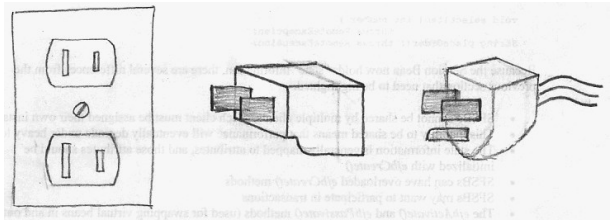
Patrons structurels

- On a des classes, il faut les organiser pour qu'elles fonctionnent ensemble

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Flyweight (195) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Patrons structurels

- Adapter : adapter une interface à une autre
- Decorator : ajouter une caractéristique à un objet
- Composite : traiter de manière identique un groupe et un élément
- Facade : clarifier/simplifier les interfaces d'un paquetage
- Flyweight : quand un objet coûte trop cher
- Bridge : pont à sens unique entre deux familles d'objets
- Proxy : l'objet est loin, le calcul est lourd, l'accès est réservé



Adapter : quand

- interface List1<T> {
  T get(int);
  int size();
  }
- interface Holder1<T> {
  /** null if unset */
  T get();
  }

- interface List2<T> {
  T get();
  void position(int);
  int size();
  }
- interface Holder2<T> {
  /** Exception if unset */
  T get();
  }

Adapter : comment

- On veut pouvoir utiliser un objet implémentant I1 avec une méthode qui veut un I2
- On écrit une classe qui implémente I2 et stocke un I1
- Les méthodes de I2 sont implémentées en utilisant les méthodes de I1

Adapter : exemple

```
interface List1<T> {
   T get(int position);
   int size();
}
```

```
interface List2<T> {
   T get();
   void position(int position);
   int size();
}
```

```
public class List1ToList2<T> implements List2<T> {
   private final List1<T> adapted;
   private int position; /* =0 */
   public List1ToList2(List1<T> adapted) { this.adapted = adapted; }
   public T get() { return adapted.get(position++); }
   public void position(int position) { this.position = position; }
   public int size() { return adapted.size(); }
}
```

```
public class List2ToList1<T> implements List1<T> {
   private final List2<T> adapted;
   public List2ToList1(List2<T> adapted) { this.adapted = adapted; }
   public T get(int position) {
     adapted.position(position); return adapted.get();
   }
   public int size() { return adapted.size(); }
}
```

Adapter : exemple

```
interface Holder1<T> {
   /** null if unset */
   T get();
}
```

```
interface Holder2<T> {
   /** NoSuchElementException if unset */
   T get();
}
```

```
public class Holder1ToHolder2<T> implements Holder2<T> {
   private final Holder1<T> adapted;
   public Holder1ToHolder2(Holder1<T> adapted) { this.adapted = adapted; }
   public T get() {
     T value = adapter.get();
     if (value == null)
        throw new NoSuchElementException();
     return value;
   }
}
```

```
public class Holder2ToHolder1<T> implements Holder1<T> {
   private final Holder2<T> adapted;
   public Holder2ToHolder1(Holder2<T> adapted) { this.adapted = adapted; }
   public T get() {
     try {
       return adapter.get();
     }
     catch (NoSuchElementException e) { return null; }
   }
}
```

Adapter : exemple

```
class Line {
        public void draw(int x1, int y1, int x2, int y2) {
          System.out.println("line from (" + x1 + ',' + y1 + ") to (" + x2
+ ',' + y2 + ')');
        }
}

class Rectangle {
        public void draw(int x, int y, int w, int h) {
          System.out.println("rectangle at (" + x + ',' + y + ") with
width " + w + " and height " + h);
        }
}
```

