# Architecture Logicielle

# Les patrons d'interfaces

Florent Nicart

Université de Rouen

*2016–2017*

# Les patrons d'interface

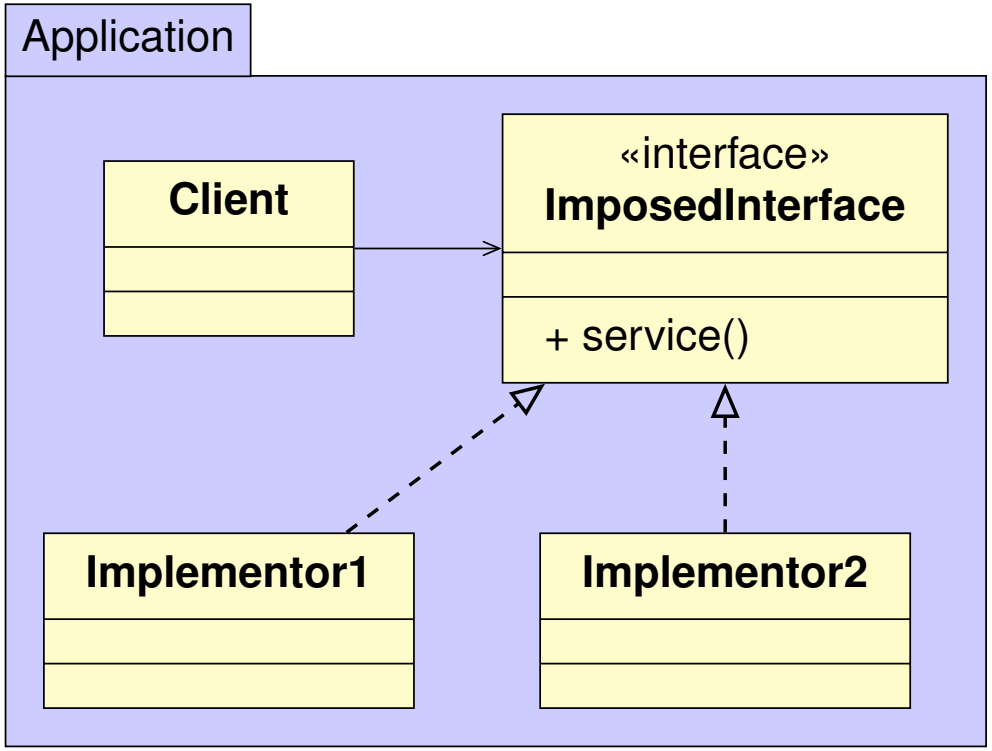
Les patrons d'interface opèrent sur les interfaces publiques des composants du système :

- Adaptateur (adapter) : fournit l'interface qu'un client attend en utilisant les services d'une classe dont l'interface est différente.
- Façade (facade) : fournit une interface facilitant l'emploi d'un sous système.
- Composite (composite) : permet au client de traiter de manière uniforme les objets et leurs composition.
- Passerelle (bridge) : découple une classe qui s'appuie sur des opérations abstraites de l'implémentation de ces opérations.

# Le patron Adaptateur

Adapter une classe (ou une interface) existante à une interface imposée.

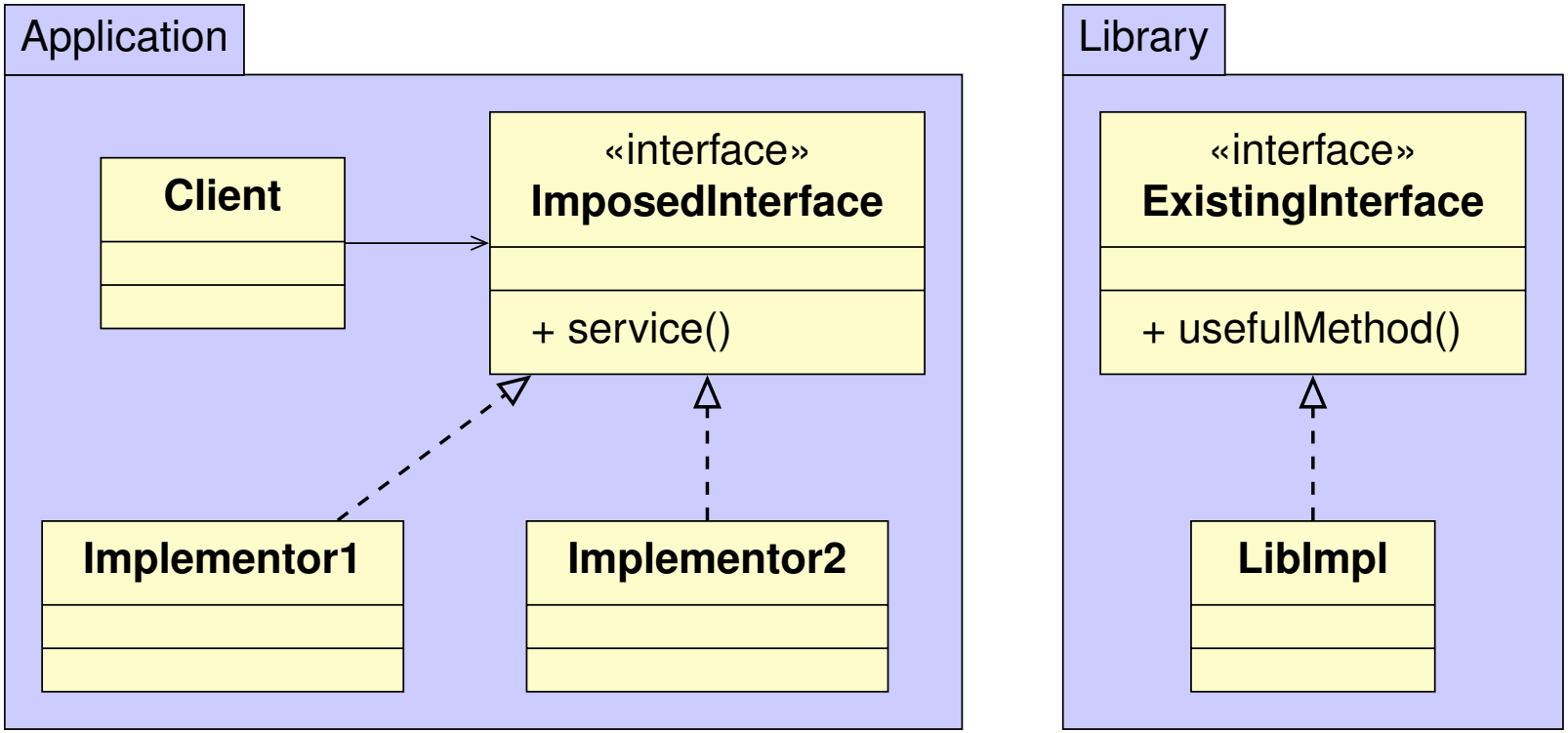
# Situation Initiale




- Une application est capable de manipuler des composants logiciels [1] dont le comportement est spécifié par une interface (contrat).

1. Dont le nombre est arbitraire.

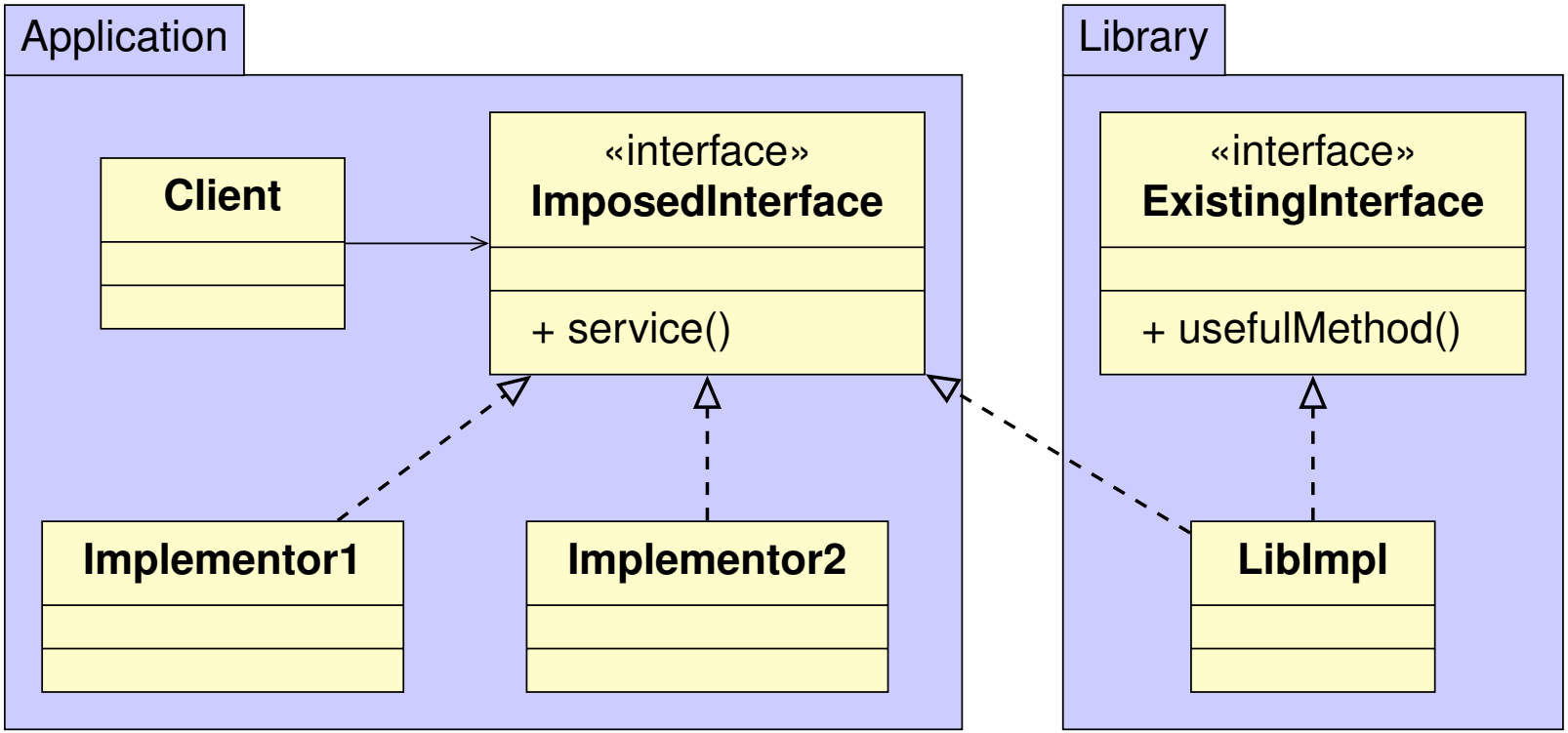
# Situation Initiale




- On souhaite ajouter un nouveau composant compatible en s'appuyant sur du code existant.
- Problème : bien que la bibliothèque remplisse parfaitement la tâche demandée, l'interface existante n'est pas compatible avec l'interface imposée.

