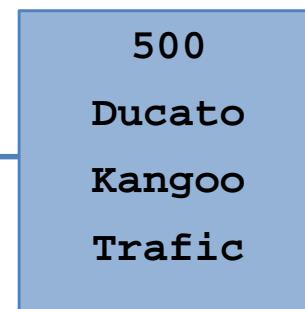
```
class Fabrique {
public:
    virtual Voiture *createVoiture() = 0;
    virtual Camionnette *createCamionnette() = 0;
};

class FabriqueFrancaise : public Fabrique {
public:
    Voiture *createVoiture() { return new Kangoo; }
    Camionnette *createCamionnette() { return new Trafic; }
};

class FabriqueItalienne : public Fabrique {
public:
    Voiture *createVoiture() { return new CinqueCento; }
    Camionnette *createCamionnette() { return new Ducato; }
};
```

ABSTRACT FACTORY : C++

```
void application(Fabrique &f) {  
    Voiture *v = f.createVoiture();  
    Camionnette *c = f.createCamionnette();  
    cout << v << endl;  
    cout << c << endl;  
}  
int main() {  
    FabriqueItalienne fi;  
    application(fi);  
    FabriqueFrancaise fr;  
    application(fr);  
    return 0;  
}
```



ABSTRACT FACTORY : C#

```
abstract class Voiture {
    abstract public string getModele();
    public override string ToString(){
        return getModele();
    }
}
class Kangoo : Voiture {
    override public string getModele() { return "Kangoo"; }
}
class CinqueCento : Voiture {
    override public string getModele() { return "500"; }
}
```

ABSTRACT FACTORY : C#

```
abstract class Camionnette {
    abstract public string getModele() ;
    public override string ToString(){
        return getModele();
    }
}
class Trafic : Camionnette {
    override public string getModele() { return "Trafic"; }
}
class Ducato : Camionnette {
    override public string getModele() { return "Ducato"; }
}
```

ABSTRACT FACTORY : C#

```
abstract class Fabrique {
    abstract public Voiture createVoiture() ;
    abstract public Camionnette createCamionnette();
}

class FabriqueFrancaise : Fabrique {
    override public Voiture createVoiture() { return new Kangoo(); }
    override public Camionnette createCamionnette() { return new
Trafic(); }
}

class FabriqueItalienne : Fabrique {
    override public Voiture createVoiture() {
        return new CinqueCento(); }
    override public Camionnette createCamionnette() {
        return new Ducato(); }
}
```

ABSTRACT FACTORY : C#

```
class Program{
    static void application(Fabrique f) {
        Voiture v = f.createVoiture();
        Camionnette c = f.createCamionnette();
        Console.WriteLine(v);
        Console.WriteLine(c);
    }
    static void Main(string[] args){
        FabriqueItalienne fi = new FabriqueItalienne();
        application(fi);
        FabriqueFrancaise fr = new FabriqueFrancaise();
        application(fr);
        Console.ReadLine();
    }
}
```

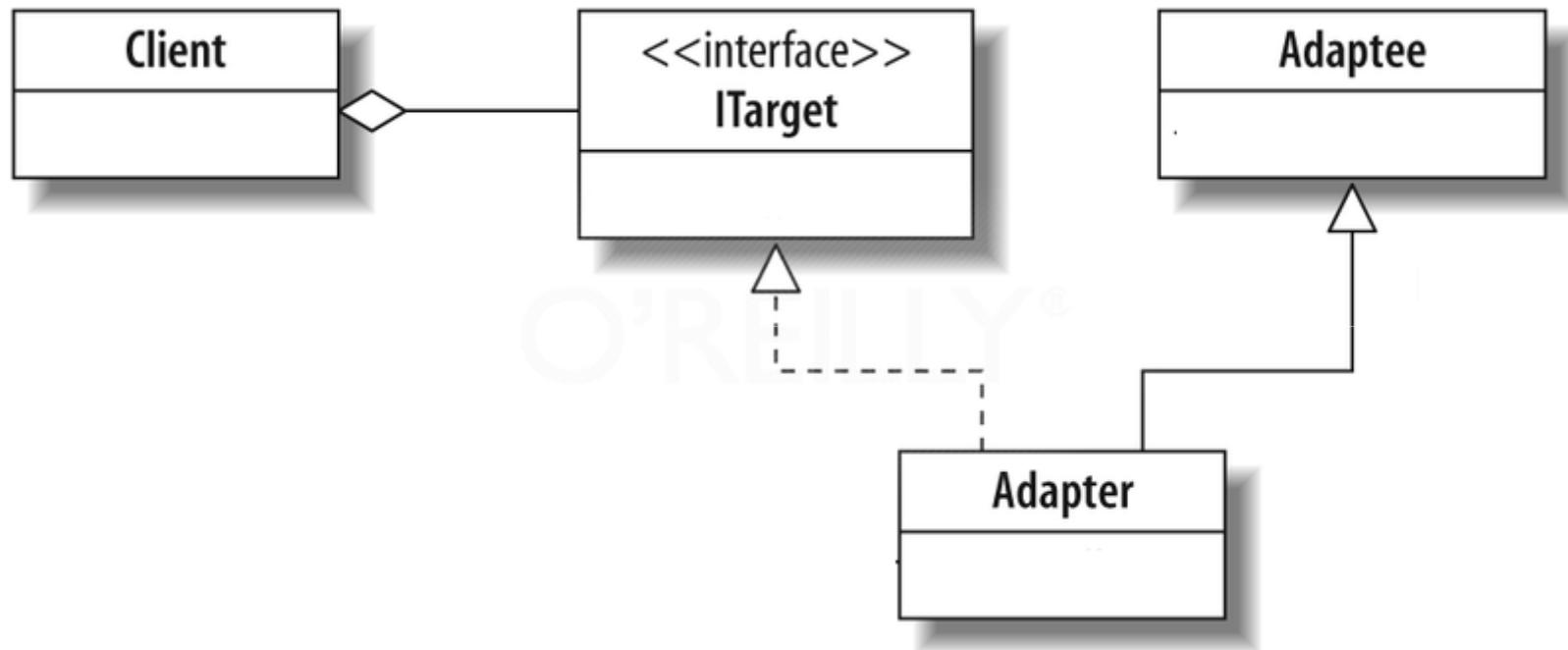
500
Ducato
Kangoo
Trafic

CHAPITRE II: PATRON DE CONCEPTION DE STRUCTURE (*STRUCTURAL DP*)

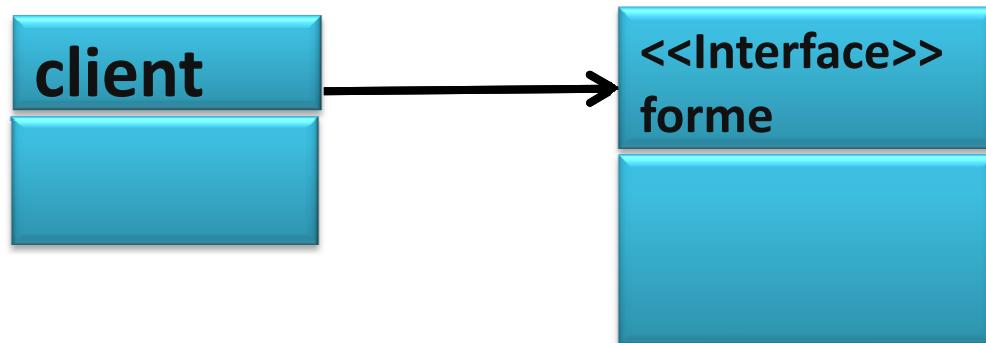
ADAPTER

- Le pattern adaptateur/adapter permet de convertir l'interface d'un objet en une autre interface (transstypage).
- Pour des raisons de compatibilité (reprendre une ancienne classe mais l'adapter à une nouvelle interface de manipulation).
- Pour une question de réutilisabilité (récupérer un objet dans un autre cadre que pour lequel il a été défini).