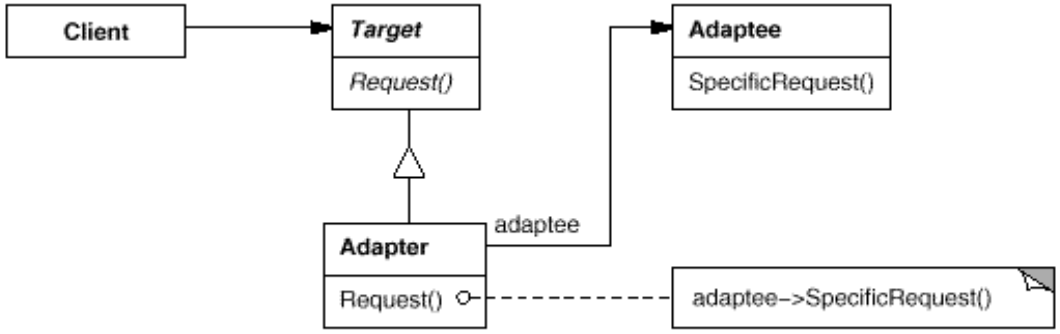
```
public class NoAdapter {

  public static void main(String[] args) {
        Object[] shapes = { new Line(), new Rectangle() };

        //.....
        //.....

        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i) {
          if (shapes[i] instanceof Line)
                ((Line) shapes[i]).draw(x1, y1, x2, y2);
          else if (shapes[i] instanceof Rectangle)
                ((Rectangle) shapes[i]).draw(
                        Math.min(x1, x2), Math.min(y1, y2),
                        Math.abs(x2 - x1), Math.abs(y2 - y1));
        }
  }
}
```

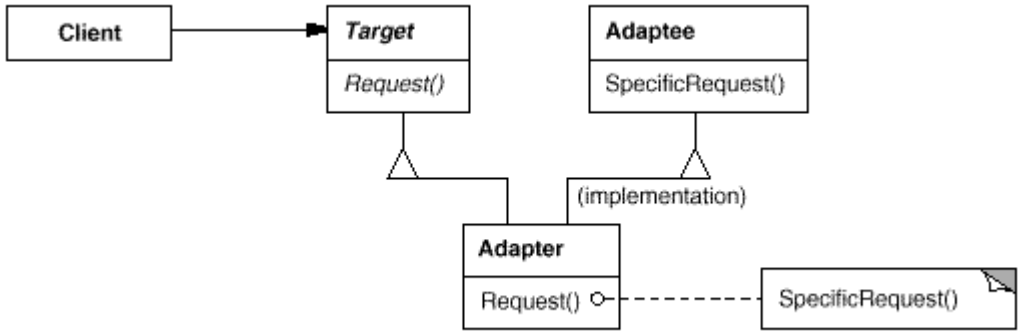
Adapter : exemple

```
interface Shape {
       void draw(int x1, int y1, int x2, int y2);
}
class NewLine implements Shape {
       private Line adaptee = new Line();
       @Override
       public void draw(int x1, int y1, int x2, int y2) {
              adaptee.draw(x1, y1, x2, y2);
       }
}
class NewRectangle implements Shape {
       private Rectangle adaptee = new Rectangle();
       @Override
       public void draw(int x1, int y1, int x2, int y2) {
         adaptee.draw(Math.min(x1, x2),
           Math.min(y1, y2), Math.abs(x2 – x1), Math.abs(y2 - y1));
       }
}
```

```
public class WithAdapter {

  public static void main(String[] args) {
    Shape[] shapes = { new NewLine(), new NewRectangle() };

    //.....

    // A begin and end point from a graphical editor
    int x1 = 10, y1 = 20;
    int x2 = 30, y2 = 60;
    for (int i = 0; i < shapes.length; ++i)
       shapes[i].draw(x1, y1, x2, y2);
...
```

Adapter's


Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()


Client
Target
Request()
Adaptee
SpecificRequest()
(implementation)
Adapter
Request()
SpecificRequest()

Decorator : quand

- Ajouter un bord à un bouton, une feuille de calcul, une combobox, avec le même code
- Crypter un fichier, un flot de données sur le réseau, une zone de la mémoire, avec le même code

