Architecture Logicielle

Quickie – Le Visiteur

Florent Nicart

Université de Rouen

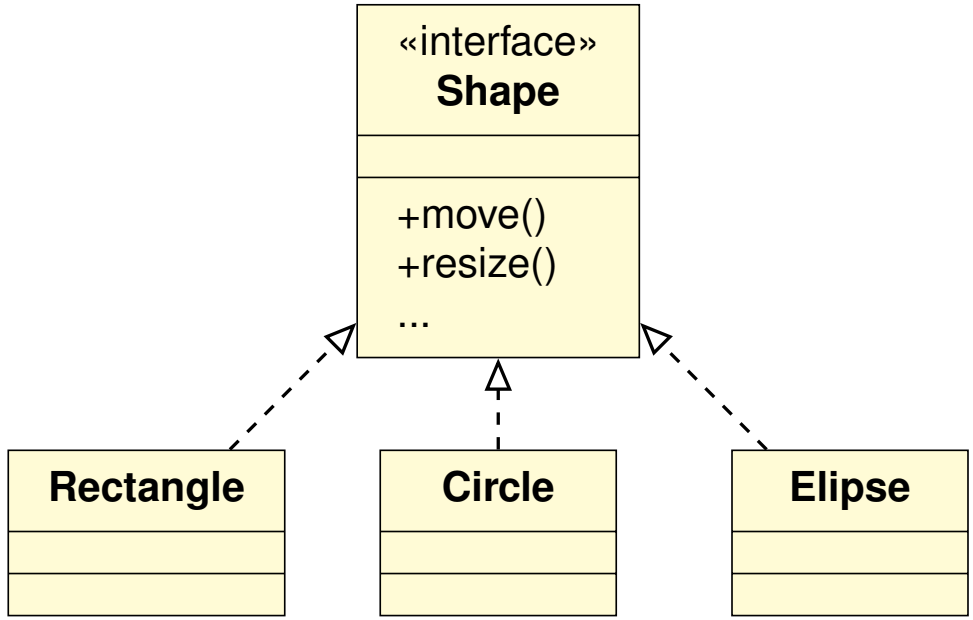
2017–2018

Le patron Visiteur

Permettre l'ajout d'opérations homogènes aux éléments d'un modèle sans couplage.

Situation Initiale

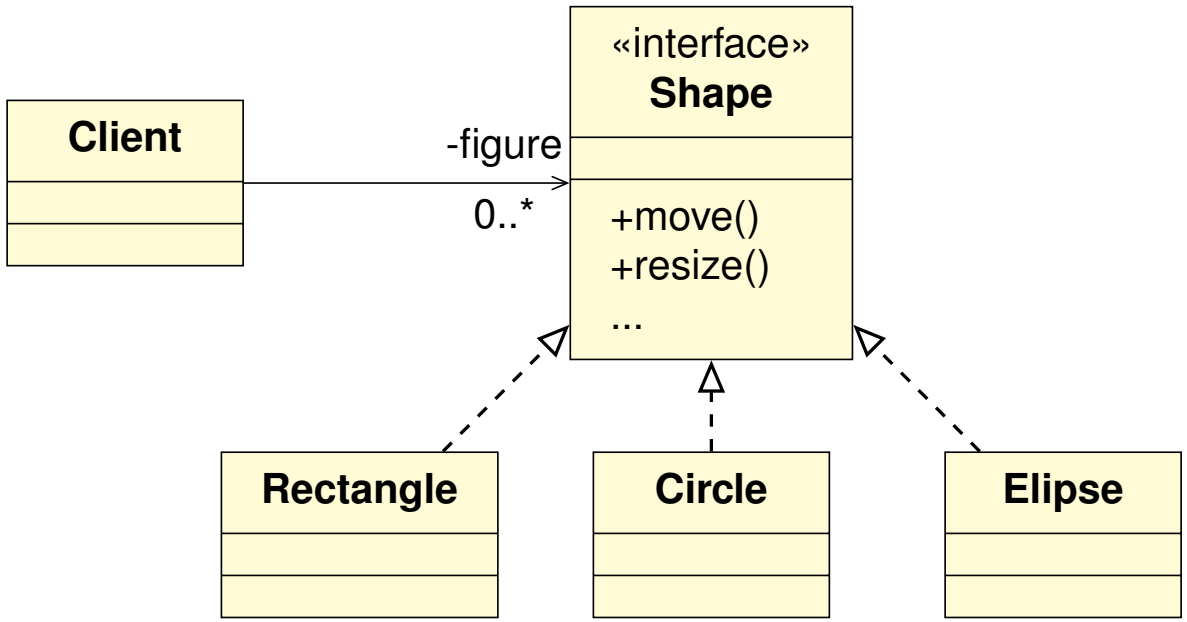
Le modèle



- Le modèle définit une collection de classes permettant de représenter des éléments graphiques.