ADAPTER: DC



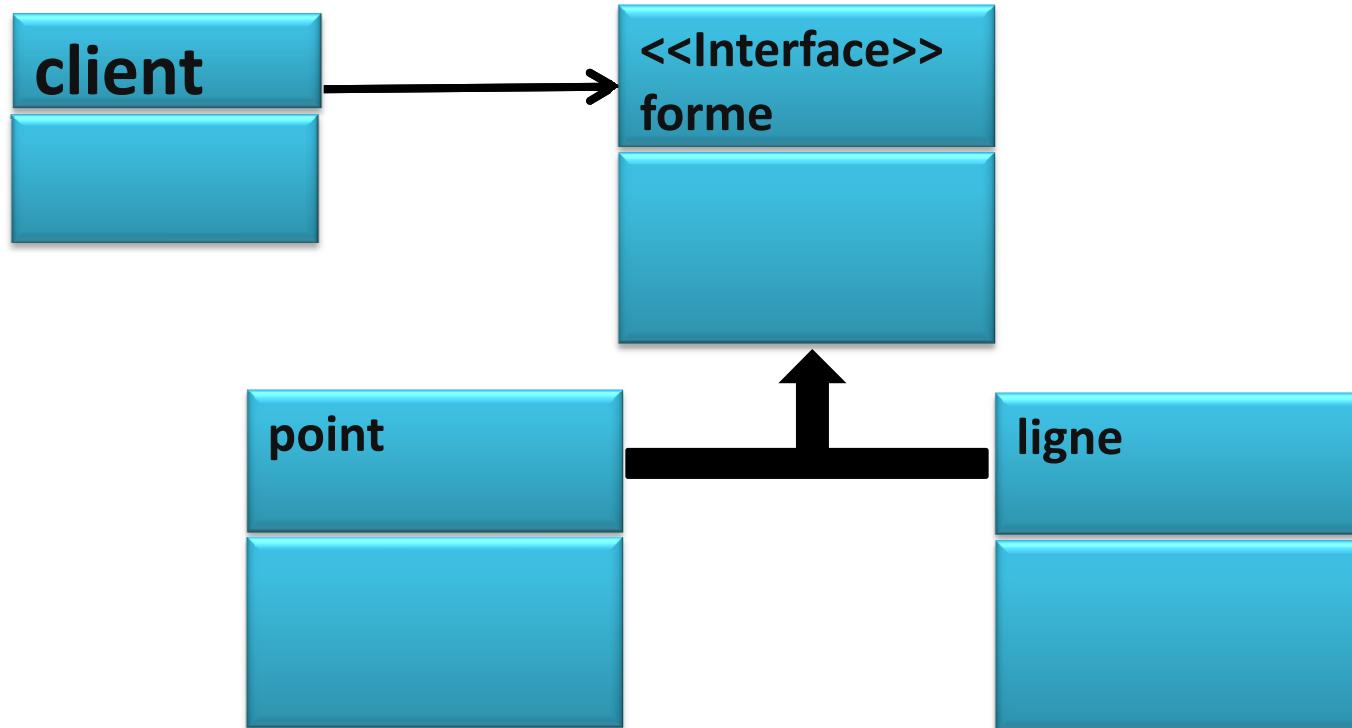
Problématique



✓ les objets clients utilisent seulement des formes

✓ mais qui sont ces formes ?