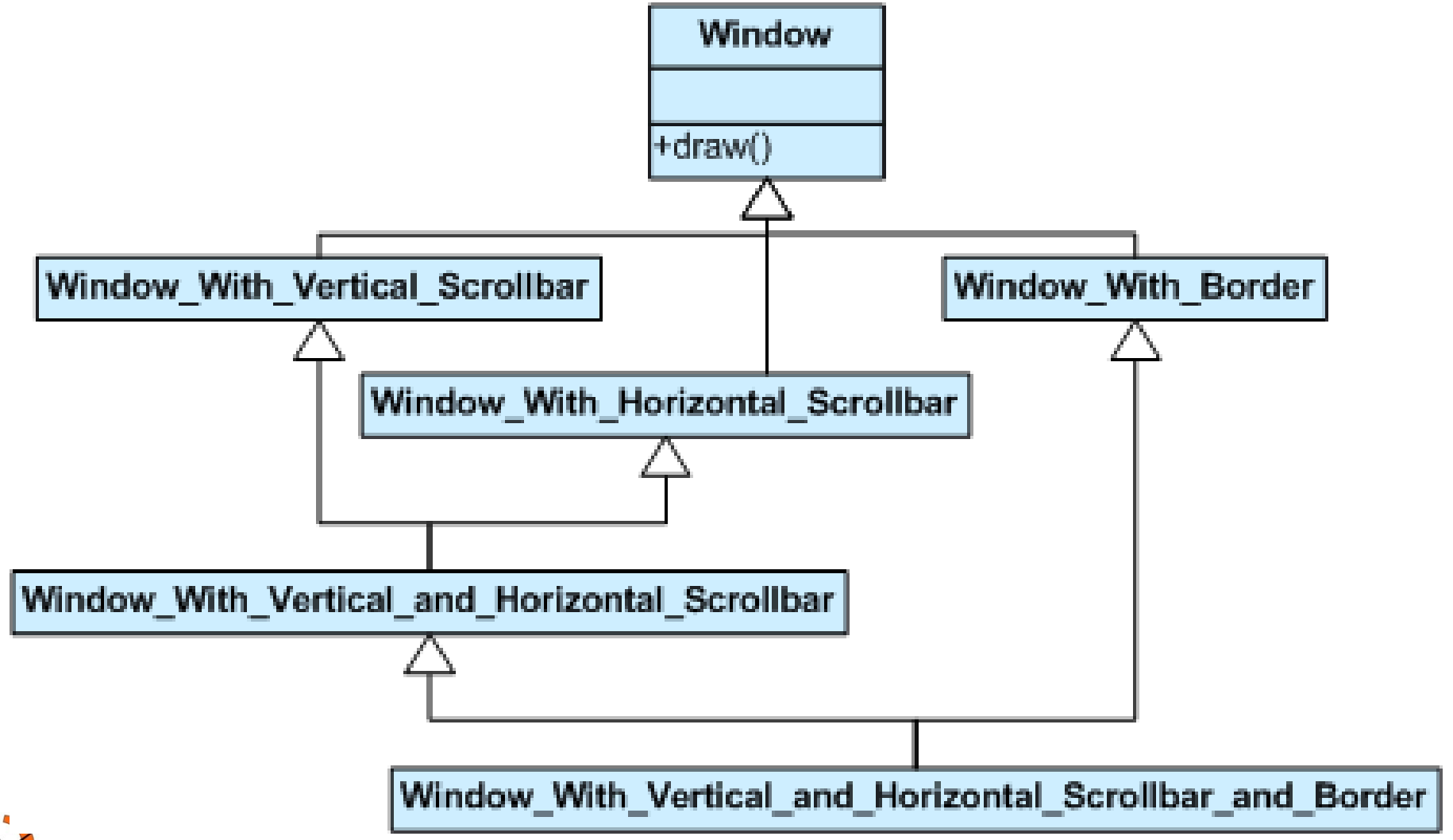
- Ajout de responsabilité, mais pas de nouveau comportement.

  => interface non modifiée !