Situation Initiale

Le modèle



- Le modèle définit une collection de classes permettant de représenter des éléments graphiques.
- Une interface commune permet de les utiliser de manière uniforme dans l'application

Situation Initiale

Utilisation uniforme

- Le client [1] fait une utilisation uniforme des éléments d'une figure,
- il opère dessus sans connaître leur nature :

```
1  public class Client {
2     static void main(String[] args) {
3        List<Shape> figure=Builder.load(args[0]);
4
5        Shape s=figure.item(5);
6        s.move(10, 15);
7        s.resize(1.2);
8
9        ...
10    }
11 }
```

1. Comme d'habitude, une partie du client.

Situation Initiale

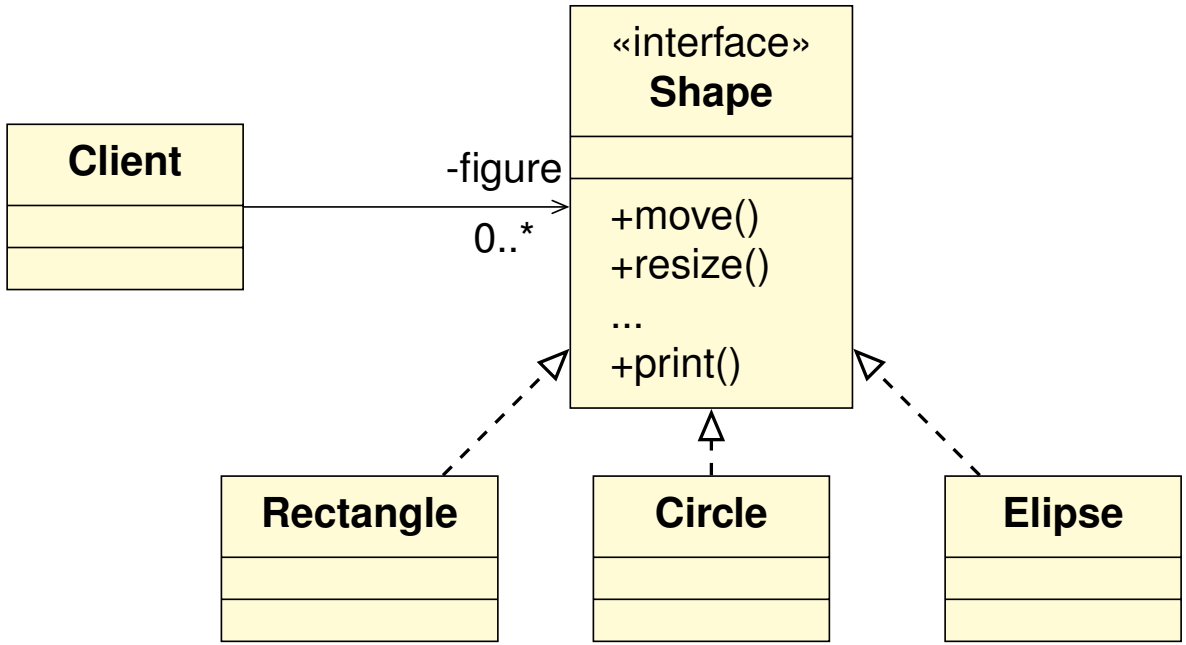
Nouvelle opération

- On souhaite pouvoir imprimer la liste des figures dans le terminal :

```
1  public class Client {
2     static void main(String[] args) {
3        List<Shape> figure=Builder.load(args[0]);
4
5        for (Shape s: figure)
6           s.print();
7     }
8  }
```

Situation Initiale

Nouvelle opération



- La nouvelle opération étant uniforme (devant être applicable à toute `Shape`), on doit la faire apparaître dans l'interface,
- et fournir une implémentation pour chaque classe.

Situation Initiale

Nouvelle opération

- Chaque classe doit comporter l'implémentation appropriée :

```
1  public class Rectangle {...
2     public void print() {
3        System.out.println("Rectangle("+x1+","+y1+"-"+x2+","+y2+")");
4     }
5     ...
6  }
7
8  public class Circle {...
9     public void print() {
10       System.out.println("Circle("+x+","+y+"-"+radius+")");
11    }
12    ...
13 }
14
15 public class Ellipse {...
16    public void print() {
17       System.out.println("Ellipse("+x+","+y+"-"+rx+","+ry+")");
18    }
19    ...
20 }
```