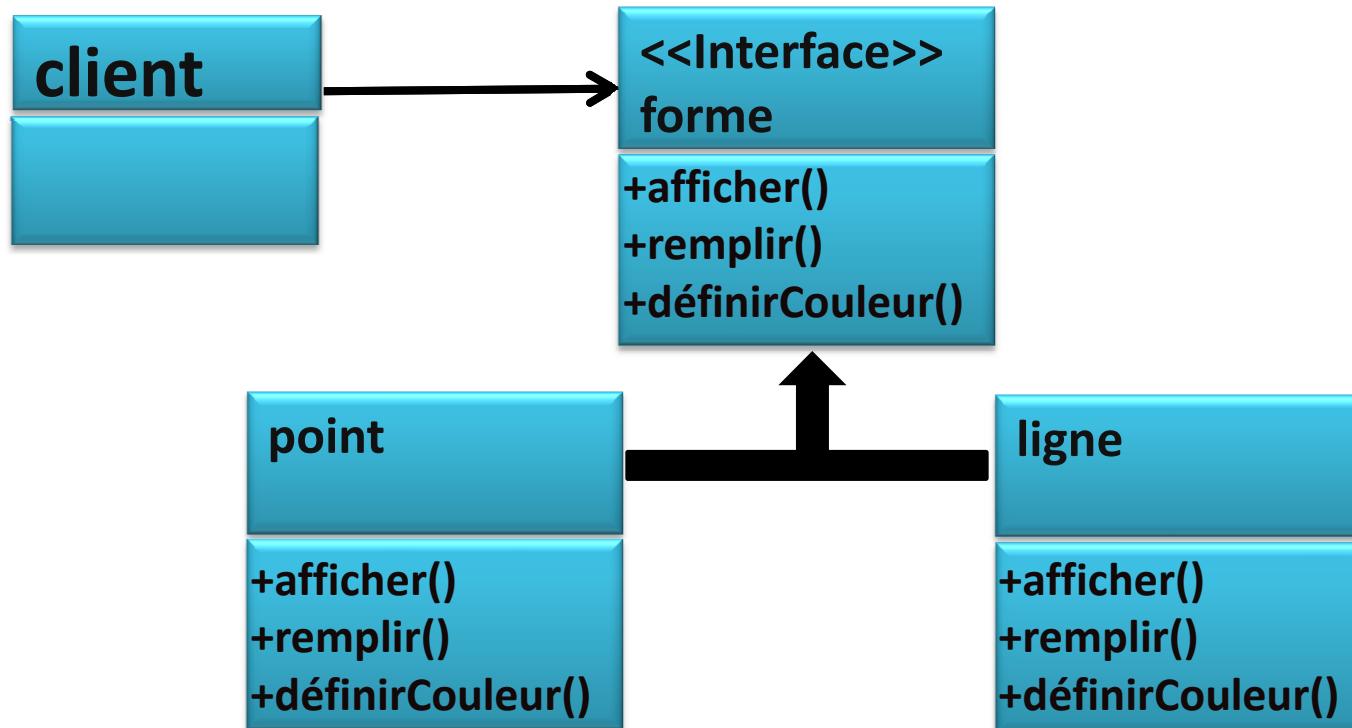
Problématique



✓ donc le point et la ligne implémentent l'interface forme

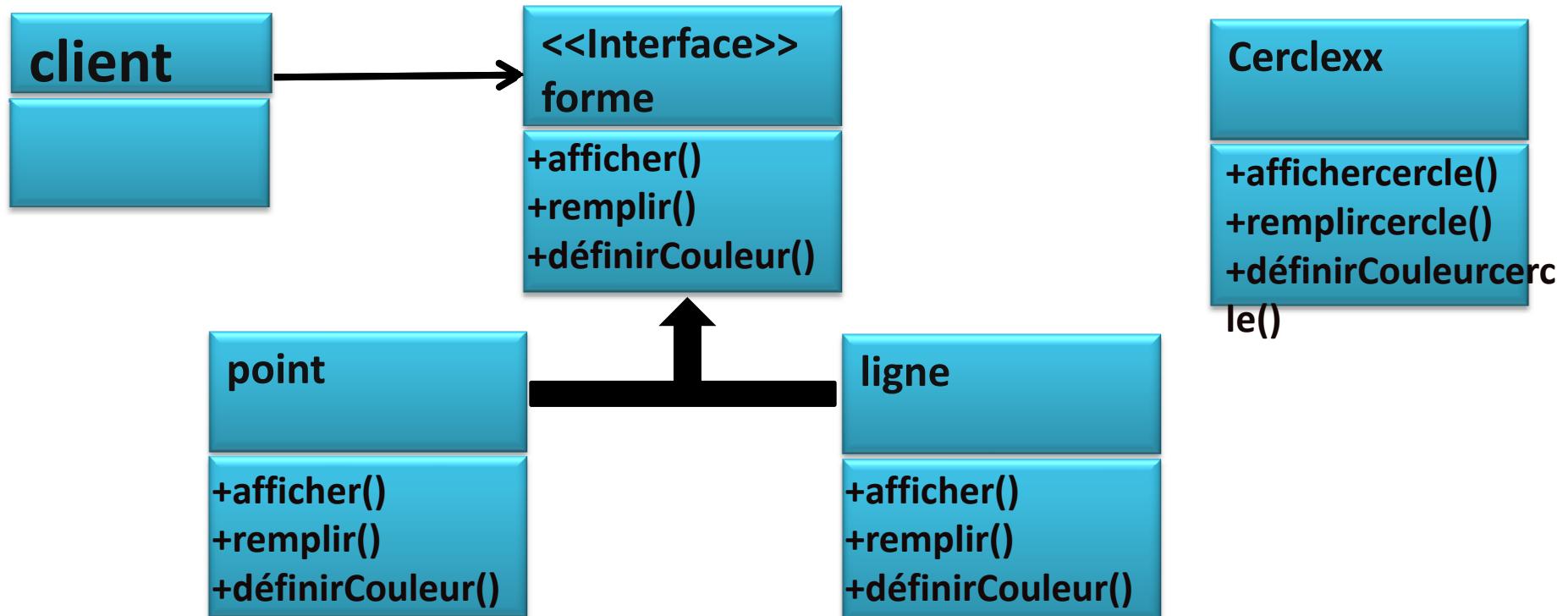
✓ mais quelle sont les conditions d'implémentation de l'interface ?

Problématique



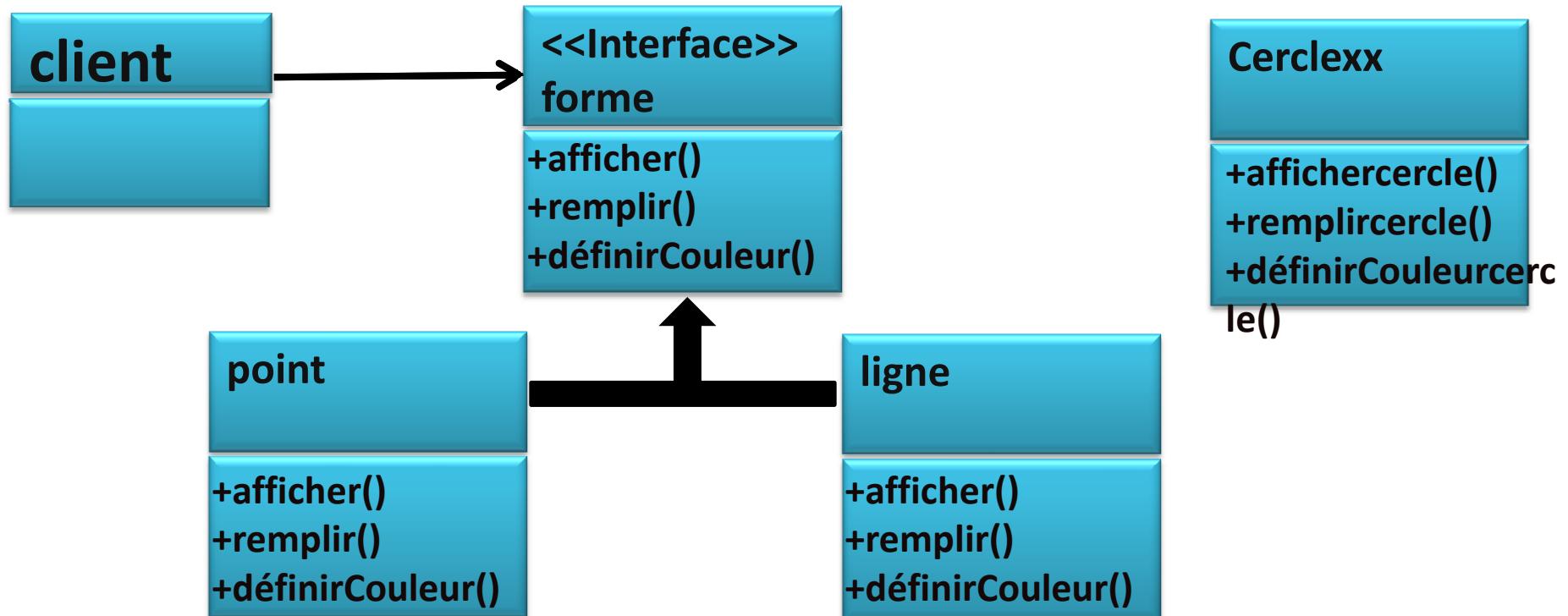
- ✓ donc il faut définir le code des méthodes de l'interface en respectant la signature des fonctions de l'interface
- ✓ Est-ce que le client est satisfait par ces deux classes ?