Decorator : comment

- La fonctionnalité supplémentaire s'intercale avec la fonctionnalité initiale de l'objet :
  - pour le bord, on dessine d'abord le contrôle, plus petit, puis on dessine le bord autour
  - pour le cryptage / décryptage :
    - On lit les données avec l'objet initial, on les décrypte
    - On crypte puis on écrit les données avec l'objet initial
- implémentation :
  - Le décorateur est une classe qui contient l'objet initial
  - Délégation des fonctions de base à l'objet décoré

Decorator: limiter le sub-classing

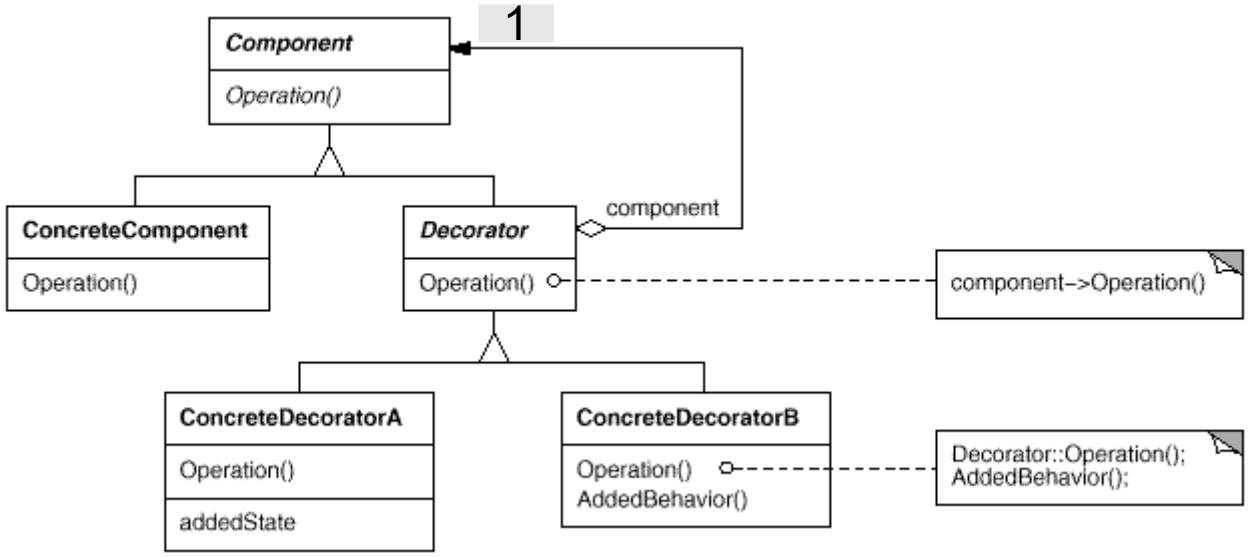


Decorator : exemple

```
public class CryptingOutputStream extends OutputStream {
   private final OutputStream decorated;
   public CryptingOutputStream(OutputStream decorated) {
     this.decorated = decorated;
   }
   public void write(byte[] buffer) {
     crypt(buffer);
     decorated.write(buffer);
   }
   …
}
```

```
public class BorderedComponent extends Component {
   private final Component decorated;
   public BorderedComponent(Component decorated) {
     this.decorated = decorated;
   }
   public void draw(Graphics g,Bounds l) {
     decorated.draw(g,new Bounds(l.x+1,l.y+1,l.w-2,l.h-2));
     g.drawRect(l.x,l.y,l.w,l.h);
   }
   …
}
```

Decorator


1
Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

Decorator : impacts

- Flexibilité, comparé à l'héritage, static
- Développement progressif, indépendant

- Decorator # objet décoré : l'identité de l'objet ne peut être utilisée :
  - ni this
  - ni type
- Beaucoup de petits objets. Peut être compliqué à appréhender et débugger

Decorator – streams Java

```
public interface InputStream {
	int read();
	int read(byte[] b);
	…
}
```

```
public class FilterInputStream implements InputStream {
…
	protected FilterInputStream(InputStream is) {
		this.inputStream = is ;
	}

	@Override
	public int read() {
		return this.inputStream.read();
	}
...
```

Decorator – to uppercase

```
public class UpperCaseInputStream extends FilterInputStream {

	protected UpperCaseInputStream(InputStream arg0) {
		super(arg0);
	}

	@Override
	public int read() throws IOException {

		return Character.toUpperCase(super.read());
	}
```

```
UpperCaseInputStream ucis = new UpperCaseInputStream(System.in);
StringBuilder sb = new StringBuilder();
int c;
while ((c = ucis.read()) > 0 && Character.isLetterOrDigit(c)) {
	sb.append(Character.toChars(c));
}
System.out.println(sb.toString());
```