Situation Initiale

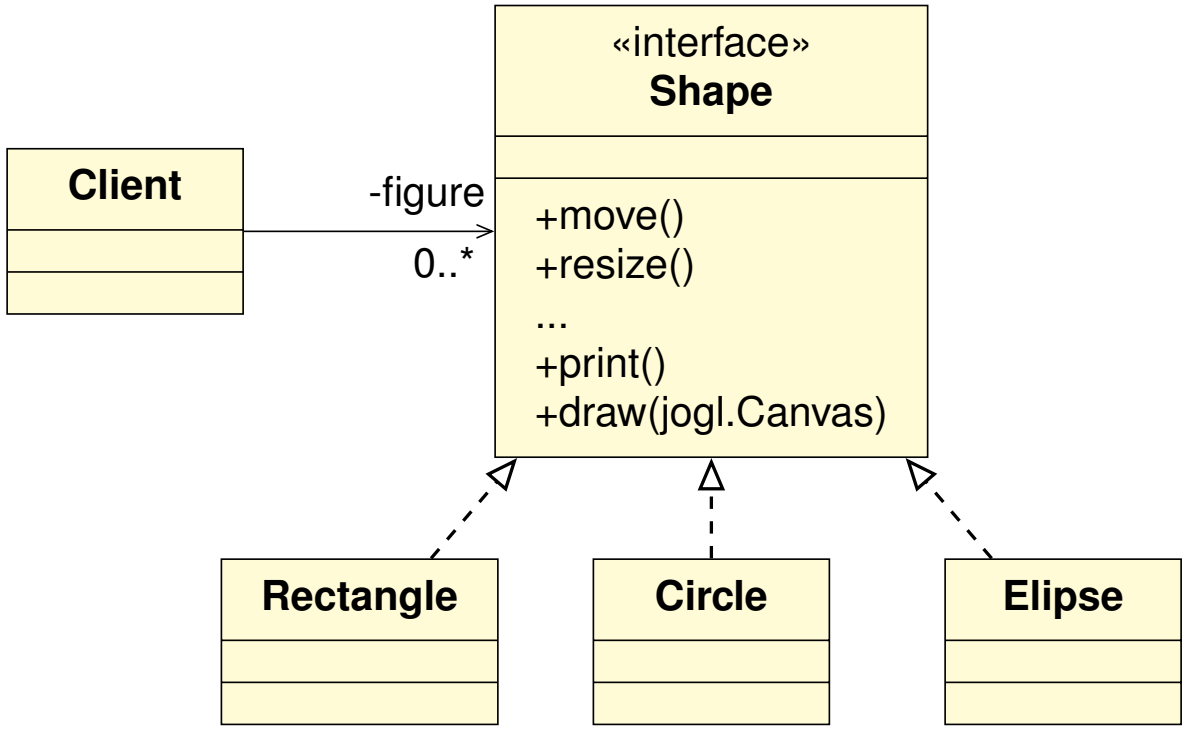
Seconde nouvelle opération

- On souhaite pouvoir procéder au rendu graphique de ces formes :

```
1  import jogl;
2
3  public class Client {
4      static void main(String[] args) {
5          List<Shape> figure=Builder.load(args[0]);
6          Canvas canvas=new Canvas();
7
8          for (Shape s: figure)
9          s.draw(canvas);
10     }
11 }
```

Situation Initiale

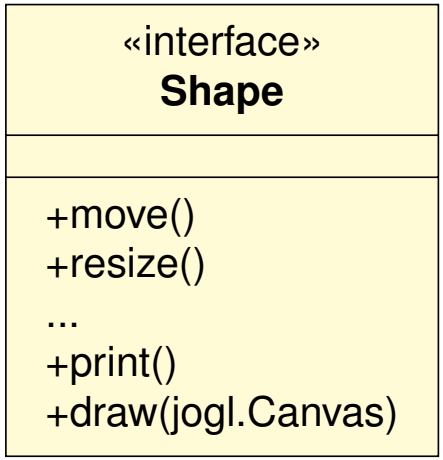
Seconde nouvelle opération



- Là aussi, il faut pouvoir appliquer l'opération à toute l'arborescence d'héritage.

Situation Initiale

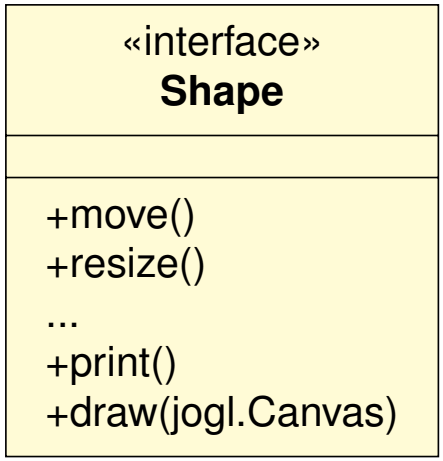
Bilan



- Cette situation est-elle acceptable ?