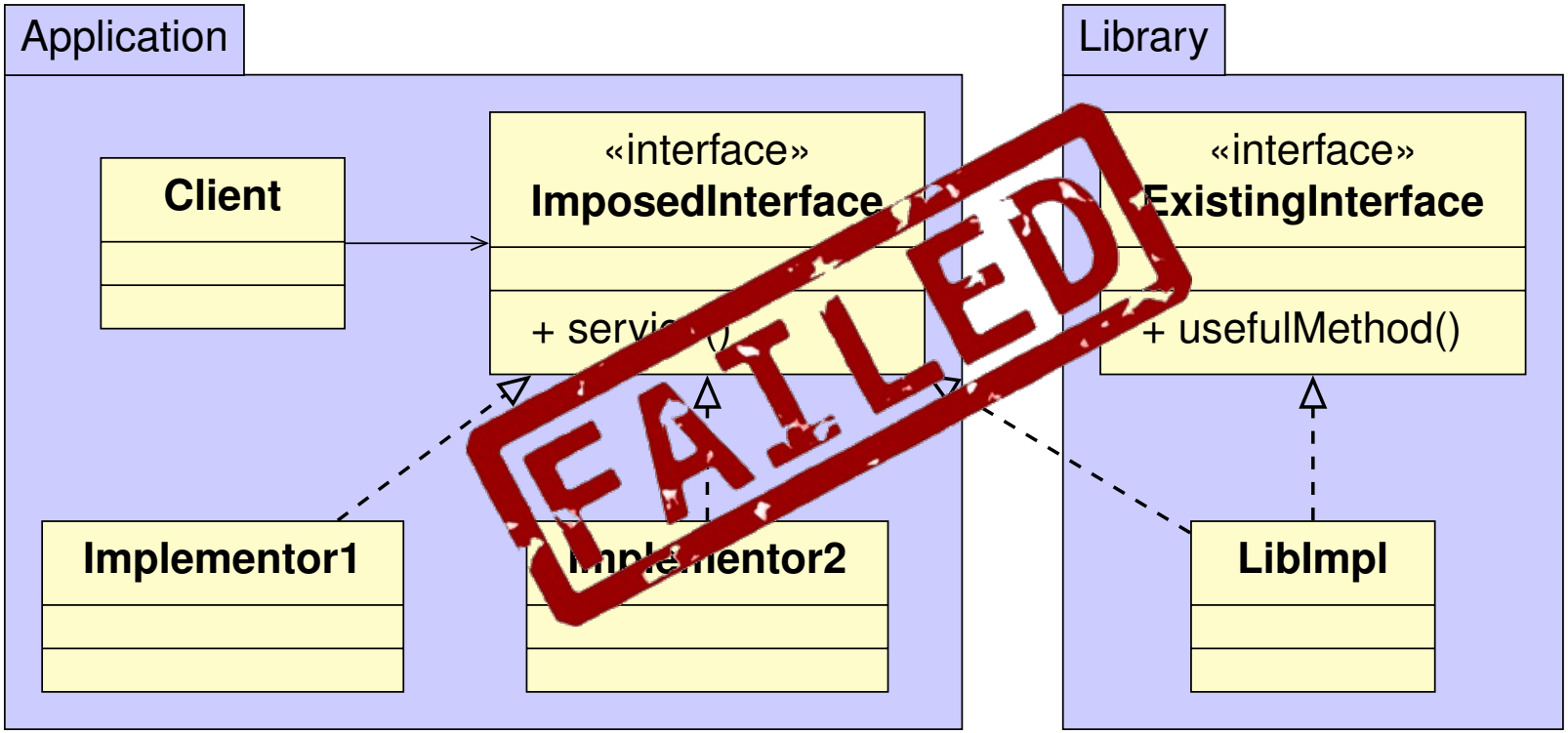
# Une première solution

## Modifier le code existant



# Une première solution

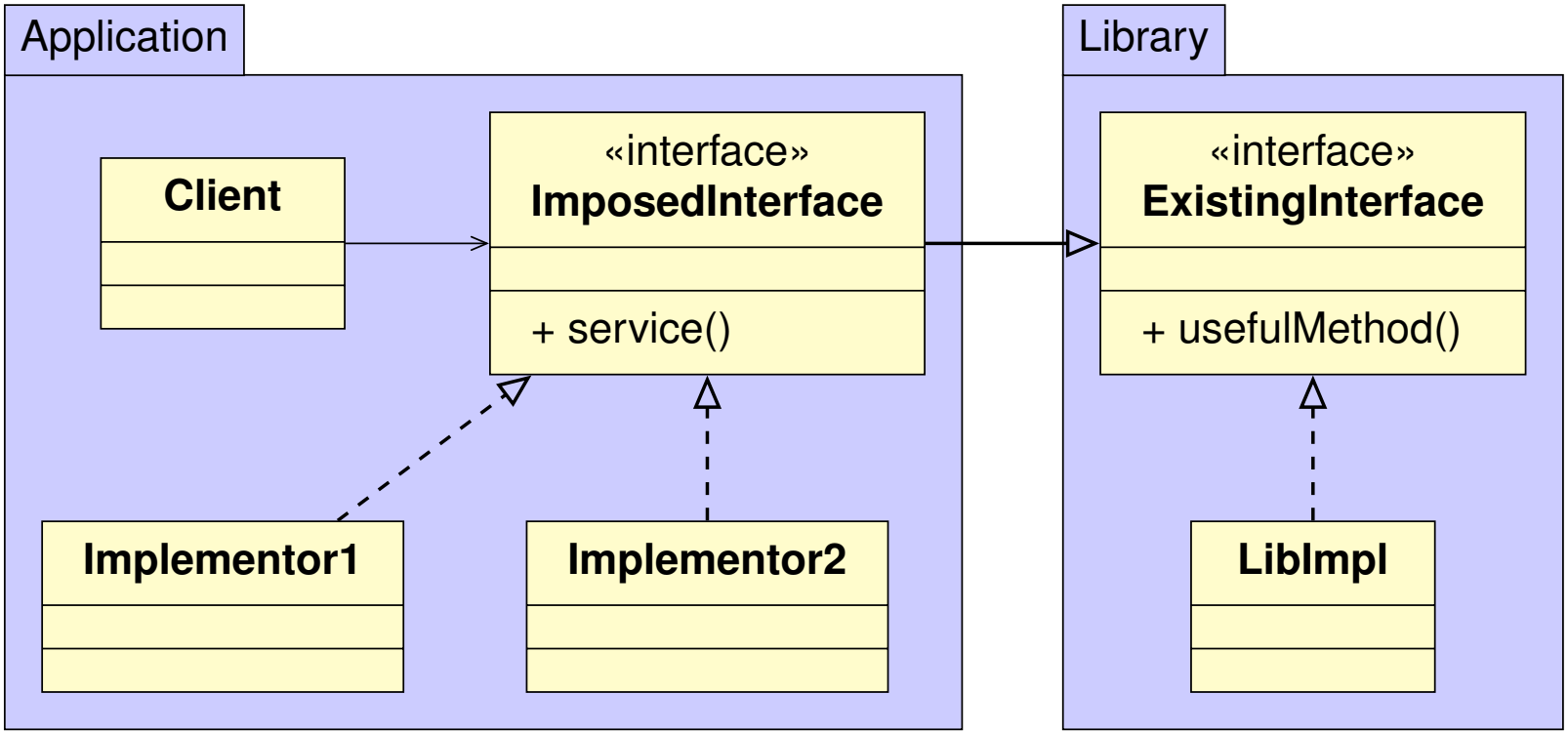
## Modifier le code existant




- Ce faisant, nous violons l'**OCP**, le **SRP** et au mieux l'**ISP** (pollution de l'interface publique des implémenteurs de la bibliothèque),

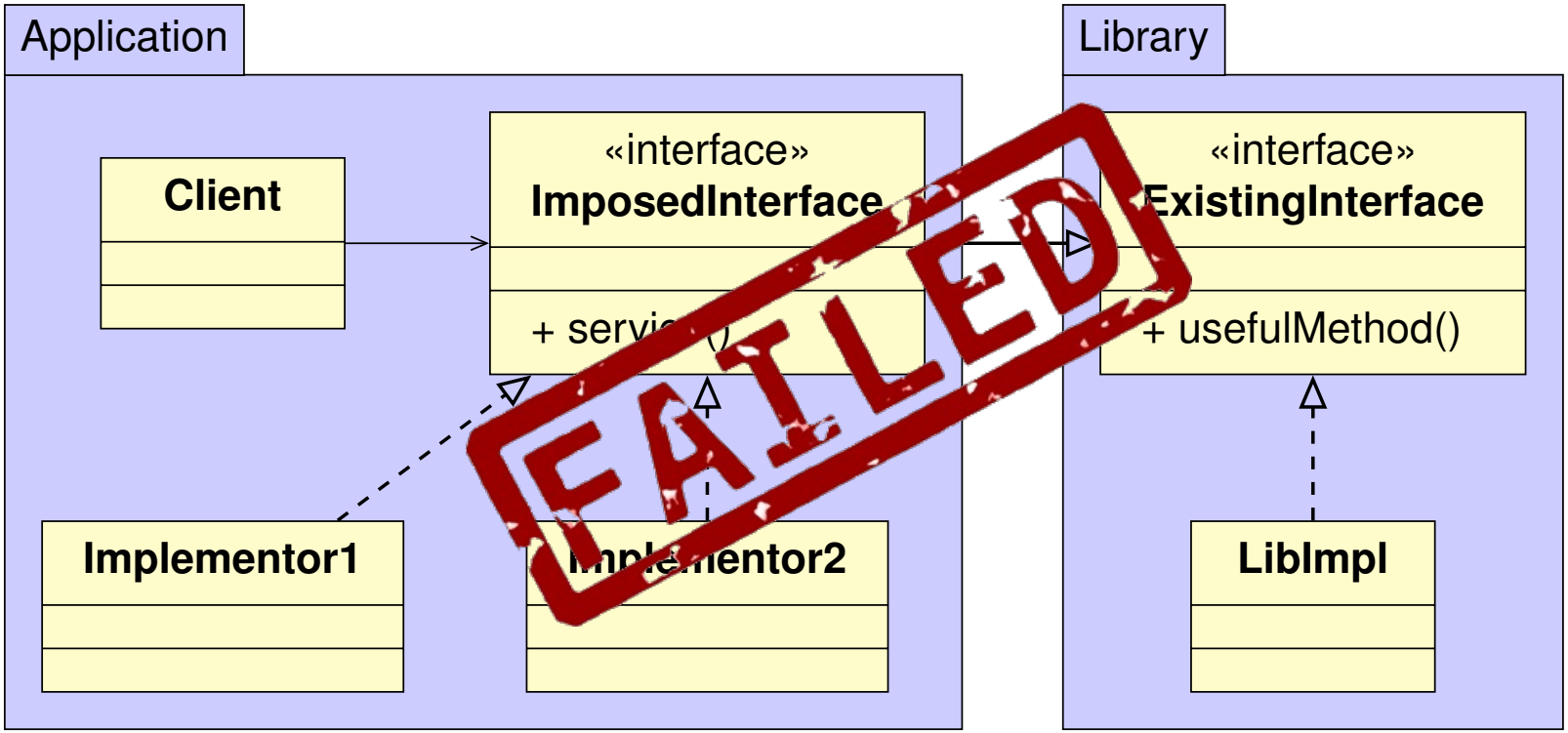
# Une seconde solution

## Adapter l'application



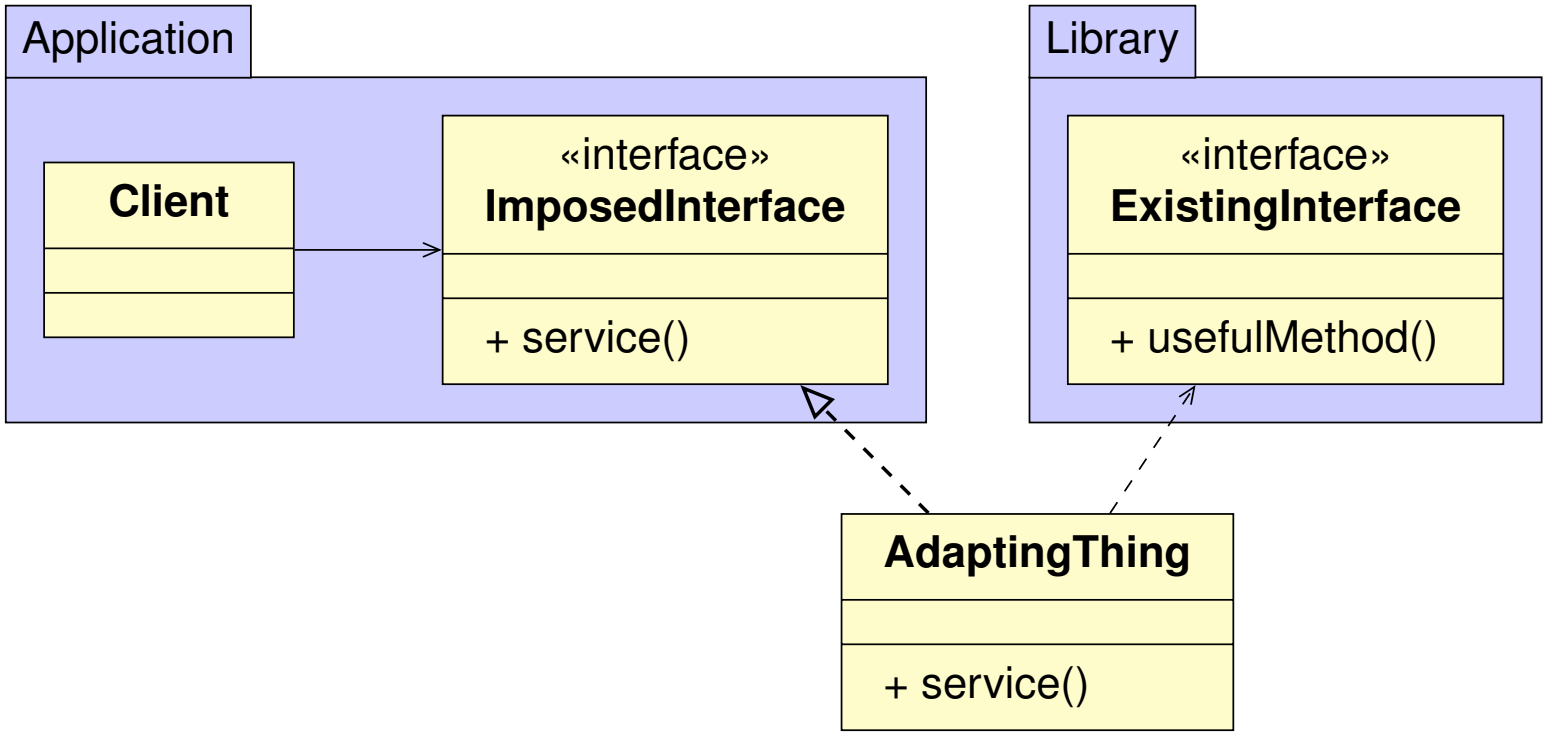
# Une seconde solution

## Adapter l'application




- Ce faisant, nous violons l'**ISP**, et au mieux l'**IDP** (`ImposedInterface` a été conçue pour les besoins de l'application).

# Adapter !




- Une solution propre consiste à ajouter un nouveau composant dans l'application qui laissera inchangés les composants et les spécifications existants.
- C'est ce que permet de réaliser le patron Adapter .

# Le patron **Adapter**

## Aussi connu comme

Wrapper

## Intention

- Adapter une classe existante à une interface imposée.
- Adapter permet à des classes de coopérer alors que leurs interfaces sont incompatibles.

## Motivation

- Réutilisation d'une boite à outils dont l'interface n'est pas compatible avec celle conçue pour l'application.
- La boite à outils ne peut être modifiée (elle est publique $\rightarrow$ OCP) ou nous n'avons pas son code source.