Problématique



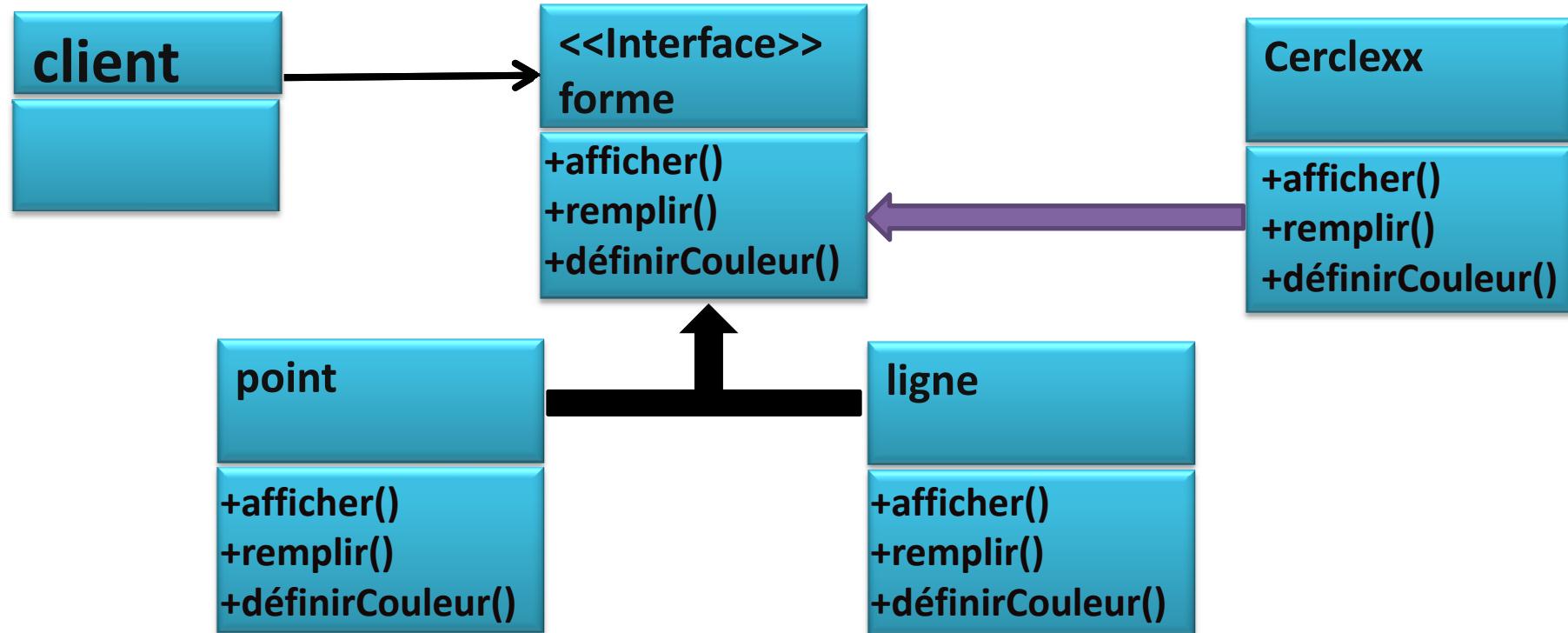
- ✓ le client a besoin d'une classe cercle
- ✓ On a une classe qui s'appelle cerclexx (qui répond au besoin de client) qui est déjà prêt avec le code de ces méthodes
- ✓ tout va bien pour l'instant mais où est le problème ?

Problématique



- ✓ on a deux problèmes :
- ✓ Le client ne peut pas utiliser des formes qui n'implémente pas l'interface forme
- ✓ La classe cerclexx ne peut pas implémenter l'inteface forme parce que les méthodes de cerclexx n'ont pas les mêmes nom et les mêmes paramètres des fonctions que les méthodes d'interface

Solution sans pattern adaptateur

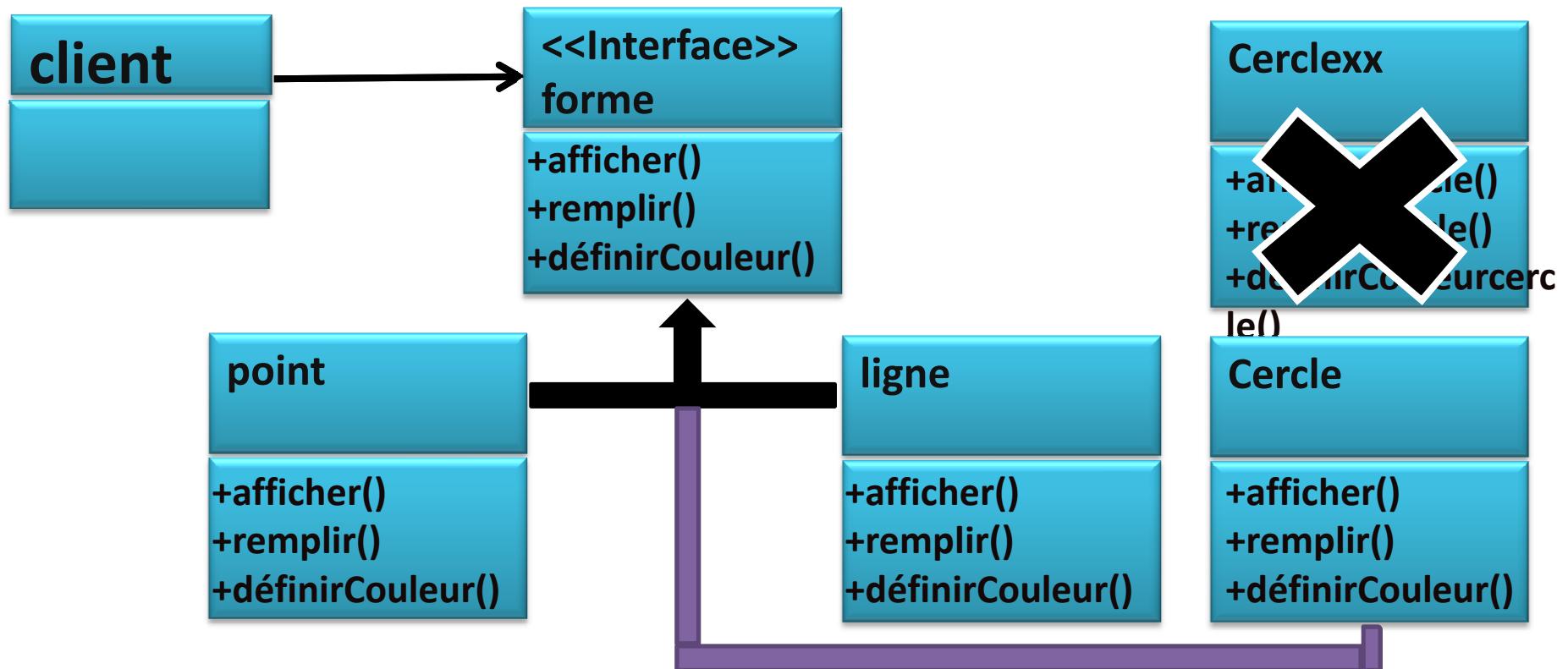


✓ Première solution :

✓ on va modifier les noms des fonctions et les paramètres des méthodes de la classe cerclexx pour qu'elle soit compatible avec celles de l'interface formes