Situation Initiale

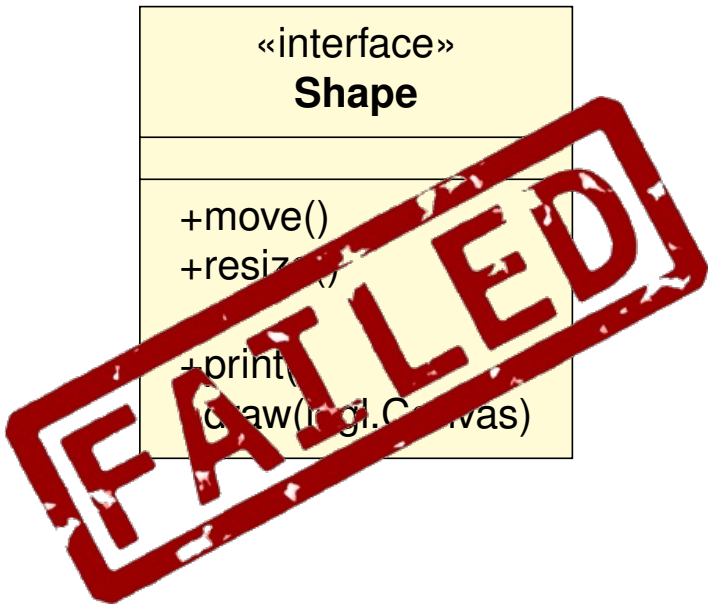
Bilan



- ISP et SRP enfreint !
- DIP : le modèle est couplé avec une bibliothèque graphique !
- plus un TDA,
- Toute nouvelle opération implique la modification de toutes les classes du modèle,

Situation Initiale

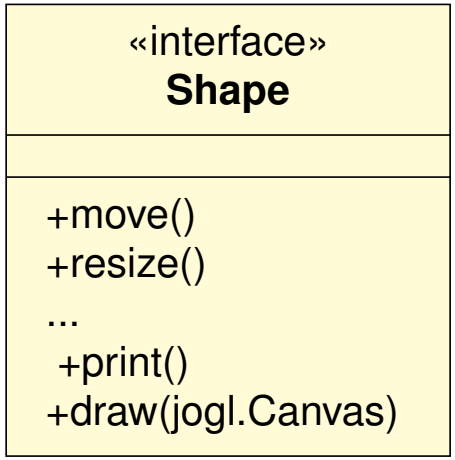
Bilan



- ISP et SRP enfreint !
- DIP : le modèle est couplé avec une bibliothèque graphique !
- plus un TDA,
- Toute nouvelle opération implique la modification de toutes les classes du modèle,

Situation Initiale

Solution

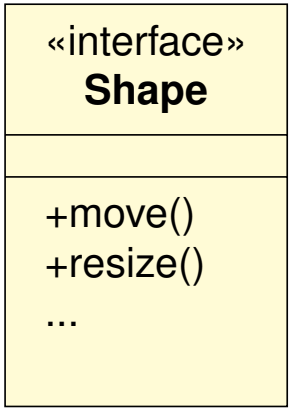


- Nous ne voulons pas de ces méthodes dans notre interface,
- Problème nous avons besoin du polymorphisme !
- Et si nous ajoutions une méthode pour toutes les opérations [2] ?
- C'est le principe du *visiteur* !

2. One method to rule them all !

Situation Initiale

Solution

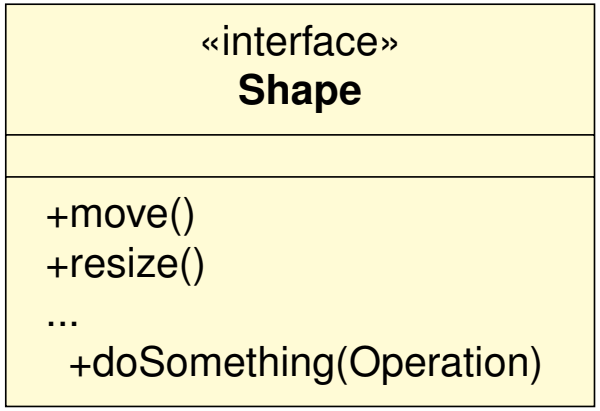


- Nous ne voulons pas de ces méthodes dans notre interface,
- Problème nous avons besoin du polymorphisme !
- Et si nous ajoutions une méthode pour toutes les opérations [2] ?
- C'est le principe du *visiteur* !

2. One method to rule them all !

Situation Initiale

Solution



- Nous ne voulons pas de ces méthodes dans notre interface,
- Problème nous avons besoin du polymorphisme !
- Et si nous ajoutions une méthode pour toutes les opérations [2] ?
- C'est le principe du *visiteur* !