# Participants du patron **Adapter**

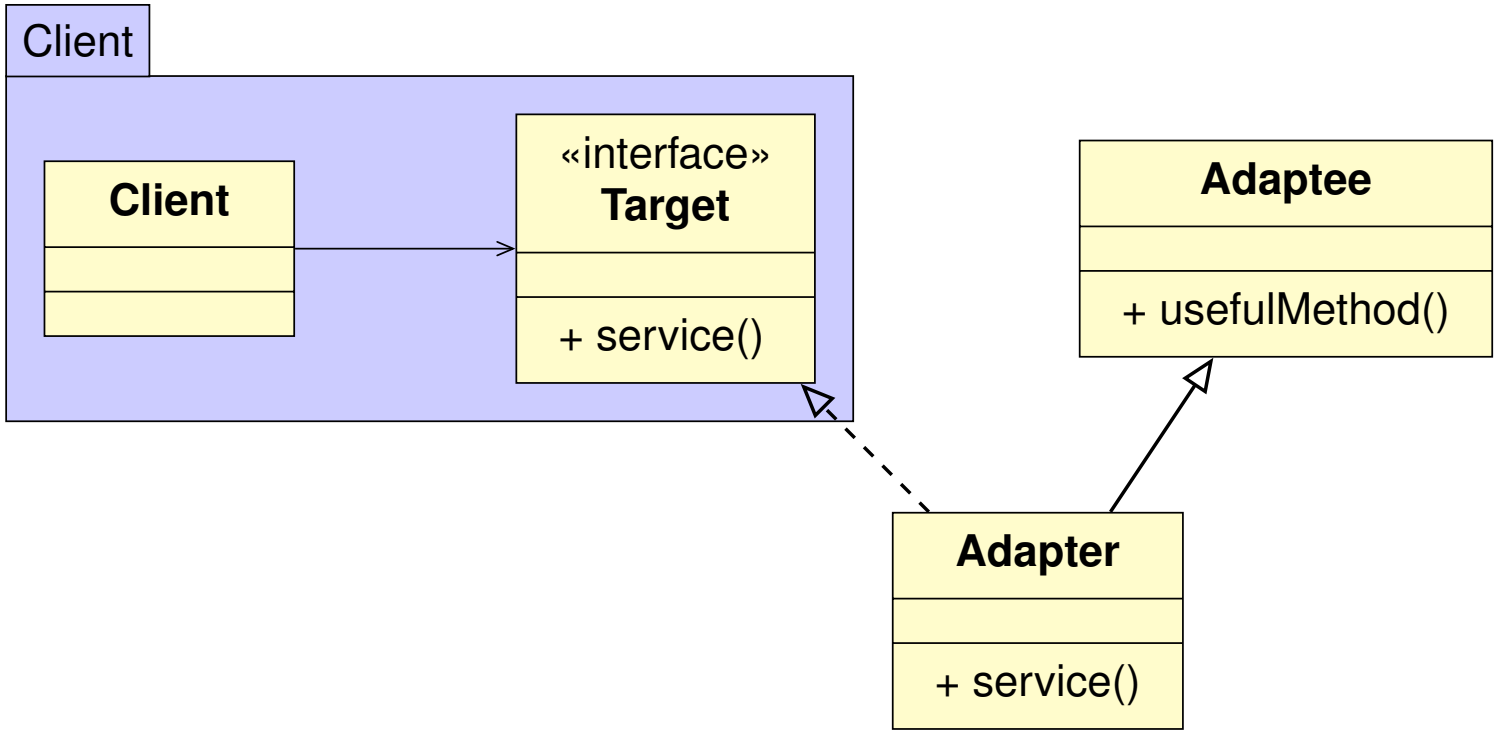
- **Client** : Utilise les objets conformément à l'interface `Target`.
- **Target** : Définit l'interface spécifique au domaine du Client.
- **Adaptee** : Interface ou classe existante qui doit être adaptée à l'interface `Target`.
- **Adapter** : Réalisation qui adapte l'interface `Adaptee` à l'interface cible `Target`.

# Différent types d'adaptateur

- Deux grands types d'adaptateurs : de classe, d'objets.
- Types secondaires : contraint, « Two way ».
- Le schéma de principe diffère légèrement pour chacun, mais le principe global reste le même.
- Chaque type possède des avantages suivant le contexte...

# Adaptateur de classe

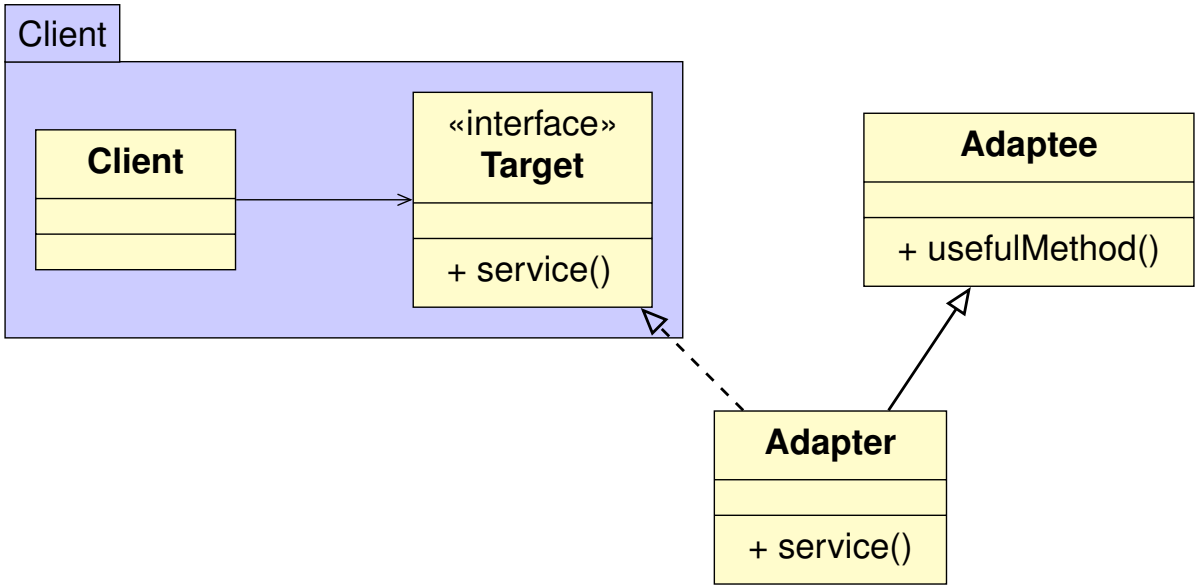
## Schéma de principe



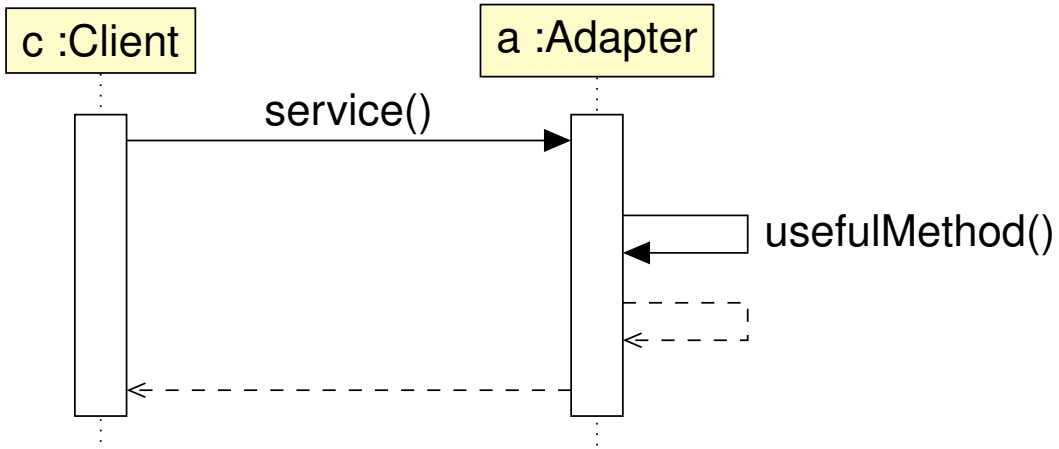

- Dans cette version, c'est la classe qui est adaptée : on instanciera `Adapter` plutôt que `Adaptee`.

# Adaptateur de classe

## Schéma de principe

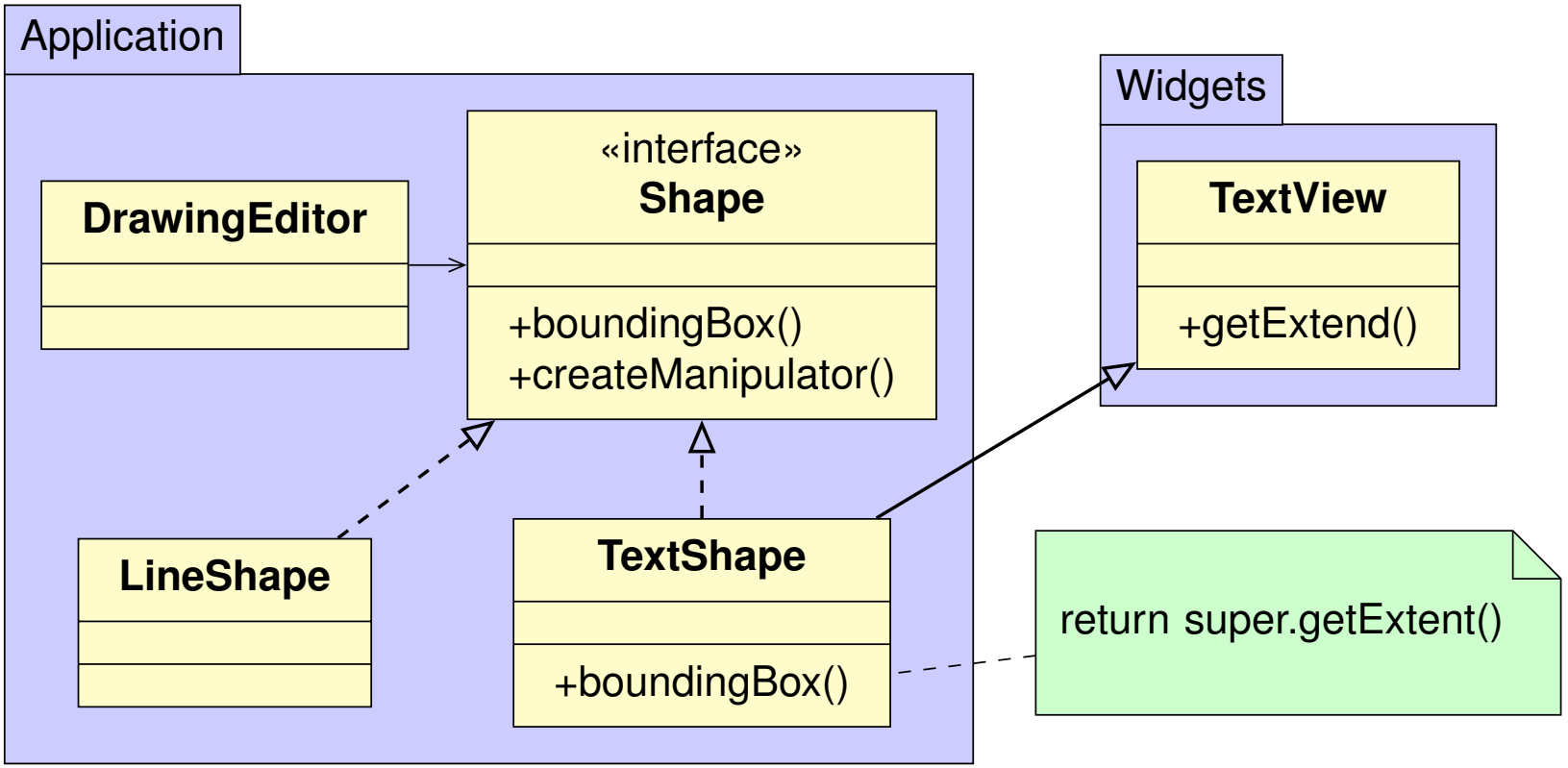



À l'exécution, on aura :