✓ Mais cette solution va nous couter beaucoup de temps et un énorme effort

Solution sans pattern adaptateur

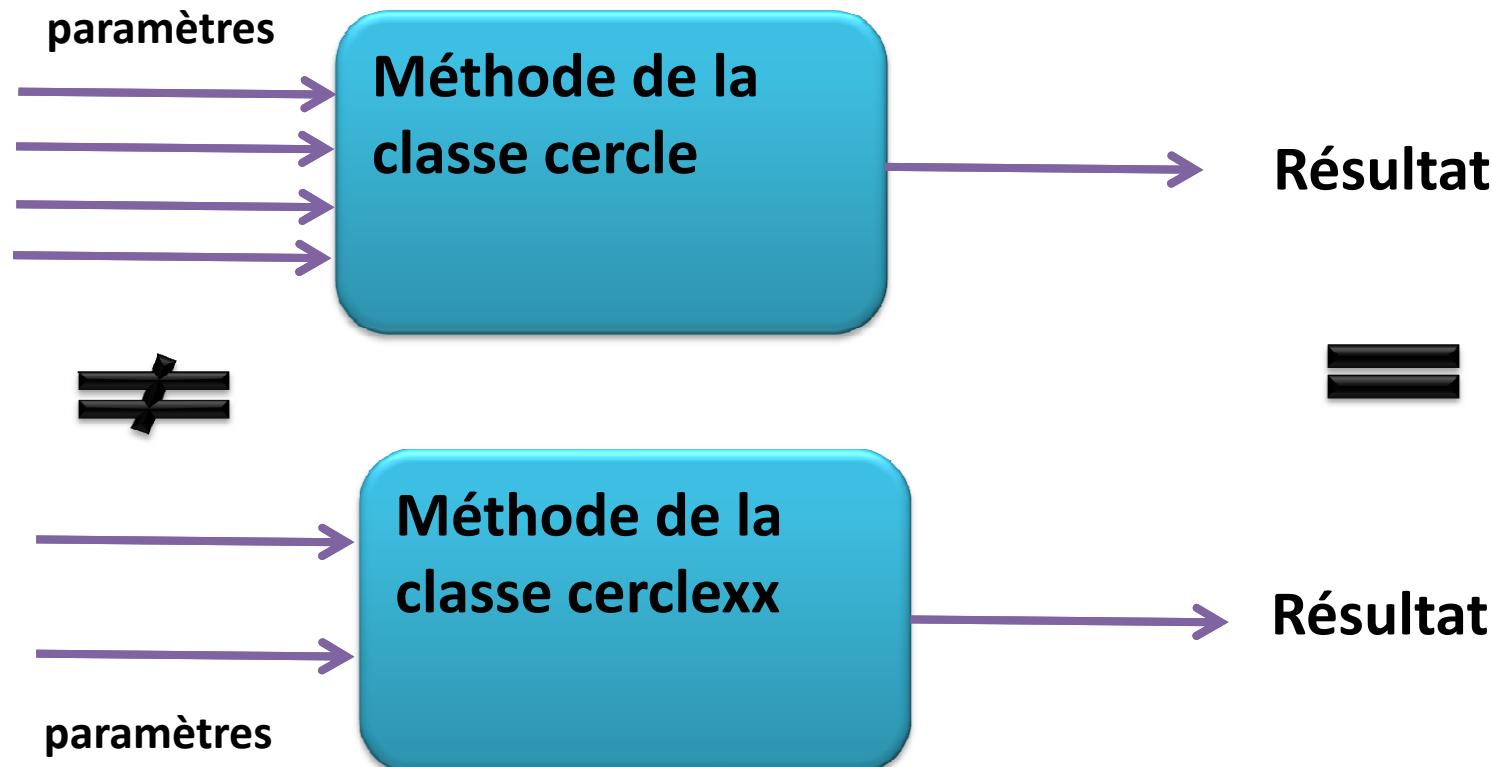


✓ Deuxième solution :

✓ on va créer une classe cercle qui implémente l'interface forme mais à condition de réécrire le code des méthodes

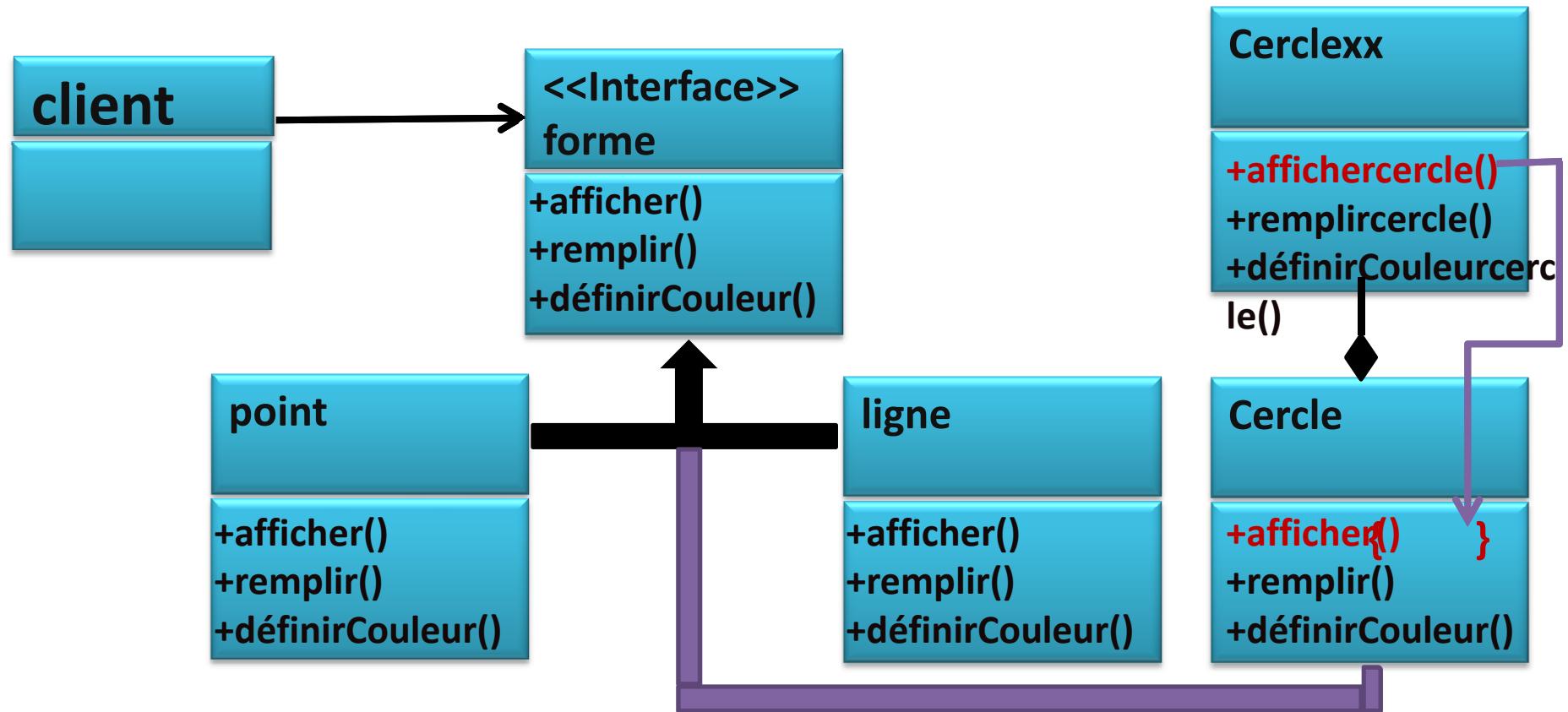
✓ Mais cette solution va nous couter aussi beaucoup de temps et un énorme effort

Solution avec pattern adaptateur



- ✓ puisque les méthodes de la classe de cerclexx donnent les fonctionnalité attendues par l'utilisateur donc nous avons seulement besoin d'adapter la classe cerclexx avec l'interface forme