2. One method to rule them all !

Le patron **Visitor**

Aussi connu comme

Intention

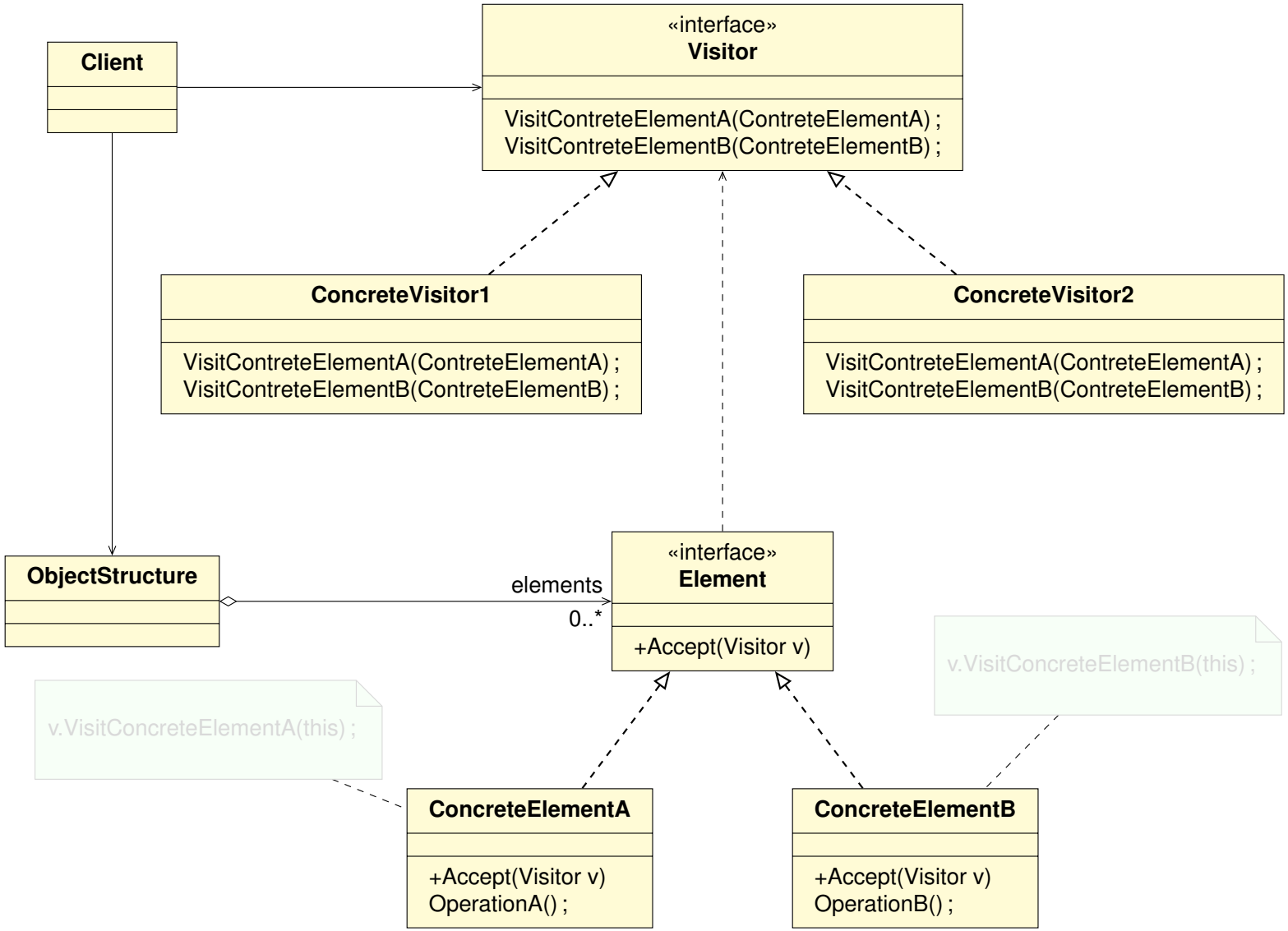
- Représenter les opérations à effectuer sur les éléments d'une structure d'objet.
- Ajouter des opérations sans modifier les classes de la structure.

Motivation

- Conserver la pureté des TDA.
- Éviter le couplage de code du modèle avec des bibliothèques
- Centraliser l'ajout de nouvelles opérations.