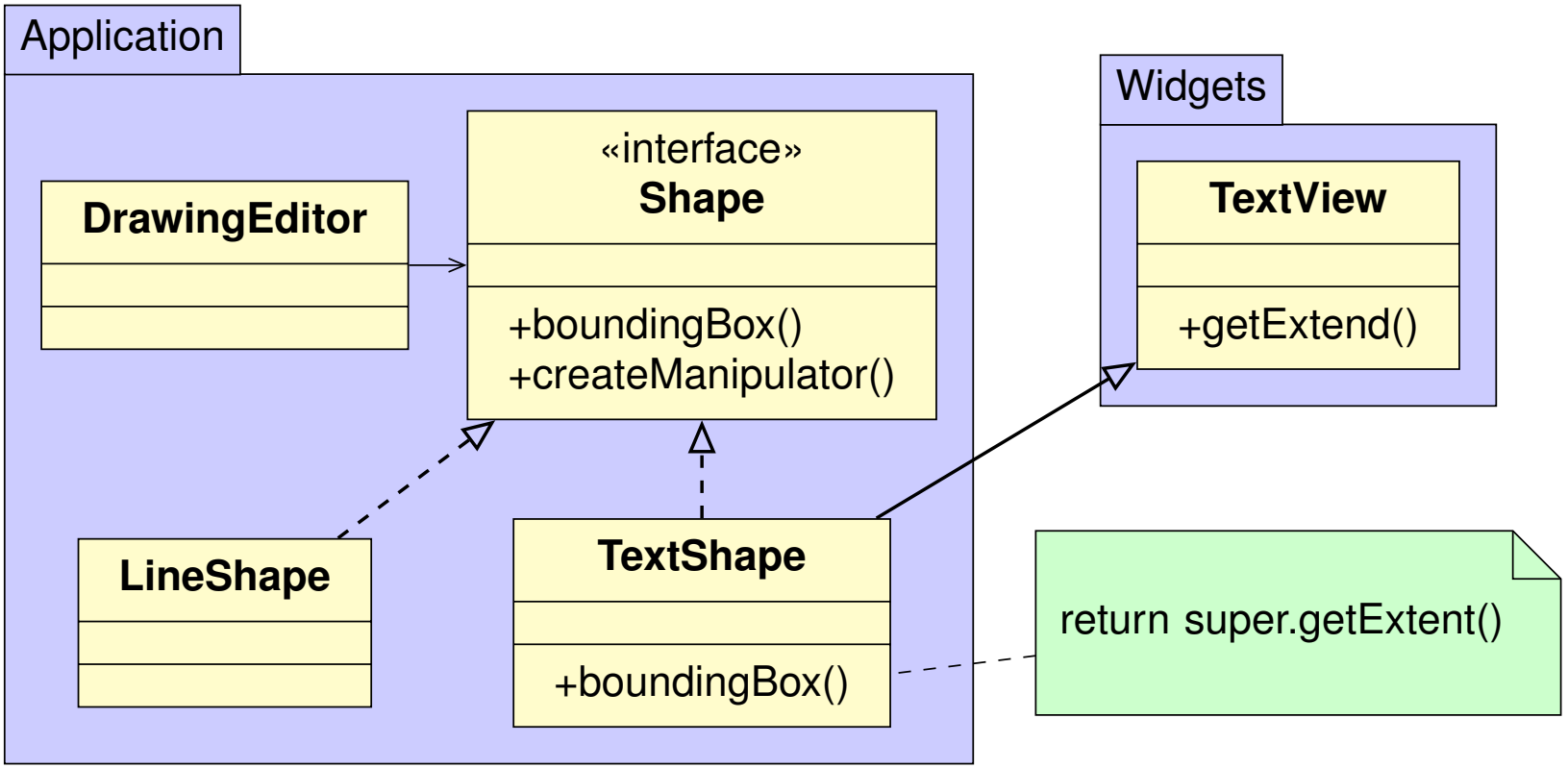
# Adaptateur de classe

## Exemple



# Adaptateur de classe

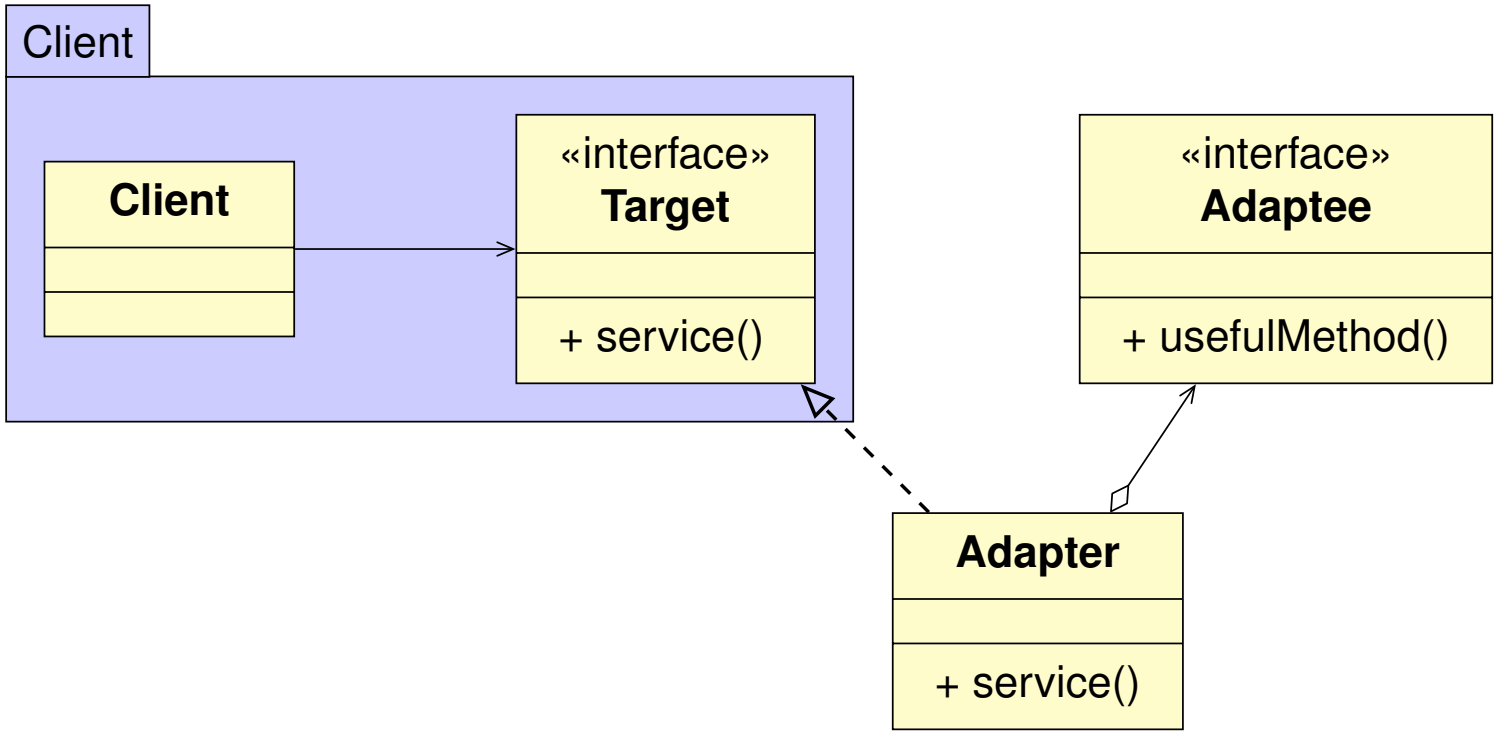
## Exemple




- Sauf qu'une forme n'est pas tout à fait une vue !

# Adaptateur d'objet

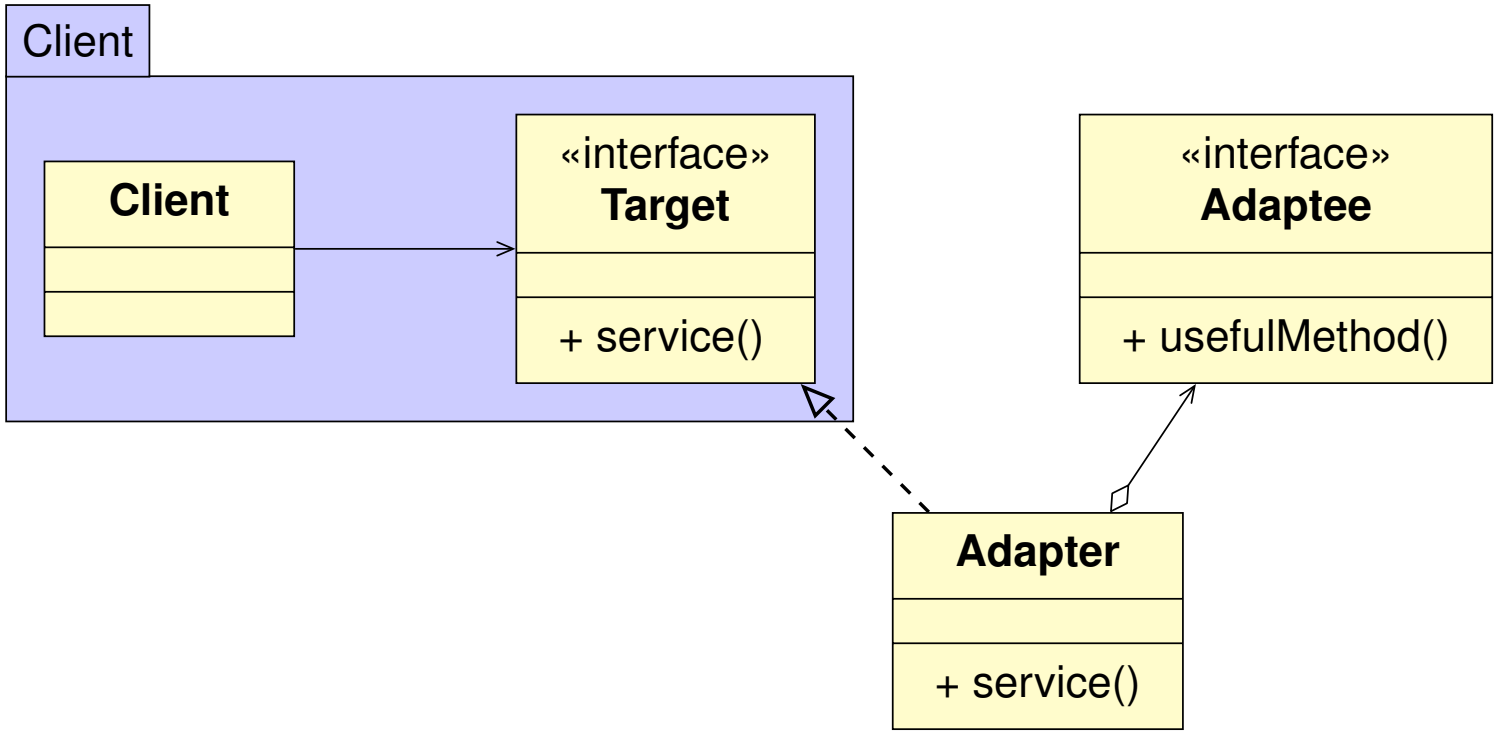
## Schéma de principe



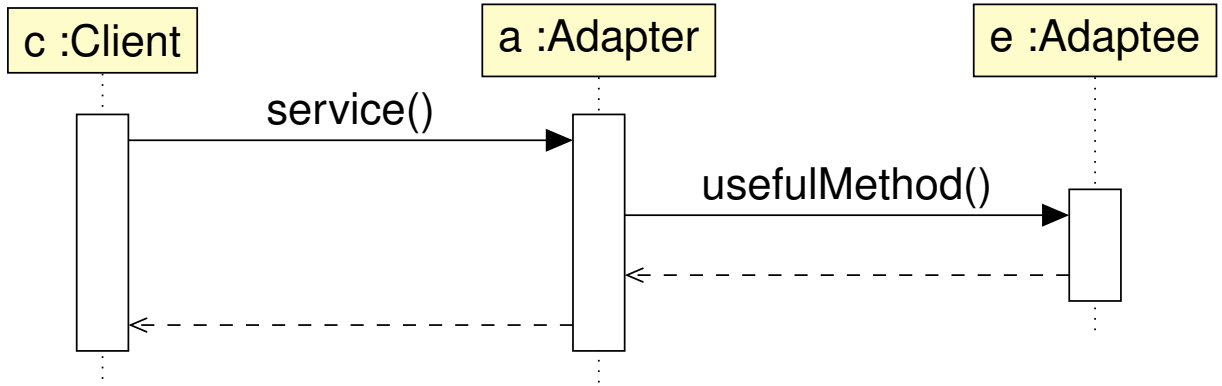

- L'adaptateur est une réalisation et une « composition »,
- `Adapter` déléguera à `Adaptee`.
- `Adaptee` peut être une interface cette fois.

# Adaptateur d'objet

## Schéma de principe

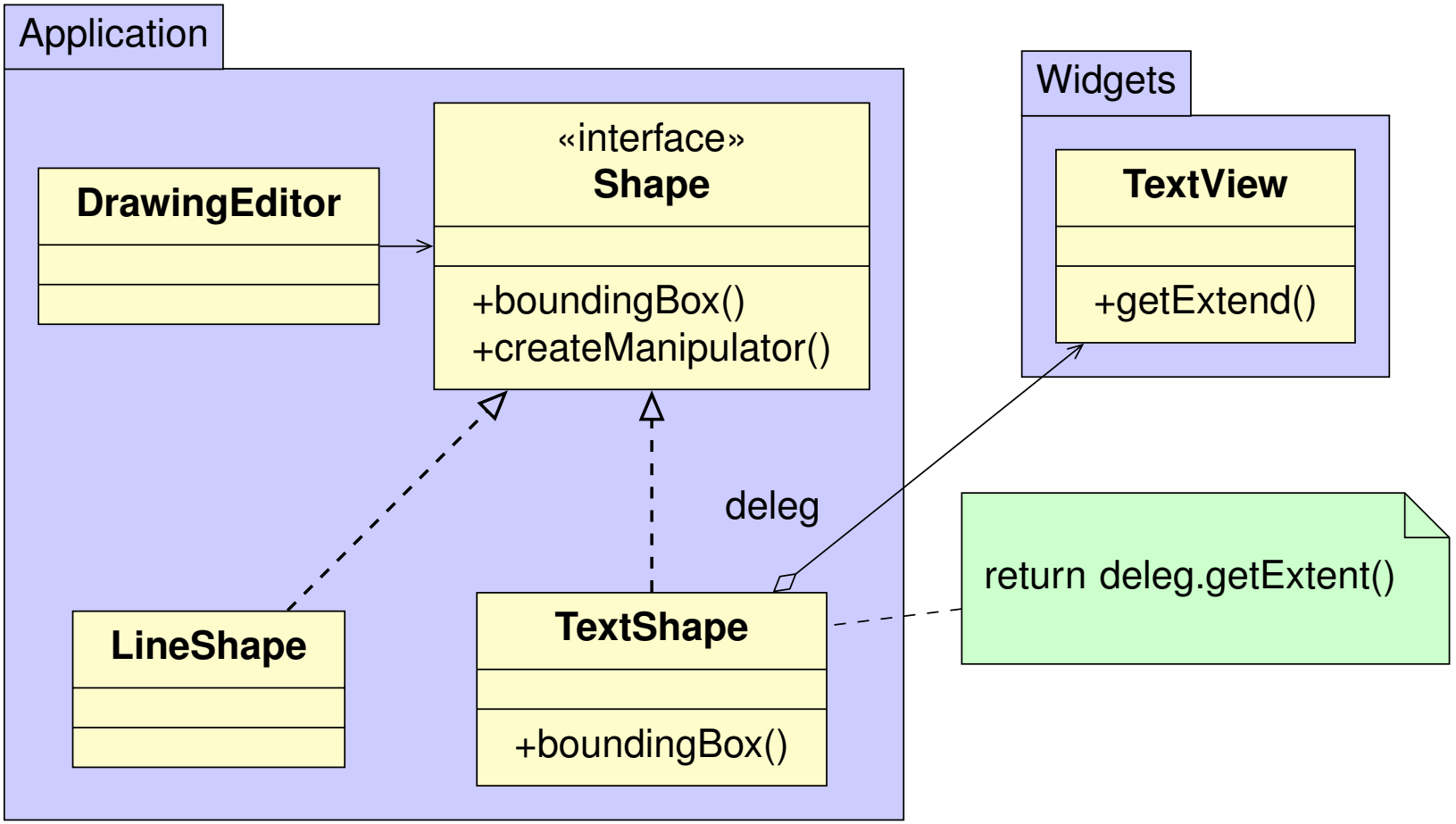



À l'exécution, on aura cette fois :