Solution avec pattern adaptateur



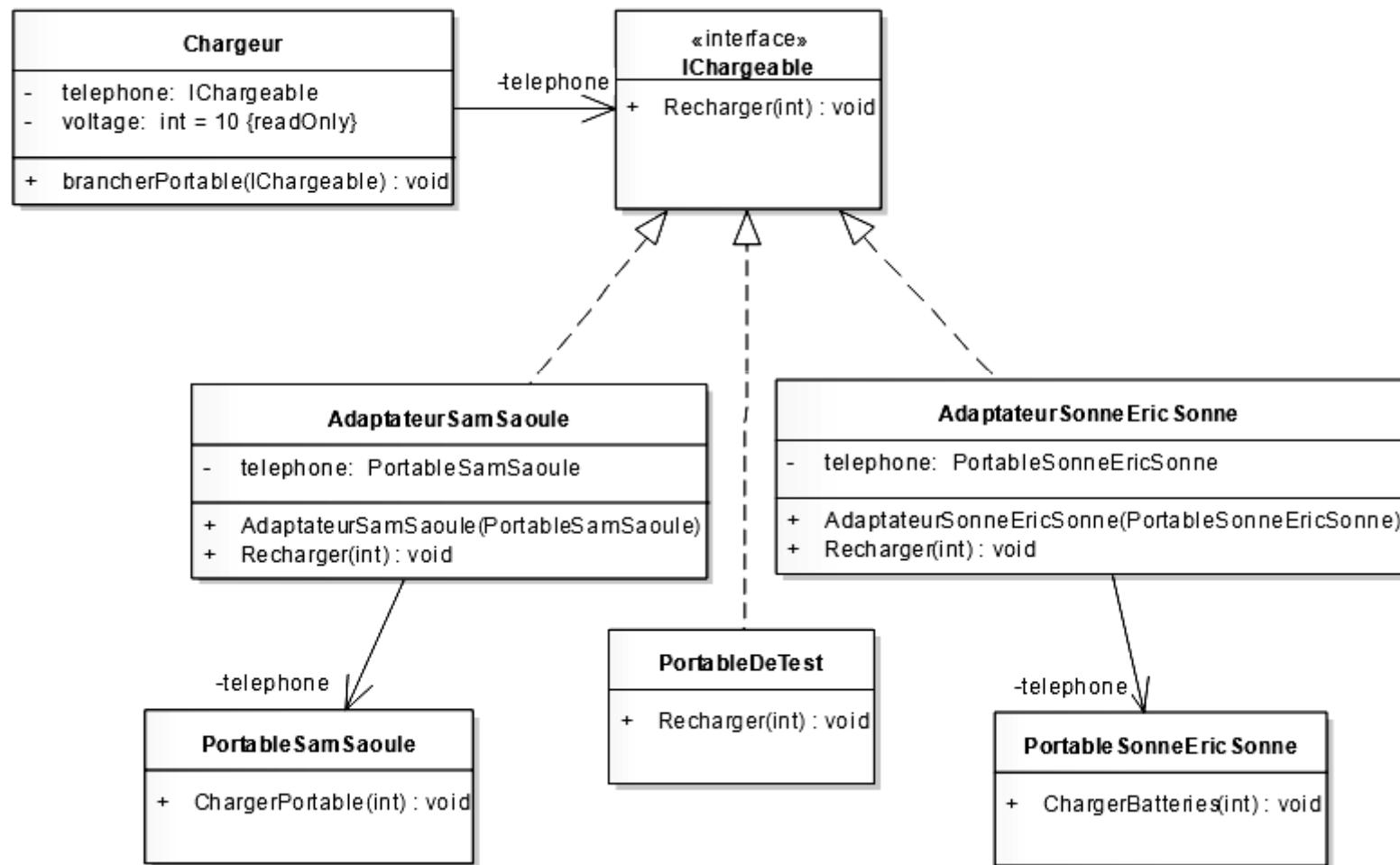
✓ On va pas réécrire le code dans les fonctions de la classe cercle mais on va utiliser les fonctions de la classe cerclesxx dedans les fonctions de la classe cercle à travers un objet de la classe cerclexx

Solution avec pattern adaptateur

- Extrait de code Java de la classe cercle avec l'utilisation du patron adaptateur

```
class Cercle implements Forme {  
  
    ...  
    private CercleXX pcx;  
    ...  
  
    public Cercle () {  
        pcx= new CercleXX();  
    }  
  
    void public afficher() {  
        pcx.afficherCercle();  
    }  
}
```

ADAPTER : DC



ADAPTER : Java

```
public interface IChargeable {  
    void Recharger(int volts);  
}  
public class Chargeur {  
    private IChargeable telephone;  
    private final int voltage = 10;  
  
    public void brancherPortable(IChargeable portable) {  
        System.out.println("branchement d'un portable");  
        this.telephone = portable;  
        this.telephone.Recharger(voltage);  
    }  
}
```

ADAPTER : Java

```
public class PortableSamSaoule {  
    // ne se recharge qu'avec du 5 volts  
    public void ChargerPortable(int volts) {  
        System.out.println("Portable SamSaoule en charge");  
        System.out.println("voltage : " + volts);  
    }  
}  
  
public class PortableSonneEricSonne {  
    // ne se recharge qu'avec du 10 volts  
    public void ChargerBatteries(int volts) {  
        System.out.println("Portable SonneEricSonne en charge");  
        System.out.println("voltage : " + volts);  
    }  
}
```

ADAPTER : Java

```
public class AdaptateurSamSaoule implements IChargeable {  
    // référence sur le portable adapté  
    private PortableSamSaoule telephone;  
  
    public AdaptateurSamSaoule(PortableSamSaoule portable) {  
        this.telephone = portable;  
    }  
  
    // le portable SamSaoule n'a besoin que de 5 volts  
    public void Recharger(int volts) {  
        // on modifie le voltage  
        int nouveauVoltage = volts > 5 ? 5 : volts;  
        this.telephone.Cha...erPortable(nouveauVoltage);  
    }  
}
```

ADAPTER : Java

```
public class AdaptateurSonneEricSonne implements IChargeable {  
    // référence sur le portable adapté  
    private PortableSonneEricSonne telephone;  
  
    public AdaptateurSonneEricSonne(  
        PortableSonneEricSonne portable) {  
        this.telephone = portable;  
    }  
  
    public void Recharger(int volts) {  
        this.telephone.ChaargerBatteries(volts);  
    }  
}
```

ADAPTER : Java

```
public class Program {
    public static void main(String[] args) {
        // on crée le chargeur
        Chargeur chargeur = new Chargeur();
        // on crée le portable et son adaptateur
        PortableSonneEricSonne portableSonne = new PortableSonneEricSonne();
        AdaptateurSonneEricSonne adapteurSonne = new AdaptateurSonneEricSonne(
            portableSonne);

        // on donne le portable à charger mais en utilisant son adaptateur
        chargeur.brancherPortable(adapteurSonne);

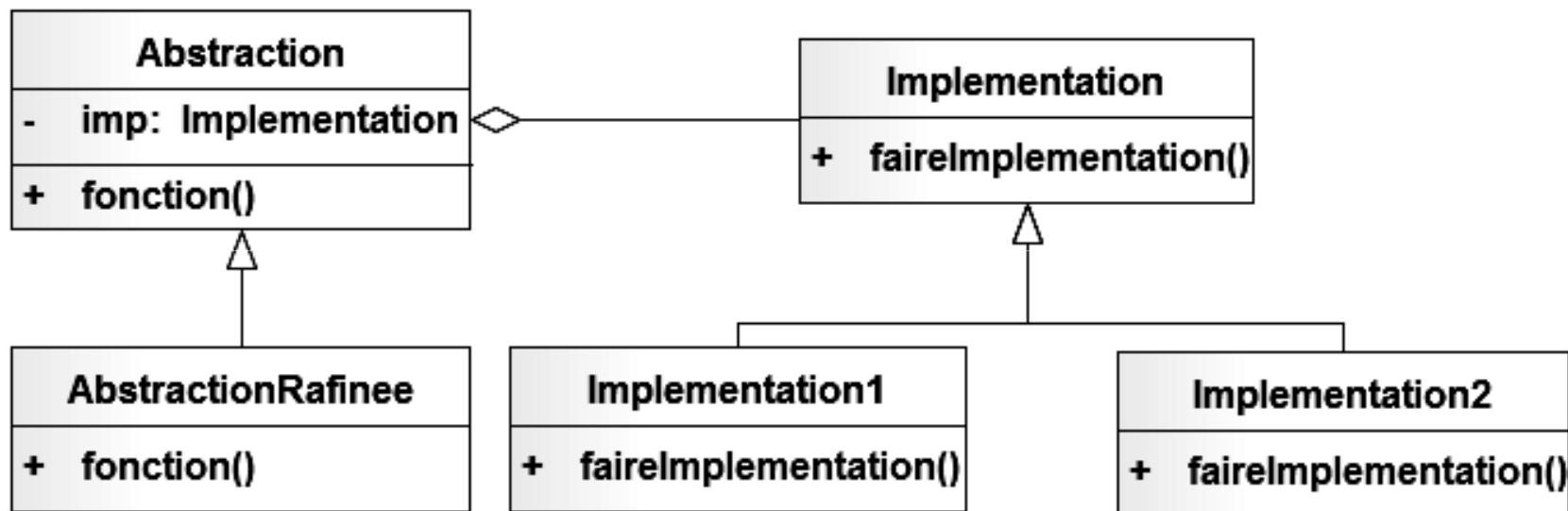
        // on crée le portable et son adaptateur
        PortableSamSaoule portableSam = new PortableSamSaoule();
        AdaptateurSamSaoule adapteurSam = new AdaptateurSamSaoule(portableSam);

        // on donne le portable à charger mais en utilisant son adaptateur
        chargeur.brancherPortable(adapteurSam);
    }
}
```