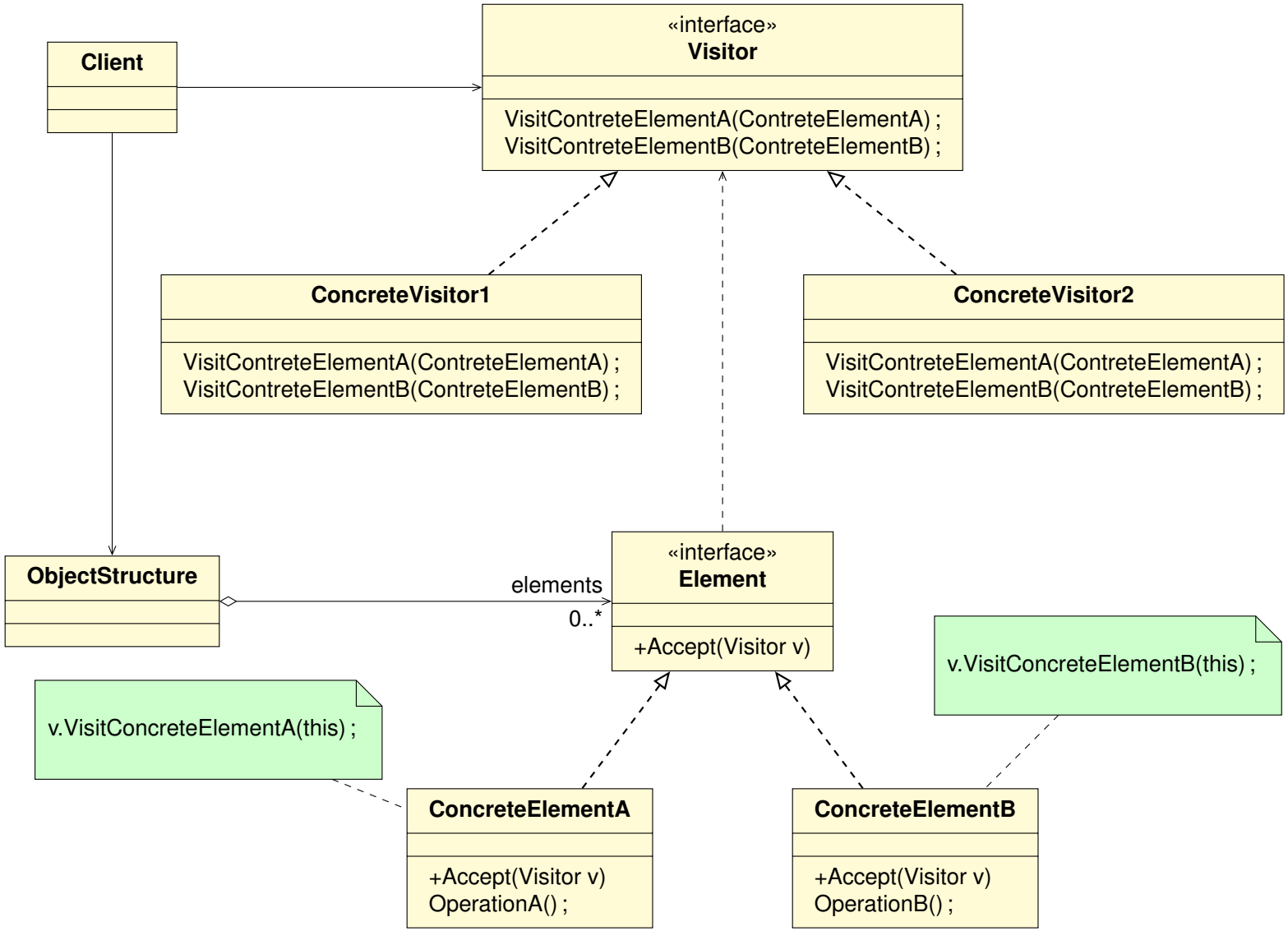
Le patron Visiteur

Schéma de principe



Le patron Visiteur

Schéma de principe

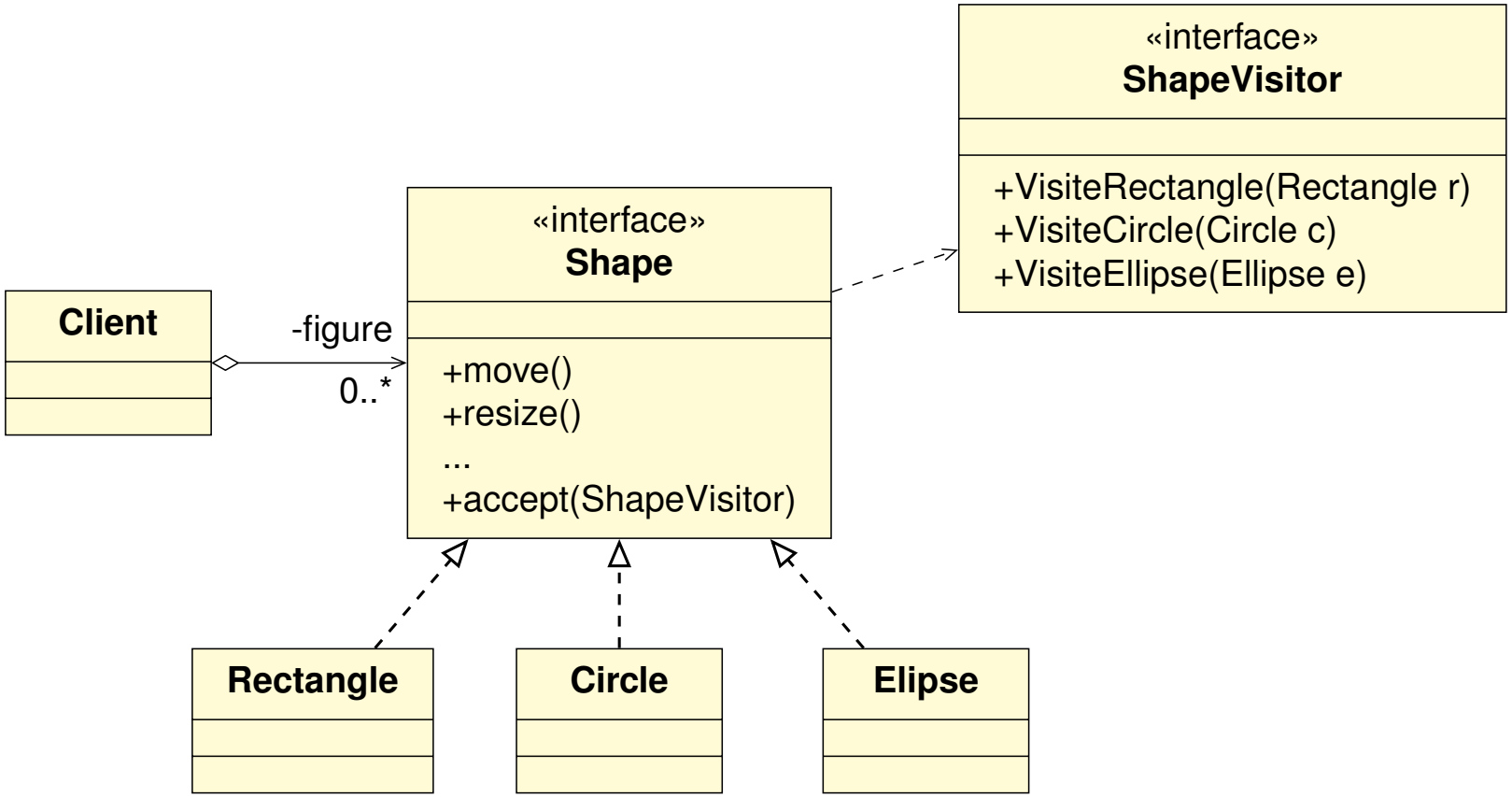


Participants du patron **Visitor**

- **Client** : Utilise la structure et applique sur celle-ci une opération à l'aide d'un visiteur concret
- **Visitor** : Définit la méthode de visite pour chaque classe du modèle.
- **ConcreteVisitor** : Fournit une implémentation d'une opération pour chaque élément du modèle.
- **Element** : Définit une opération d'acceptation d'un visiteur.
- **ConcreteElement** : Implémente l'opération d'acceptation conduisant au choix de l'implémentation appropriée de l'opération.
- **ObjectStructure** : Partie du client qui manipule la structure d'objets.