# Adaptateur d'objet

## Exemple

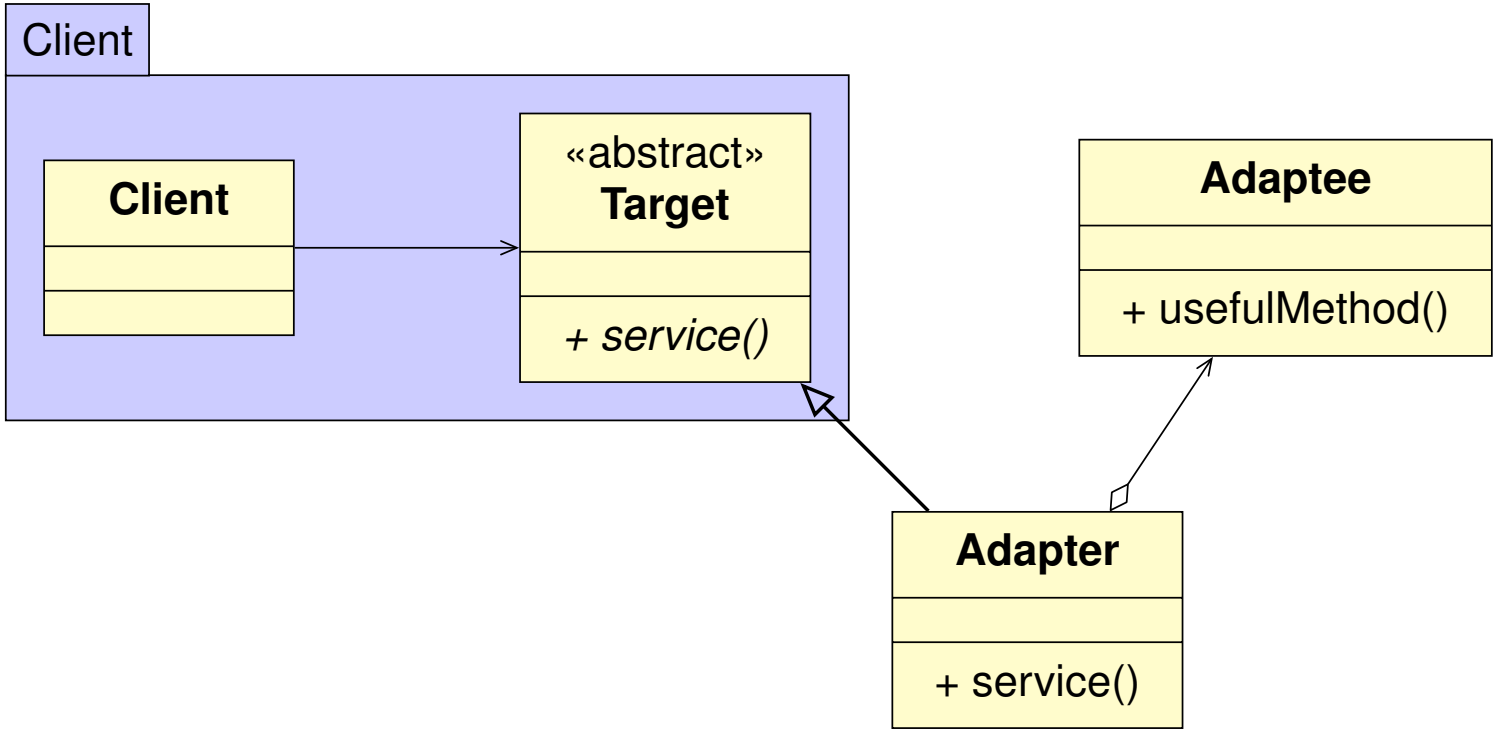



- L'adaptateur délègue à l'adapté,
- il est possible de remplacer l'adapté sans impacter le client.

# Adaptateur (d'objet) contraint

## Schéma de principe

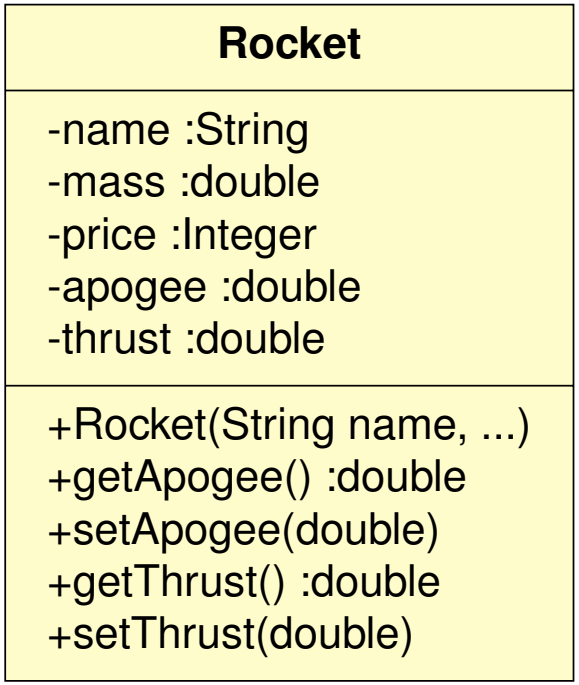
- Il peut arriver que l'interface `Target` soit imposée sous forme de classe abstraite (ou non).
- En l'absence d'héritage multiple, la délégation devient obligatoire.



# Exemple 1

## Feux d'artifices

- Une application de vente de feux d'artifice (fusée).
- On dispose de la classe `Rocket` pour représenter une fusée

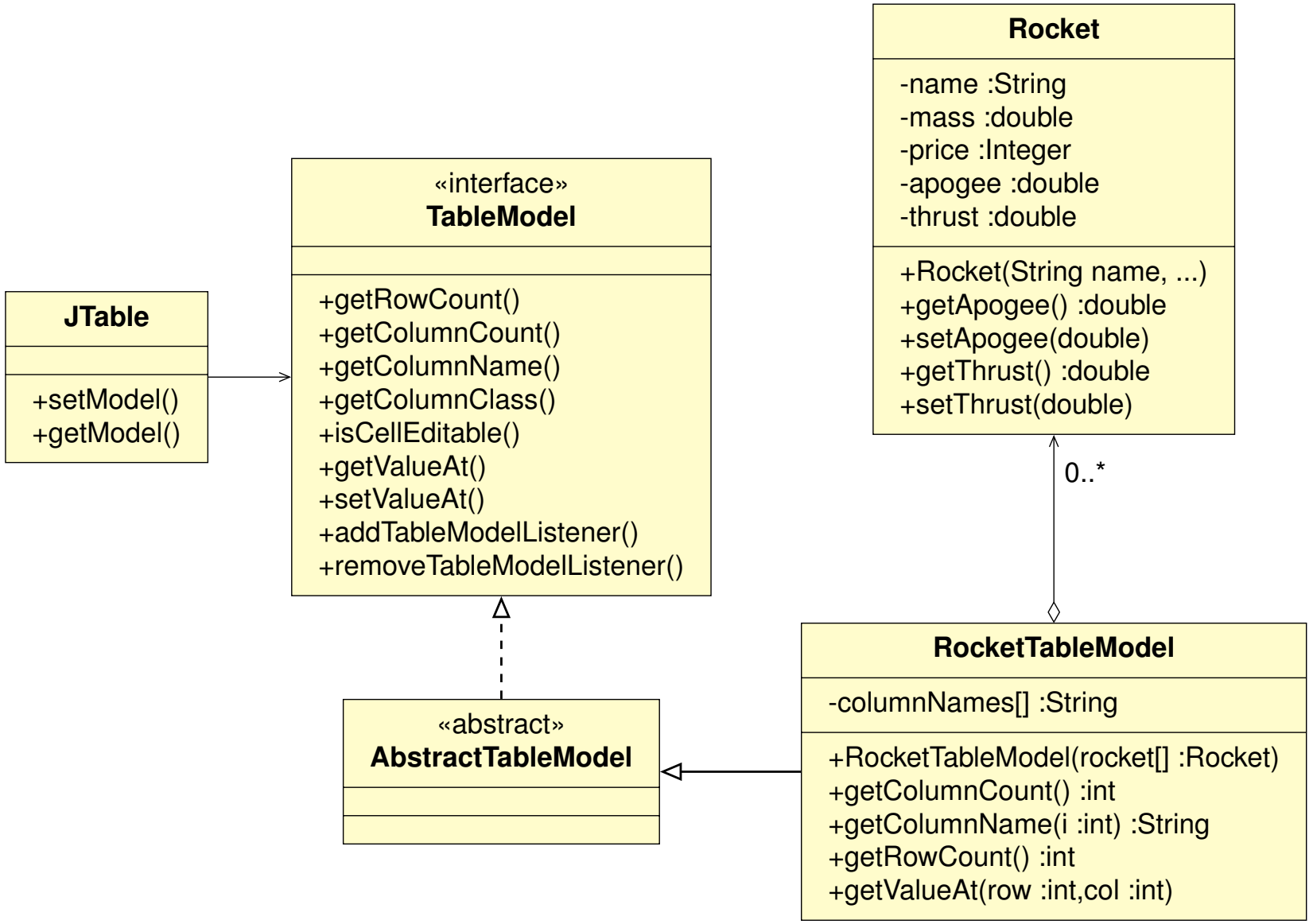



- On désire naturellement une IHM pour lister les fusées grâce à une JTable :

| Name | Price | Apogee |
|---|---|---|
| Shooter | $3.95 | 50.0 |
| Orbit | $29.03 | 5000.0 |

# Exemple 1

## Feux d'artifices : solution globale



# Exemple 1

## Feux d'artifices : code

```
1  package fire;
2
3  public class Rocket {
4       private String name;      private double mass;
5       private Integer price;    private double apogee;
6       private double thrust; // poussee
7
8       public Rocket(String name, double mass, Integer price, double apogee,
                double thrust) {
9            this.name = name;           this.mass = mass;
10           this.price = price;         this.apogee = apogee;
11           this.thrust = thrust;
12      }
13      // The height (in meters) that the rocket is expected to reach.
14      public double getApogee() { return apogee; }
15      public void setApogee(double value) { apogee = value; }
16      // The rated thrust (or force, in newtons) of this rocket.
17      public double getThrust() {return thrust;}
18      public void setThrust(double value) {thrust = value;}
19
20      public String getName() { return name; }
21      public double getMass() { return mass; }
22      public Integer getPrice() { return price; }
23 }
```

# Exemple 1

## Feux d'artifices : code

```
1  package adapter;
2  import javax.swing.table.*;
3  import fire.Rocket;
4
5  // Adapt a collection of rockets for display in a JTable.
6  public class RocketTableModel extends AbstractTableModel {
7       protected Rocket[] rockets;
8       protected String[] columnNames = new String[]{"Name","Price","Apogee"};
9
10      // Construct a rocket table from an array of rockets.
11      public RocketTableModel(Rocket[] rockets) {
12           this.rockets = rockets;
13      }
14
15      // Return the number of columns in this table.
16      public int getColumnCount() {
17           return columnNames.length;
18      }
19
20      // Return the name of the indicated column.
21      public String getColumnName(int i) {
22           return columnNames[i];
23      }
24
25      ...
```

# Exemple 1

## Feux d'artifices : code

```
1       ...
2
3       // Return the number of rows in this table.
4       public int getRowCount() {
5           return rockets.length;
6       }
7
8       // Return the value at the indicated row and column.
9       public Object getValueAt(int row, int col) {
10          switch (col) {
11              case 0: return rockets[row].getName();
12              case 1: return rockets[row].getPrice();
13              case 2: return new Double(rockets[row].getApogee());
14              default: return null;
15          }
16      }
17  }
```

# Exemple 1

## Feux d'artifices : code

```
1  package FireShop;
2  import javax.fire.adapter.*;
3  import fire.Rocket;
4
5  // Demonstration class.
6  public class ShowRocketTable {
7       private static RocketTableModel getRocketTable() {
8            Rocket r1 = new Rocket("Shooter", 1.0, new Integer(395), 50.0, 4.5);
9            Rocket r2 = new Rocket("Orbit", 2.0, new Integer(2903), 5000, 3.2);
10           return new RocketTableModel(new Rocket[] { r1, r2 });
11      }
12
13      // Display a Swing component.
14      public static void display(Component c, String title) {
15           JFrame frame = new JFrame(title);
16           frame.getContentPane().add(c);
17           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18           frame.pack();
19           frame.setVisible(true);
20      }
21
22      public static void main(String[] args) {
23           JTable table = new JTable(getRocketTable());
24           JScrollPane pane = new JScrollPane(table);
25           pane.setPreferredSize(new java.awt.Dimension(300, 100));
26           display(pane, "Rockets");
27      }
28 }
```

# Exemple 2

## Gestionnaire d'événements

- Soit le code suivant :

```
1  ^^|public class ButtonDemo {
2  ^^|     public ButtonDemo() {
3  ^^|          Button button = new Button("Press␣me");
4  ^^|          button.addActionListener(new ActionListener() {
5  ^^|               public void actionPerformed(ActionEvent e) {
6  ^^|                    doOperation();
7  ^^|               }
8  ^^|          });
9  ^^|     }
10 ^^|     public void doOperation() {
11 ^^|          // whatever
12 ^^|     }
13 ^^|}
14 ^^|
```

# Exemple 2

## Gestionnaire d'événements

- Soit le code suivant :

```
1  ^^|public class ButtonDemo {
2  ^^|     public ButtonDemo() {
3  ^^|          Button button = new Button("Press me");
4  ^^|          button.addActionListener(new ActionListener() {
5  ^^|               public void actionPerformed(ActionEvent e) {
6  ^^|                    doOperation();
7  ^^|               }
8  ^^|          });
9  ^^|     }
10 ^^|     public void doOperation() {
11 ^^|          // whatever
12 ^^|     }
13 ^^|}
14 ^^|
```

- **Question :** où est le patron adaptateur[2] ?