Decorator – to uppercase

```
public class UpperCaseInputStream extends FilterInputStream {

	protected UpperCaseInputStream(InputStream arg0) {
		super(arg0);
	}

	@Override
	public int read() throws IOException {

		return Character.toUpperCase(super.read());
	}
	@Override
	public int read(byte[] arg0, int arg1, int arg2)
			throws IOException {
		??
	}

	@Override
	public int read(byte[] arg0) throws IOException {
		??
	}
```

Abstract ?

Decorator ++

	PROS	CONS
Héritage Ex JDK : JButton	- simple pour les cas simples - type des objets conservé	- complexité hiérarchie classes - typage static
Wrapper (GoF) Ex JDK : BufferedInputStream	- implémentation simple - implémentations indépendantes - utilisation récursive	- type non conservé - this décoré # this - beaucoup de delegate de base
Externe Ex JDK : JComponent	- le type des objets ne change jamais - implémentations indépendantes	- la décoration doit être « prévue » par l'objet décoré - ne peut pas « changer » mais juste « ajouter »

Composite : quand

- Un dessin est un ensemble de traits
  - les deux classes sont « dessinables »
- Un somme contient un certain nombre de termes
  - la somme est un terme
- On veut traiter les nœuds et les feuilles de l'arbre avec une interface commune

Composite : comment

- La classe réalisant la composition implémente l'interface I et contient une liste de I

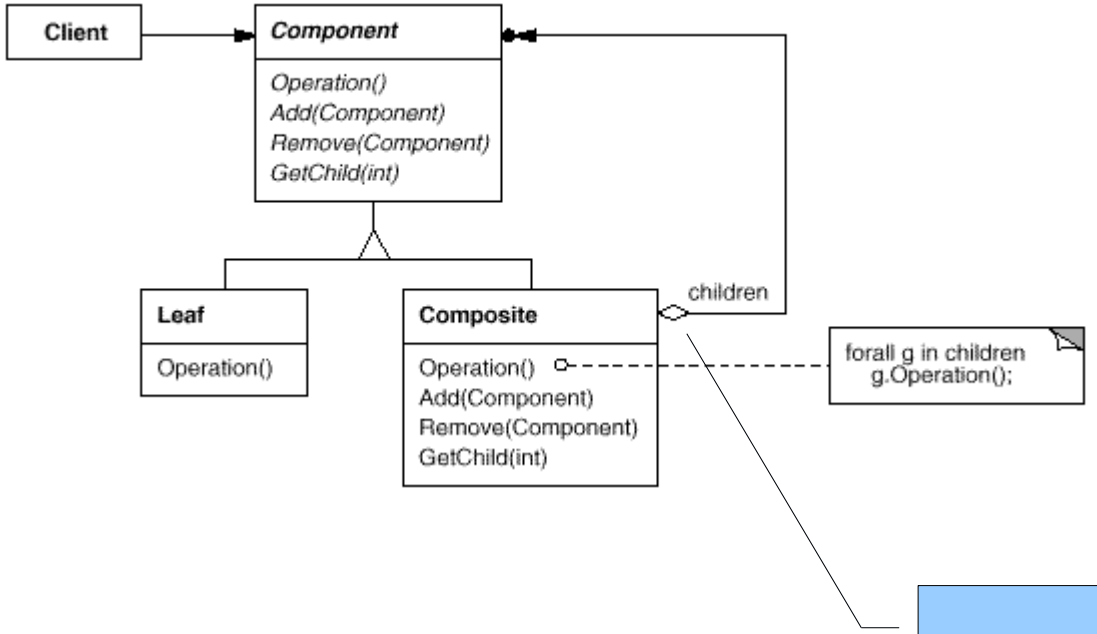
```
public interface Drawable {
   void draw(Graphics g);
}
```

```
public class CompositeDrawable implements Drawable {
  private final ArrayList<Drawable> drawables;

  public void draw(Graphics g) {
    for(Drawable d : drawables)
       d.draw(g);
  }
}
```

Attention à l'ordre, il peut être important !