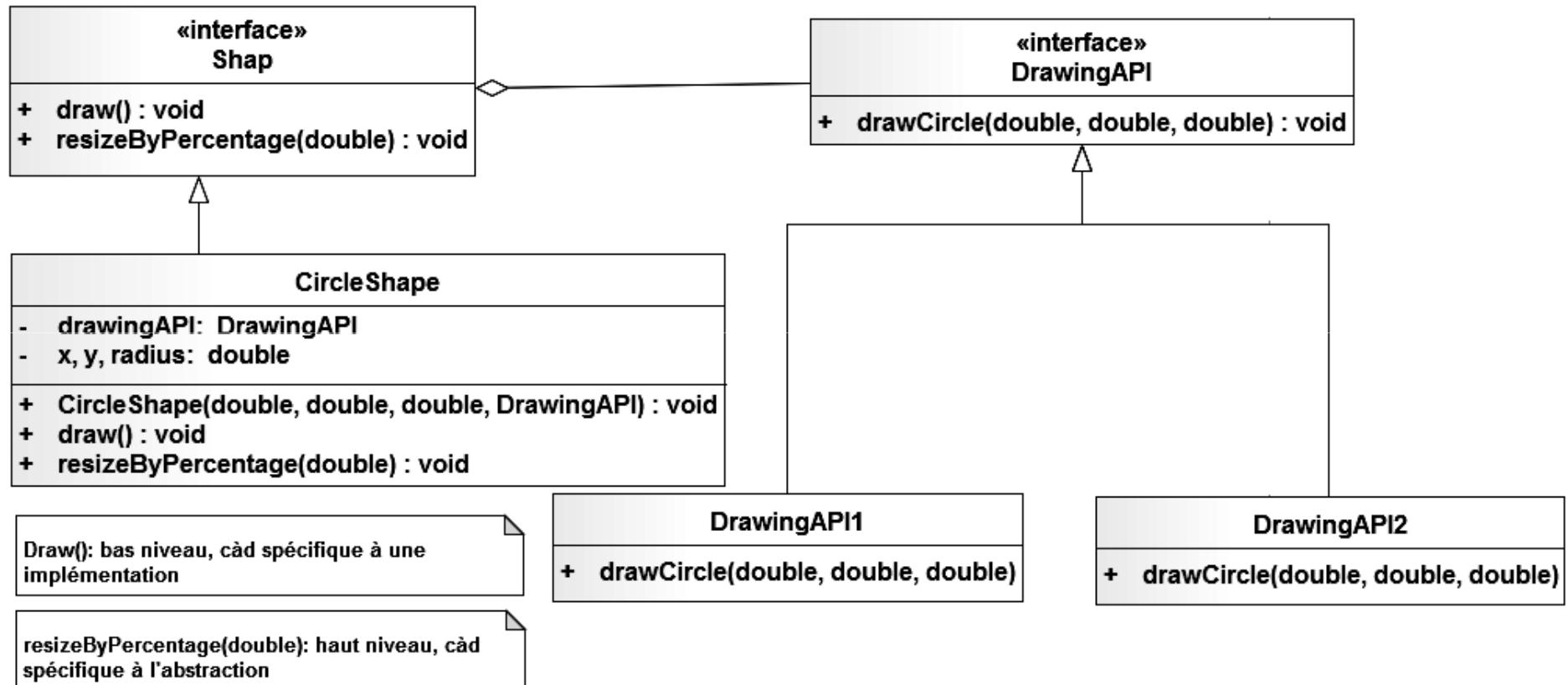
Pont (bridge)

- Le **pont** est un patron de conception de la famille structuration
- Le **pont** est un patron de conception qui permet de découpler l'interface d'une classe et son implémentation.
- L'implémentation d'une classe peut se modifier ou changer sans devoir modifier le code client.

Pont: DC



Pont: DC



Pont: Java

```
/** "Implémentation" */
interface DrawingAPI
{
    public void drawCircle(double x, double y, double radius);
}

/** "Implémentation1" */
class DrawingAPI1 implements DrawingAPI
{
    public void drawCircle(double x, double y, double radius)
    {
        System.out.printf("API1.cercle position %f:%f rayon %f\n", x, y, radius);
    }
}

/** "Implémentation2" */
class DrawingAPI2 implements DrawingAPI
{
    public void drawCircle(double x, double y, double radius)
    {
        System.out.printf("API2.cercle position %f:%f rayon %f\n", x, y, radius);
    }
}
```

Pont: Java

```
/** "Abstraction" */
interface Shape
{
    public void draw();                  // bas niveau
    public void resizeByPercentage(double pct); // haut niveau
}

/** "AbstractionRaffinée" */
class CircleShape implements Shape
{
    private double x, y, radius;
    private DrawingAPI drawingAPI;

    public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI)
    {
        this.x = x; this.y = y; this.radius = radius;
        this.drawingAPI = drawingAPI;
    }
}
```

Pont: Java

```
// bas niveau, càd spécifique à une implémentation
public void draw()
{
    drawingAPI.drawCircle(x, y, radius);
}

// haut niveau, càd spécifique à l'abstraction
public void resizeByPercentage(double pct)
{
    radius *= pct;
}
```