2. Pour commencer où est la classe adaptateur ?

# Exemple 2

## Gestionnaire d'événements

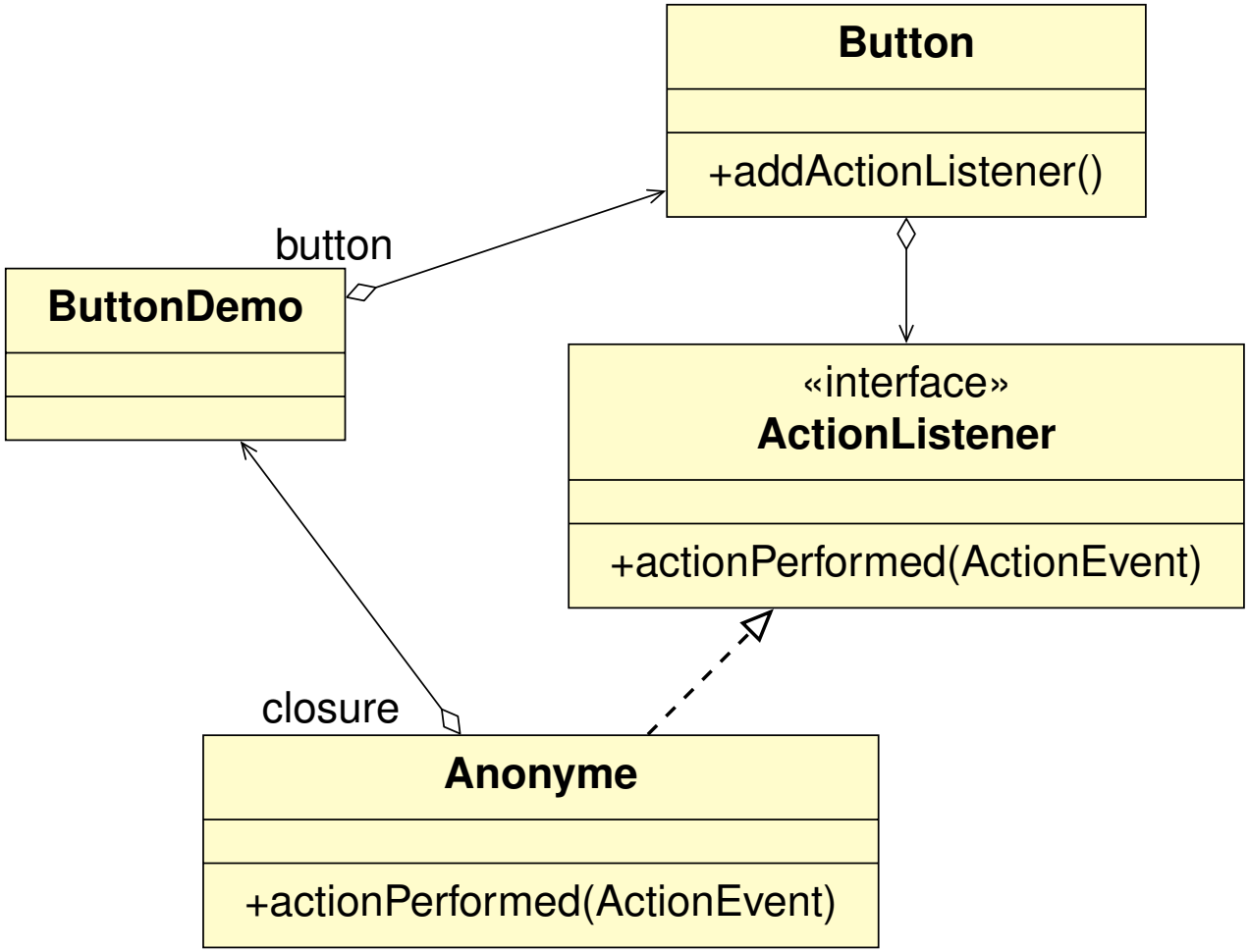
- Soit le code suivant :

```
1  ^^|public class ButtonDemo {
2  ^^|     public ButtonDemo() {
3  ^^|          Button button = new Button("Press me");
4  ^^|          button.addActionListener(new ActionListener() {
5  ^^|               public void actionPerformed(ActionEvent e) {
6  ^^|                    doOperation();
7  ^^|               }
8  ^^|          });
9  ^^|     }
10 ^^|     public void doOperation() {
11 ^^|          // whatever
12 ^^|     }
13 ^^|}
14 ^^|
```

- *Indice :* la syntaxe `new ActionListener() { ...}` correspond à l'instanciation d'une classe anonyme obtenue par héritage depuis `ActionListener`.