Composite

- Version transparente



- Version sûre ?

Composite : check-list

- Dans les objets du domaines: a-t-on des collections d'objets dont certains peuvent être aussi des collections
- Définir une classe de base commune, représentant le socle commun des composants et composés, qui suffit en général au client.
- Définir les collections et les éléments comme classes dérivées de cette classe commune
- Définir une relation « has-a » 1..n des classes composites vers la classe de base
  - Typiquement avec une abstract class pour la classe composite

Ls ...

```
class File {
  public File( String name ) {     m_name = name;   }
  public void ls() {       System.out.println( Composite.g_indent + m_name );   }
  private String m_name;
}
class Directory {
  public Directory( String name ) {     m_name = name;   }
  public void add( Object obj ) {       m_files.add( obj );   }
  public void ls() {
    System.out.println( Composite.g_indent + m_name );
    Composite.g_indent.append( "    " );
    for (int i=0; i < m_files.size(); ++i) {
      Object obj = m_files.get(i);
      // ***** Recover the type of this object ******
       if (obj.getClass().getName().equals( "Directory" ))
         ((Directory) obj).ls();
       else
         ((File) obj).ls();
    }
    Composite.g_indent.setLength( CompositeDemo.g_indent.length() - 3 );
  }
  private String    m_name;
  private ArrayList m_files = new ArrayList();
}
public class CompositeDemo {
  public static StringBuffer g_indent = new StringBuffer();
  public static void main( String[] args ) {
    Directory one = new Directory("dir111"),
              two = new Directory("dir222"),
              thr = new Directory("dir333");
    File  a = new File("a"), b = new File("b"),
          c = new File("c"), d = new File("d"),
          e = new File("e");
    one.add( a );   one.add( two );   one.add( b );
    two.add( c );   two.add( d );     two.add( thr );
    thr.add( e );
    one.ls();
  }
}
```

```
// dir111
//    a
//    dir222
//       c
//       d
//       dir333
//          e
//    b
```

Ls ...

```
interface AbstractFile {
   public void ls();
}
// * File implements the "lowest common denominator"
class File implements AbstractFile {
   public File( String name ) {      m_name = name;   }
   public void ls() {System.out.println(CompositeDemo.g_indent+m_name);}
   private String m_name;
}
// Directory implements the "lowest common denominator"
class Directory implements AbstractFile {
   public Directory( String name ) {m_name = name; }
   public void add( AbstractFile f ) {m_files.add( f );  }
   public void ls() {
      System.out.println( CompositeDemo.g_indent + m_name );
      CompositeDemo.g_indent.append( "   " );
       For (AbstractFile f : m_files) {
          f.ls();
       }
      CompositeDemo.g_indent.setLength(CompositeDemo.g_indent.length() - 3 );
   }
   private String    m_name;
   private ArrayList<AbstractFile> m_files = new ArrayList<AbstractFile>();
}
public class CompositeDemo {
   public static StringBuffer g_indent = new StringBuffer();
   public static void main( String[] args ) {
      Directory one = new Directory("dir111"),
                two = new Directory("dir222"),
                thr = new Directory("dir333");
      File  a = new File("a"), b = new File("b"), c = new File("c"), d = new File("d"),
            e = new File("e");
      one.add( a );   one.add( two );   one.add( b );  two.add( c );   two.add( d );     two.add( thr );
      thr.add( e );
      one.ls();
   }
}
```

```
// dir111
//    a
//    dir222
//       c
//       d
//       dir333
//          e
//    b
```

exercice

```
public class TestCompositeTree {

	public static void main(String[] args) {
		Menu m = new CompositeMenu("main");
		m.add(new MenuItem("label1"))
		 .add(new MenuItem("label2"))
		 .add(new CompositeMenu("sub1")
				.add(new MenuItem("sub1--1"))
				.add(new MenuItem("sub1--2")))
		 .add(new MenuItem("label3"));

		m.display(0);
	}

}
```

```
main
	label1
	label2
	sub1
		sub1--1
		sub1--2
	label3
```

exercice

```
interface Menu
{
	void display(int level);
	Menu add(Menu menu);
}
```