Pont: Java

```
/** Classe utilisatrice */
class BridgePattern
{
    public static void main(String[] args)
    {
        Shape[] shapes = new Shape[2];
        shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
        shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());

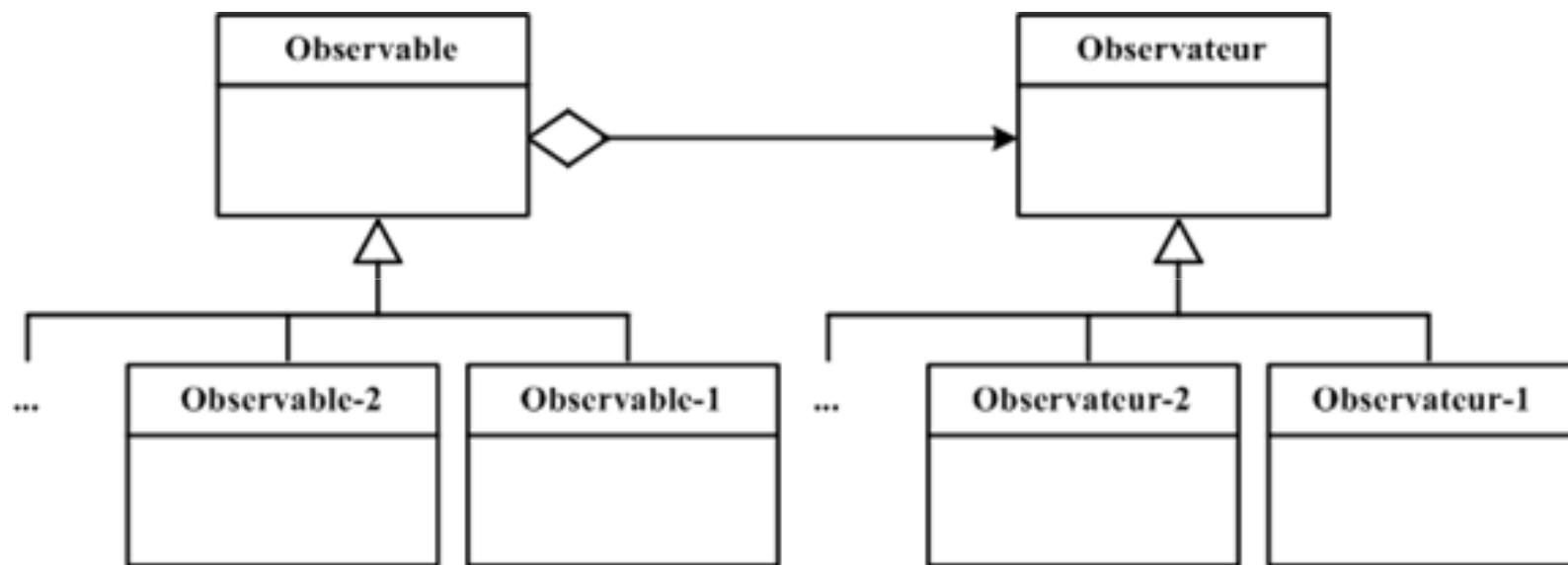
        for (Shape shape : shapes)
        {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
}
```

CHAPITRE III: PATRON DE CONCEPTION DE COMPORTEMENT (*BEHAVIOR DP*)

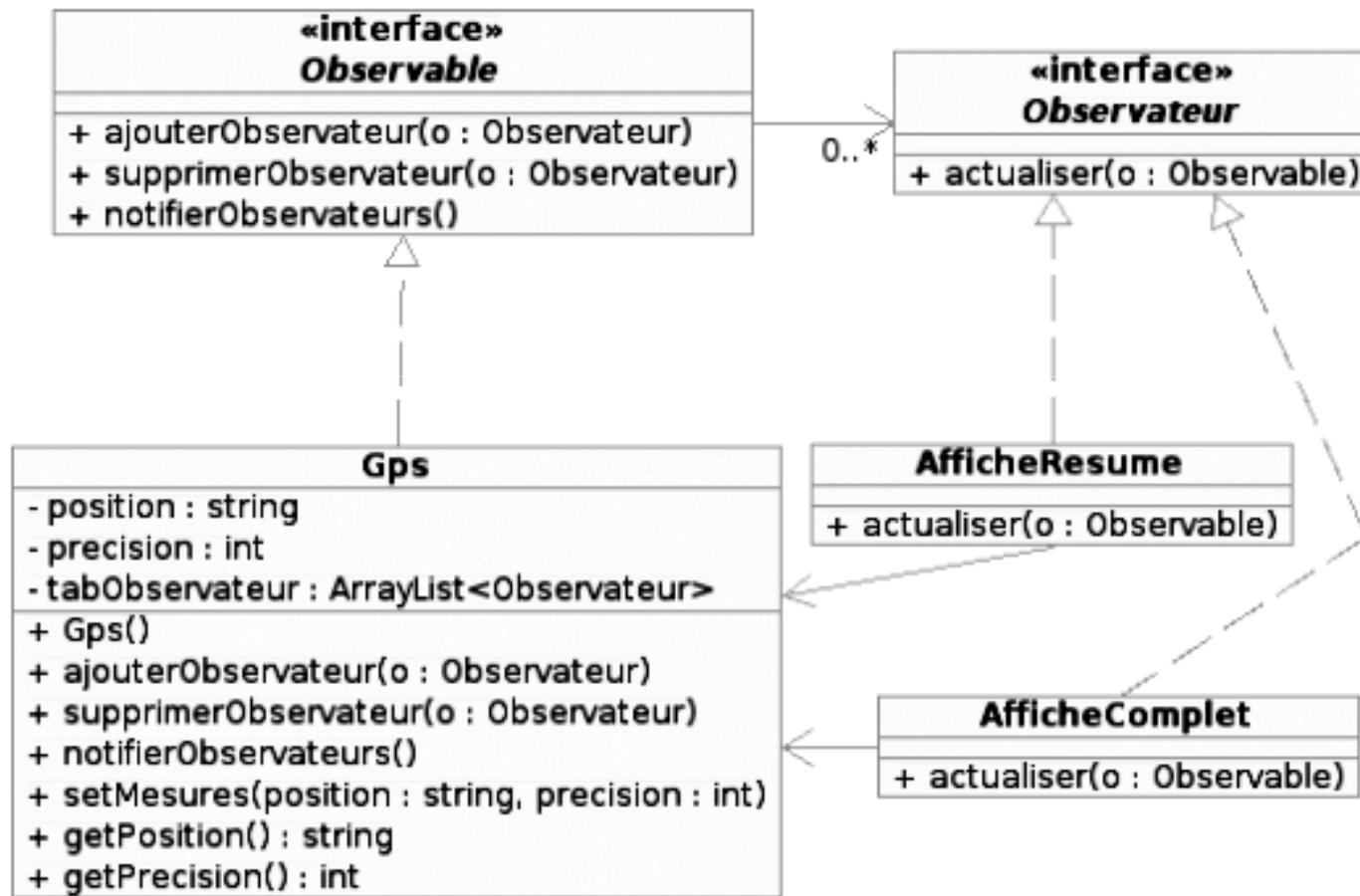
Observateur (observer)

- Le patron de conception **observateur** est utilisé en programmation pour envoyer un signal à des modules qui jouent le rôle d'**observateurs**.
- En cas de notification, les **observateurs** effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent (les « *observables* »).

Observateur: DC



Observateur: DC



Observateur (observer)

- Afin d'illustrer l'implémentation du pattern Observateur en Java réalisons une petite application permettant de se positionner grâce au GPS.
- Le principe du Global Positioning System est simple. Une personne souhaitant connaître sa position utilise un récepteur GPS. Ce récepteur reçoit des informations (position, date précise...) d'au moins quatre satellites (sur un total de 24 satellites).
- Grâce à la date transmise, le récepteur peut calculer la distance le séparant du satellite dont il connaît la position. Il renouvelle l'opération avec trois autres satellites et peut donc en déduire sa position dans l'espace (procédé appelé là trilateration).

Observateur (observer)

- Considérons que notre ordinateur est relié à un récepteur GPS par un réseau sans fil.
- On va concevoir une classe nommée Gps qui va stocker les informations du récepteur (positionnement, précision...). Puis deux autres classes (AfficheResume et AfficheComplet) permettant d'afficher de deux façons différentes ces informations.
- Comme dans la définition du pattern Observateur, on trouve également deux interfaces Observateur et Observable. Pour résumer la classe Gps sera observable et les classes AfficheResume et AfficheComplet seront ses observateurs. Voyons plus en détails le diagramme UML.