Retour à l'exemple

Intégration du visiteur



Retour à l'exemple

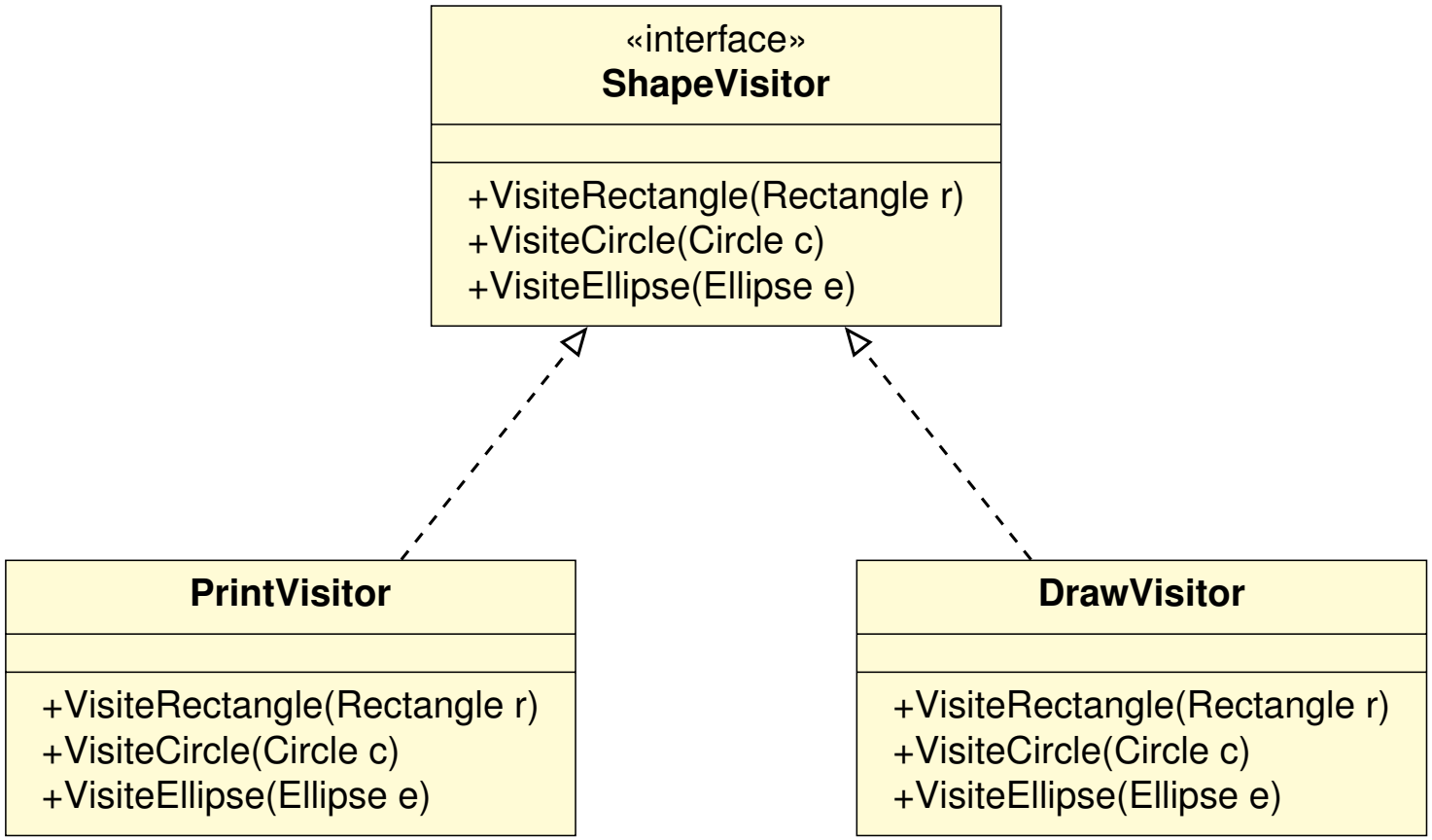
Intégration du visiteur

- Chaque classe connaît l'implémentation à appeler :

```
1  public class Rectangle {...
2     public void Accept(ShapeVisitor v) {
3     v.VisitRectangle(this);
4     }
5     ...
6  }
7
8  public class Circle {...
9     public void Accept(ShapeVisitor v) {
10       v.VisitCircle(this);
11    }
12    ...
13 }
14
15 public class Ellipse {...
16    public void Accept(ShapeVisitor v) {
17       v.VisitEllipse(this);
18    }
19    ...
20 }
```

Retour à l'exemple

Ajout d'opérations



Retour à l'exemple

Ajout d'opérations

- Un visiteur concret contient toutes les implémentations d'une même opération :

```
1  public class PrintVisitor {...
2     public void VisiteRectangle(Rectangle r) {
3        System.out.println("Rectangle("+r.x1+","+r.y1+"-"+r.x2+","+r.y2+")")
               ;
4     }
5     public void VisiteCircle(Circle c) {
6        System.out.println("Circle("+c.x+","+c.y+"-"+c.radius+")");
7     }
8     public void VisiteEllipse(Ellipse e) {
9        System.out.println("Ellipse("+e.x+","+e.y+"-"+e.rx+","+e.ry+")");
10    }
11    ...
12 }
```

- Idem pour `DrawVisitor` ou toute autre opération.

Retour à l'exemple

Utilisation

- On souhaite pouvoir imprimer la liste des figures dans le terminal :

```
1  public class Client {
2     static void main(String[] args) {
3        List<Shape> figure=Builder.load(args[0]);
4
5        PrintVisitor pv=new PrintVisitor();
6
7        for (Shape s: figure)
8           s.accept(pv);
9
10       DrawVisitor dv=new DrawVisitor();
11
12       for (Shape s: figure)
13          s.accept(dv);
14    }
15 }
```

Principes respectés

- OUPS ! À vous de jouer ! ;-)

Principes respectés

- OUPS ! À vous de jouer ! ;-)
- **S.R.P.** :
- **O.C.P.** :
- **L.S.P.** :
- **I.S.P.** :
- **D.I.P.** :

Conclusion

Ne pas oublier

- Question ?
- Terminer le TP *Composite* pour vendredi prochain.