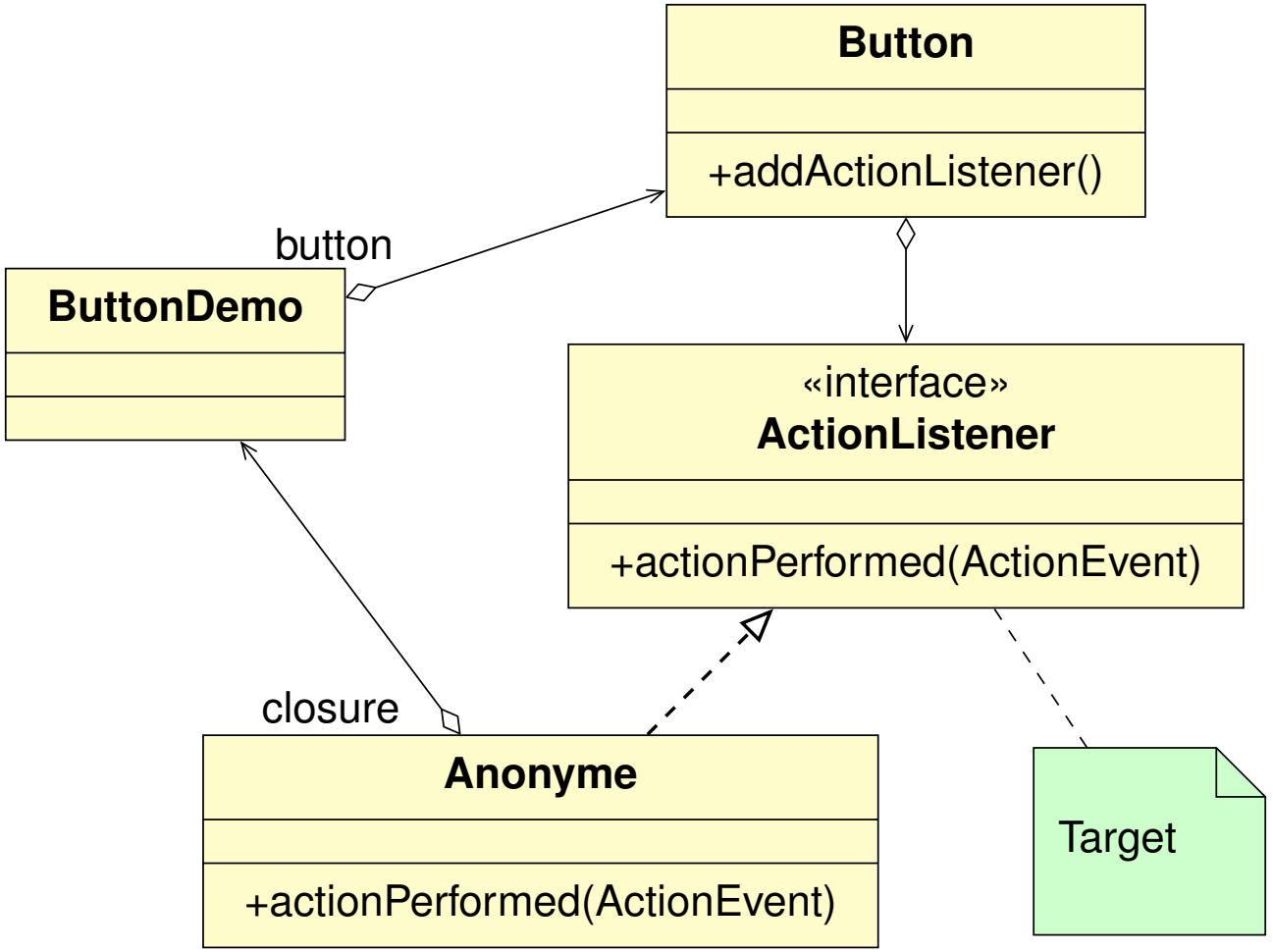
# Exemple 2

## Gestionnaire d'événements



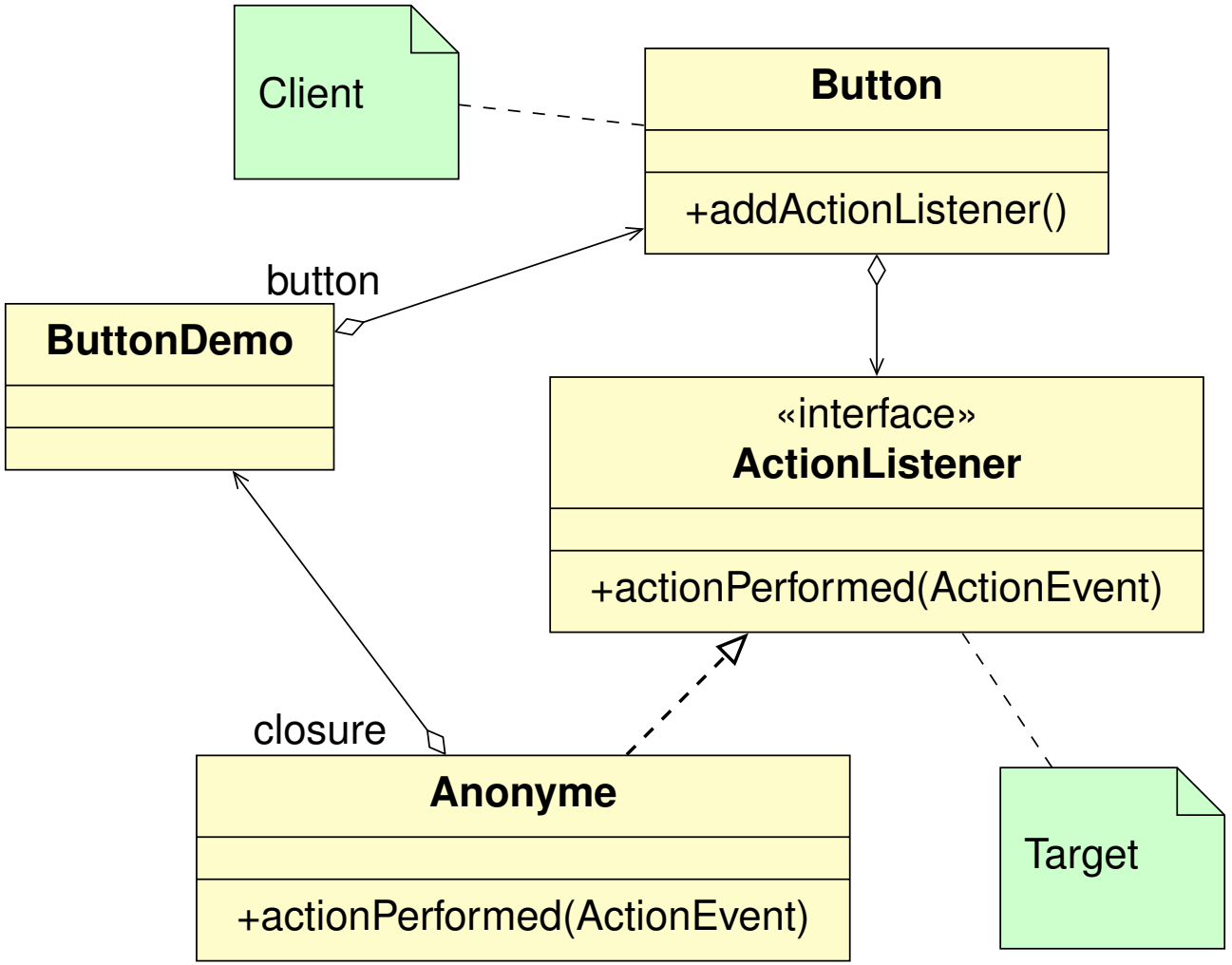
# Exemple 2

## Gestionnaire d'événements



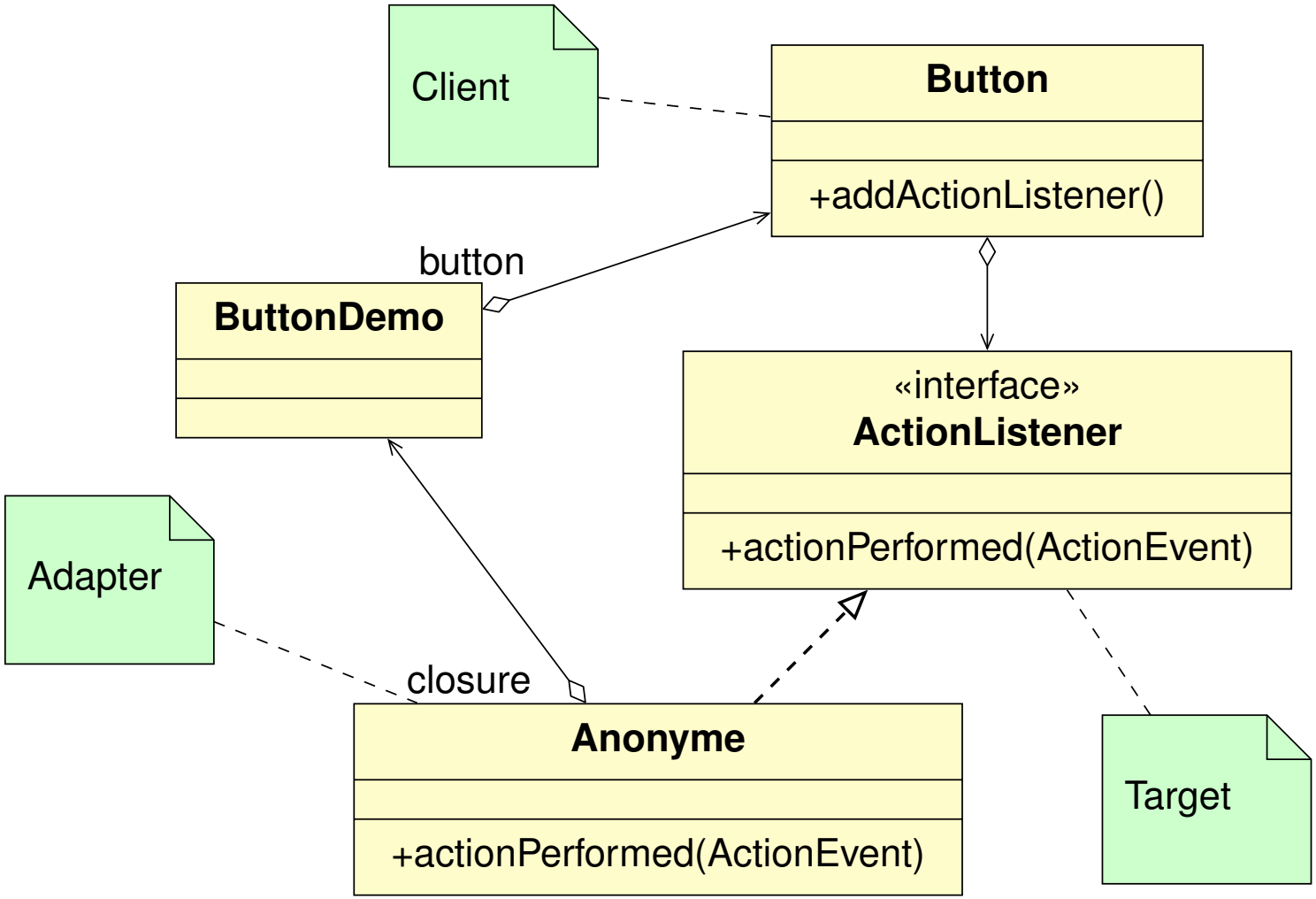
# Exemple 2

## Gestionnaire d'événements



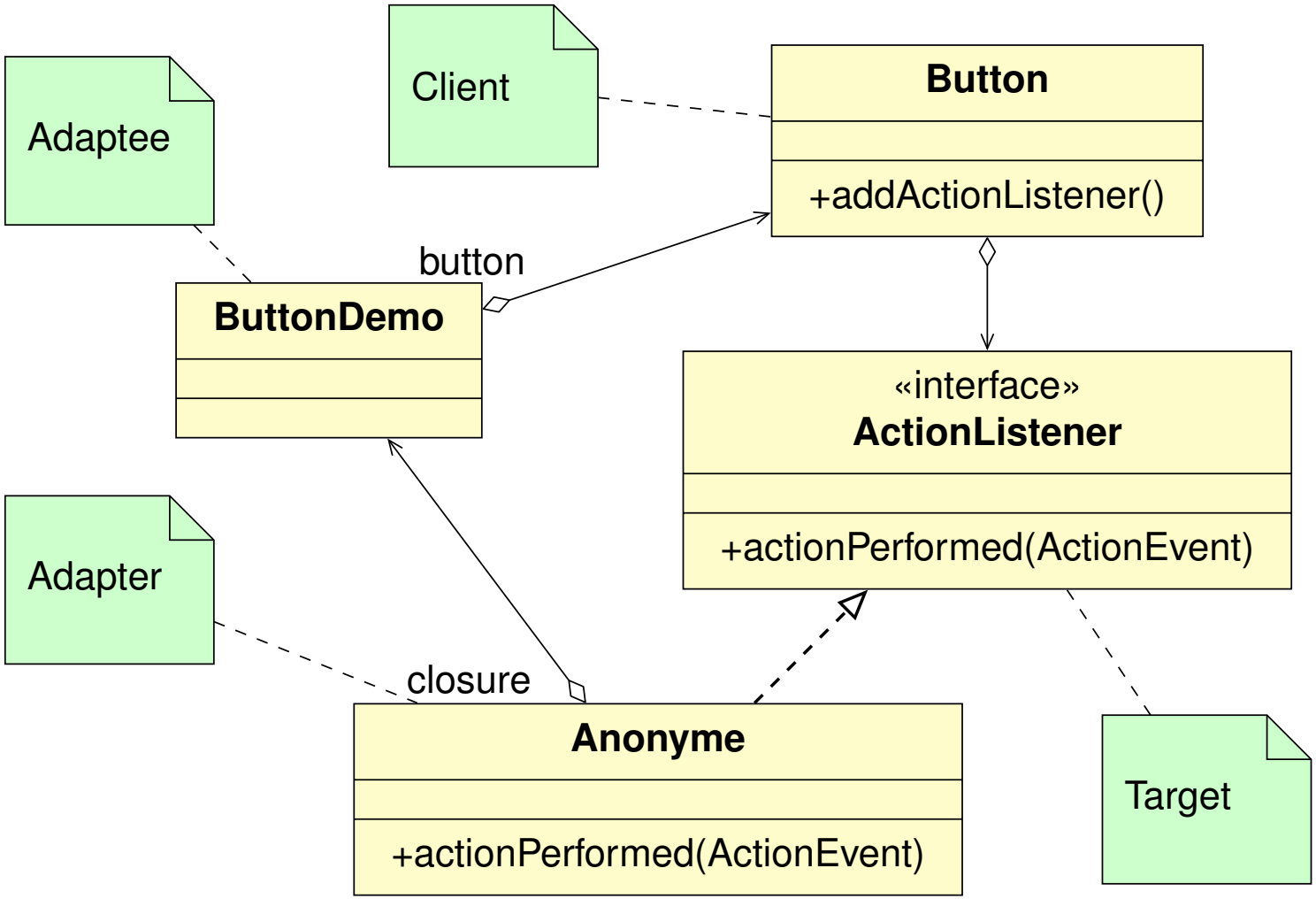
# Exemple 2

## Gestionnaire d'événements



# Exemple 2

## Gestionnaire d'événements



# Exemple 2

## Gestionnaire d'événements

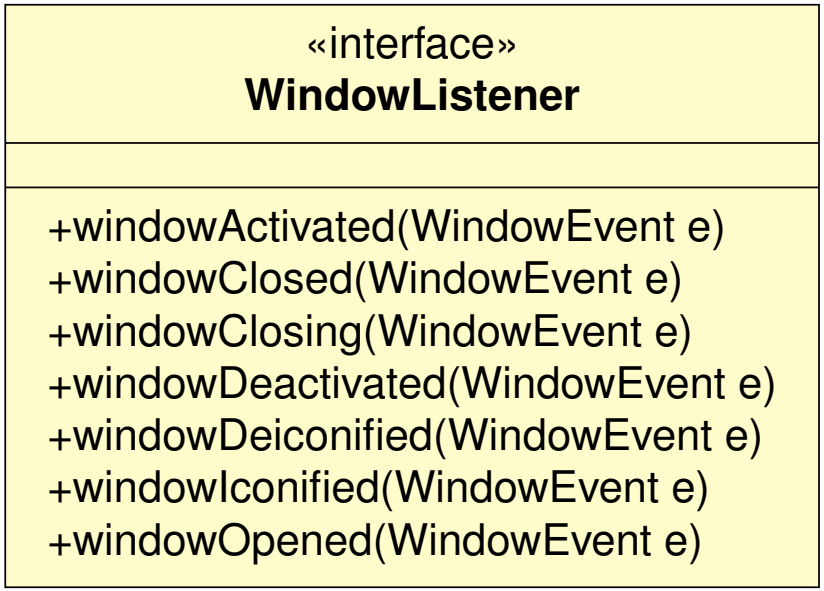
- Les participants dans le code :

```
1  ^^|public class ButtonDemo {^^| // ADAPTEE
2  ^^|     public ButtonDemo() {
3  ^^|         // CLIENT :
4  ^^|         Button button = new Button("Press␣me");
5  ^^|
6  ^^|         button.addActionListener(
7  ^^|             // ADAPTER : anonymous
8  ^^|             new ActionListener() { // TARGET : implicit inherit
9  ^^|             public void actionPerformed(ActionEvent e) {
10 ^^|                 // A closure is hapening here :
11 ^^|                 doOperation();
12 ^^|             }
13 ^^|         });
14 ^^|     }
15 ^^|     public void doOperation() {
16 ^^|         // whatever
17 ^^|     }
18 ^^|}
19 ^^|
```

# Exemple 3

## Auditeurs AWT

- Les interfaces de type auditeur (listener) de l'*AWT* possèdent plusieurs méthodes qui doivent toutes être réalisées par un écouteur d'événements.

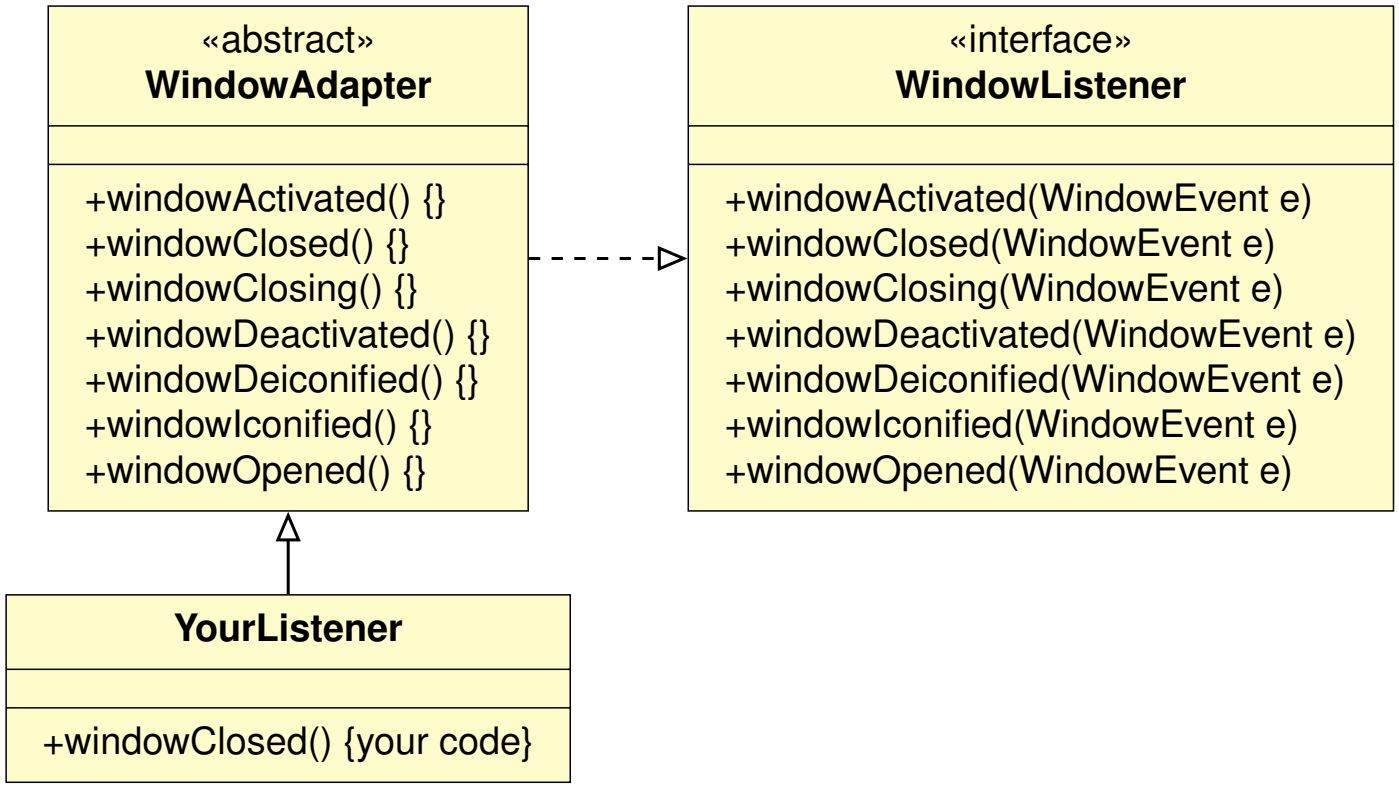



Par exemple l'interface `WindowListener` possède 7 méthodes.
Dans la plupart des cas seules quelques méthodes présentent un réel intérêt, comme celle qui guette l'événement `WindowClosing`.