Version transparente

```
class MenuItem implements Menu {
	private String label;
	public MenuItem(String label) {
		super();
		this.label = label;
	}
	@Override
	public void display(int level) {
		for (int i=0;i<level;++i) System.out.print('\t');
		System.out.println(label);

	}
	@Override
	public Menu add(Menu menu) {
		throw new UnsupportedOperationException();
	}
}
```

Indentation (pour test)

exercice

```
class CompositeMenu implements Menu {
	private List<Menu> sub = new ArrayList<Menu>();
	private String label;

	public CompositeMenu(String label) {
		super();
		this.label = label;
	}
	@Override
	public Menu add(Menu menu) {
		sub.add(menu);
		return this;
	}
	@Override
	public void display(int level) {
		for (int i=0;i<level;++i) System.out.print('\t');
		System.out.println(label);
		for (Menu m : sub) {
			m.display(level+1);
		}
	}
}
```

Il peut y avoir plusieurs classes de Menu composite !
On ne peut que retourner un Menu

Proxy : quand

- On a des données :
  - dont l'accès doit être contrôlé
  - qui sont trop lourdes pour les avoir en entier en mémoire
  - qui sont longues à charger/recharger
  - qui sont difficiles à calculer
  - qui sont distantes
  - ...

Proxy : comment

- On place un objet proxy entre l'objet réel et le client, dont l'interface est la même, qui :
  - cache des informations en mémoire (calcul long, temps de chargement long)
  - retarde le chargement des données au moment où elles sont vraiment nécessaire (données lourdes)
  - Transporte une requête via un canal de communication (applications distribuées)
  - Contrôle l'accès (authentification)
  - Effectue des pré/post-traitements (log, ...)

Proxy : exemples

- Cache:
  - Proxy cache HTTP (demander au CRI !)
- Chargement lazy
  - Chargement d'image : le proxy ne lit que l'en-tête, pour avoir:
    - taille, colorspace…
  - il ne charge l'image que si on souhaite la dessiner réellement
- Objets distants:
  - Proxy RPC, RMI, Corba, ...
- Contrôle d'accès
  - Proxy HTTP avec authentification
- pré/post traitement :
  - log, traçabilité
  - analyse performance (profiling)
  - vérification (valgrind: proxy de malloc/free !)

Proxy : Virtual Proxy

- Virtual Proxy
  - Je ne crée que si c'est vraiment nécessaire

```
Interface MyService {
        public void action1();
        ...
}
```

```
class MyServiceImpl implem    s MyService {
        public void action1() {    }
        ...
}
```

```
class Test {
        public static void main(...) {
                MyService service;
                If (…) service=new MyServiceProxy()
                Else service=new MyServiceImpl();
        }
}
```

Dépendance : peu acceptable

```
class MyServiceProxy implements MyService {
        public void action1() {
                check();
                realObject.action1();
        }
        private void check() {
                if (realObject==null) realObject=new MyServiceImpl();
        }
        private MyServiceImpl realObject = null;
}
```

Dépendance : peut être pas grave

Proxy : Contrôle d'accès

- Ici, c'est protégé !

```
Interface MyService {
        public String connect(String user,
                              String pwd);
        public void action1();
        ...
}
```

```
class MyServiceImpl implements MyService {
        public String connect(...) { return null;}
        public void action1() { ...}
        ...
}
```

```
class MyServiceProxy implements MyService {
        public void action1() {
                check();
                realObject.action1();
        }
        private void check() {
                if (sessionId==null) throw new …
        }
        MyServiceProxy(MyService o)
                { realObject=o;}
        private MyService realObject;
        private String sessionId;
}
```

```
class Test {
        public static void main(...) {
            MyService service =
                ServiceFactory.create(...);
            service.connect(...);
            service.action1();
        }
}
```

Proxy : Remote

- Il y a du chemin à faire !

```
Interface MyService {
        public void action1();
        ...
}
```