# Exemple 3

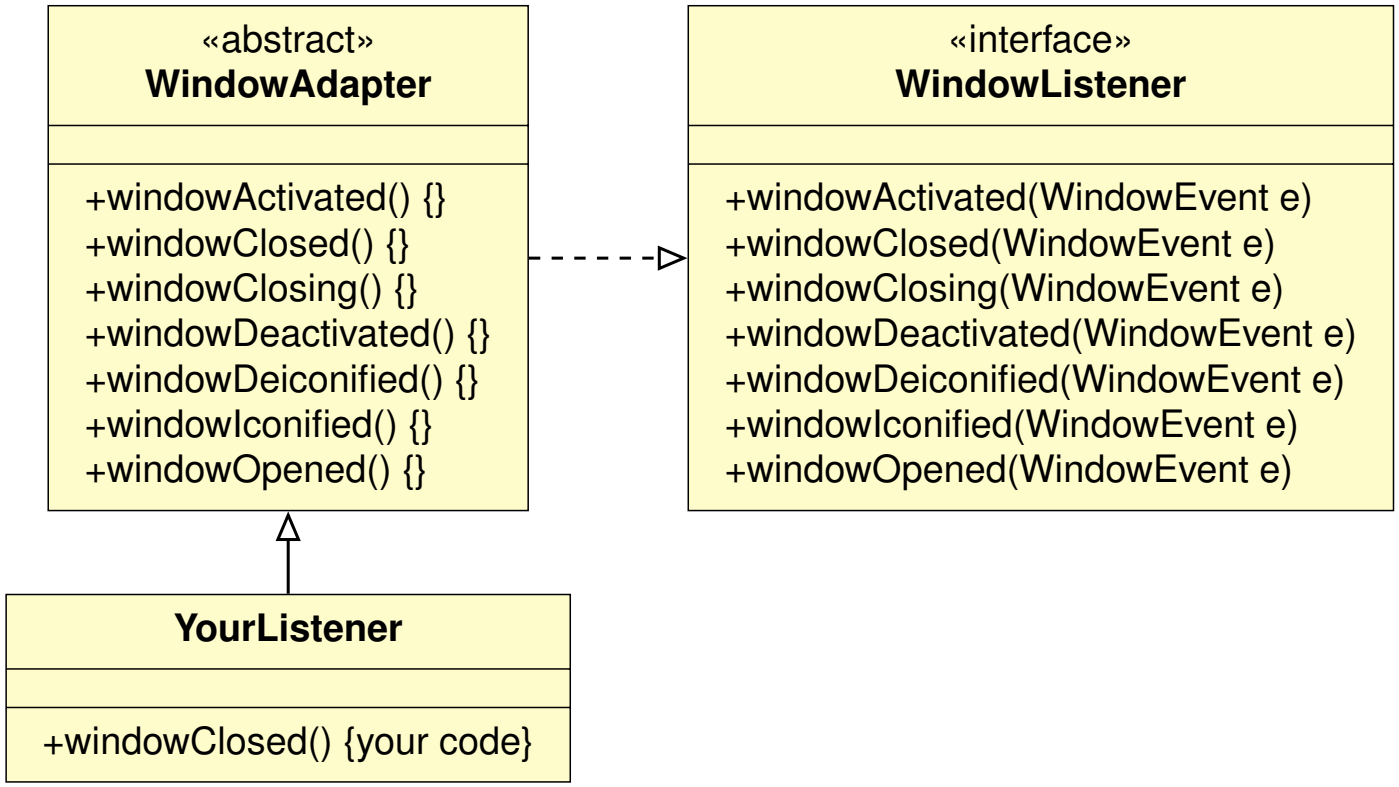
## Auditeurs AWT

- La bibliothèque de Sun propose des classes comme `WindowAdapter` qui implémente `WindowListener` avec des définitions de méthodes vides.



# Exemple 3

## Auditeurs AWT




Faux Ami !

- `WindowAdapter` est une souche et non un adaptateur au sens du patron.
- Il n'adapte pas une interface à une autre.

# Comparaison Adaptateur de classe vs d'objet

## Généricité

L'adaptateur de classe adapte une et une seul classe :

```
1 public class Adapter implements Target extends Adaptee { ...
```

L'adaptateur d'objet adapte toute classe dérivée ou implémenteurs :

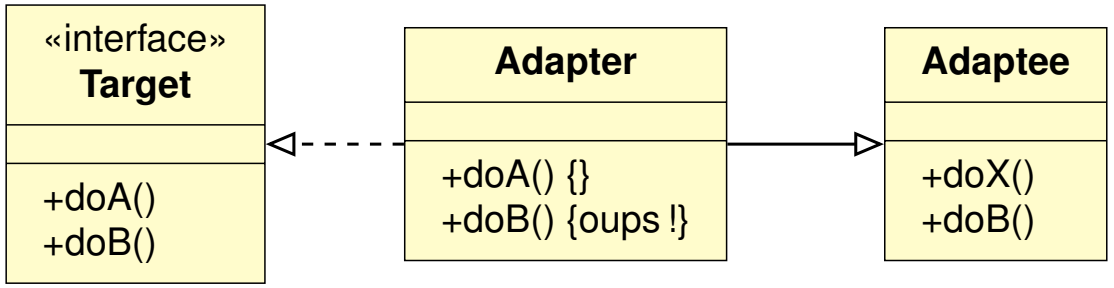
```
1 public class Adapter implements Target {
2     private Adaptee delegate;
3
4     public Adapter(Adaptee theAdaptee) { delegate=theAdaptee; }
5     ...
```

- Object wins !

# Comparaison Adaptateur de classe vs d'objet

## Redéfinitions

- L'adaptateur de classe peut redéfinir des méthodes héritées de l'adapté :

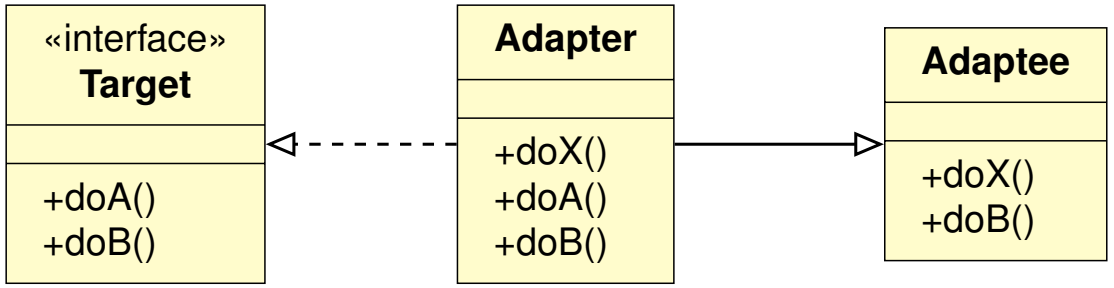



- Puisque les interfaces peuvent être proches, il est possible de redéfinir par accident une méthode de l'adapté (voir chapitre 1).
- L'adaptateur d'objet respect l'encapsulation.
- Object wins !

# Comparaison Adaptateur de classe vs d'objet

## Opacité

- L'interface publique de l'adaptateur de classe contient les méthodes de l'interface imposée mais aussi celles héritées de l'adapté :

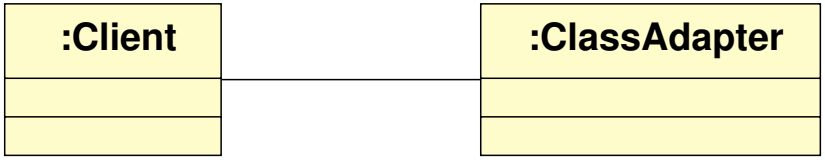



- Même si l'ISP n'est pas violé, l'interface de l'adaptateur est pollué par des synonymes d'opérations.
- Le client peut être « Tenté »d'utiliser cette connaissance visible.
- L'adaptateur d'objet cache ces détails et rend le découplage complet.
- Object wins !

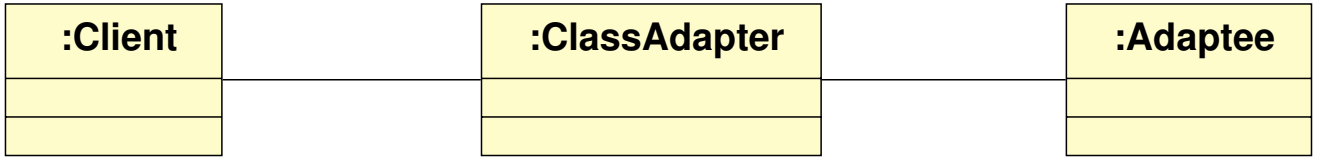
# Comparaison Adaptateur de classe vs d'objet

## Nombre d'objets à l'usage

- L'instance de l'adaptateur de classe se substitue à celle de l'adapté :




- Celle de l'adaptateur d'objet s'ajoute à celle de l'adapté :



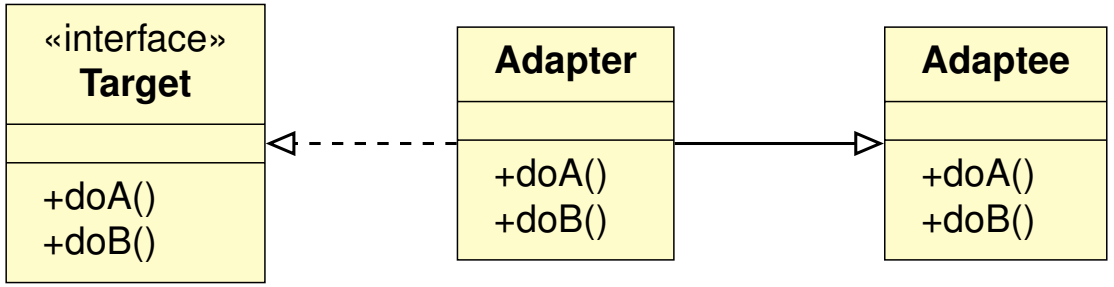

- Une instance peut être économisée si le client gère l'instanciation lui-même.

- Class wins ! (mais attention aux interfaces)

# Comparaison Adaptateur de classe vs d'objet

## Ajout d'interface

- L'adaptateur de classe peut être utilisé pour nommer simplement une interface déjà implémentée par l'adapté :

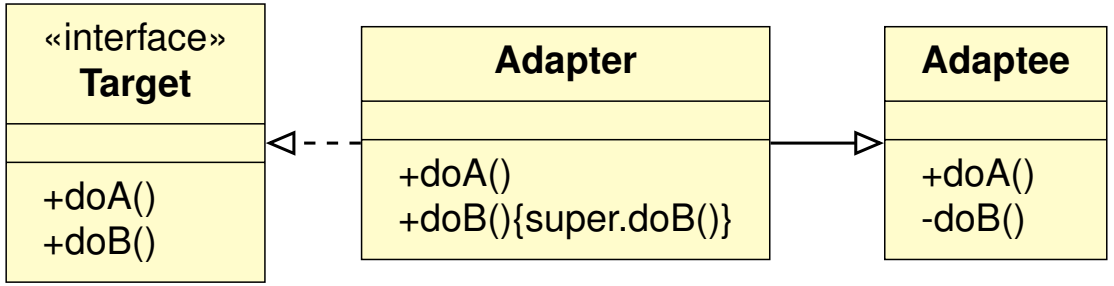



- En effet, mais si les méthodes de la cible sont intégralement implémentées dans l'adaptée, seul la déclaration `implements Target` le rend compatible avec cette interface.
- L'adaptateur de classe corrige ce problème sans ajouter d'objet (si l'instanciation est gérée par le client).
- Class wins !

# Comparaison Adaptateur de classe vs d'objet

## Changement de visibilité

- L'adaptateur de classe peut également suffire pour changer la visibilité d'une méthode (mais seulement de protégé vers publique) :




- Note : ce sont deux méthodes différentes, il est tout de même nécessaire de déléguer.
- Class wins !

# Comparaison Adaptateur de classe vs d'objet

## Contrainte d'instanciation

- Seul l'adaptateur d'objet peut être employé si le client ne gère pas l'instanciation lui-même.
- Ce peut être le cas si l'adapté est obtenu par une *Factory* (cf chapitre 5) :

```
1     public test() {
2          Vessel m = Game.getVesselFactory().createVessel();
3          Sprite s=new VesselSpriteAdapter(m);
4          addToScene(s);
5      }
```

- ou par paramètre :

```
1     public test(Vessel m) {
2          addToScene(new VesselSpriteAdapter(m));
3      }
```

- Object wins !

# Principes respectés

- **O.C.P.** : le code réutilisé n'est pas modifié, les bibliothèques sont conservées intactes et continueront de fonctionner avec fiabilité avec le reste du système.
- **D.I.P.** : `Target` est une abstraction issue de l'univers « métier »du client, `Adaptee` une abstraction de bas niveau fournie par la boîte à outils. Conserver ces deux interfaces intactes contribue à respecter le **D.I.P.**.
- **I.S.P.** : en conservant les interfaces `Target` et `Adaptee` séparée, on évite la pollution d'interface et l'on respect l'**I.S.P.**.