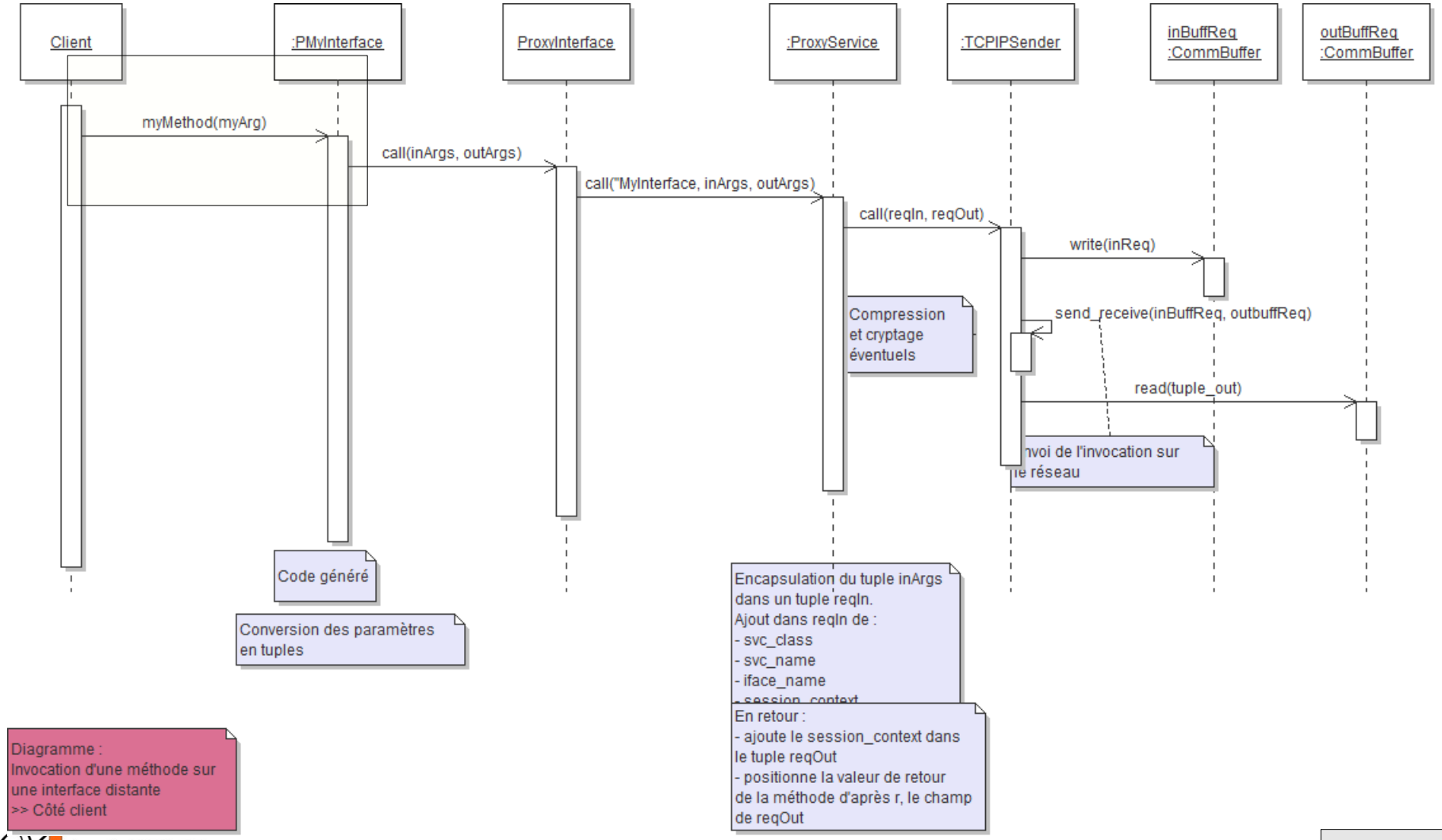
```
class MyServiceImpl implements MyService {
        public void action1() { ...}
        ...
}
```

```
class MyServiceProxy implements MyService {
        public void action1() {
                … build request ...
                connection.sendRequest(...);
        }
        MyServiceProxy(String target)
                { …  open connection ...;}
        Private … connection;
}
```

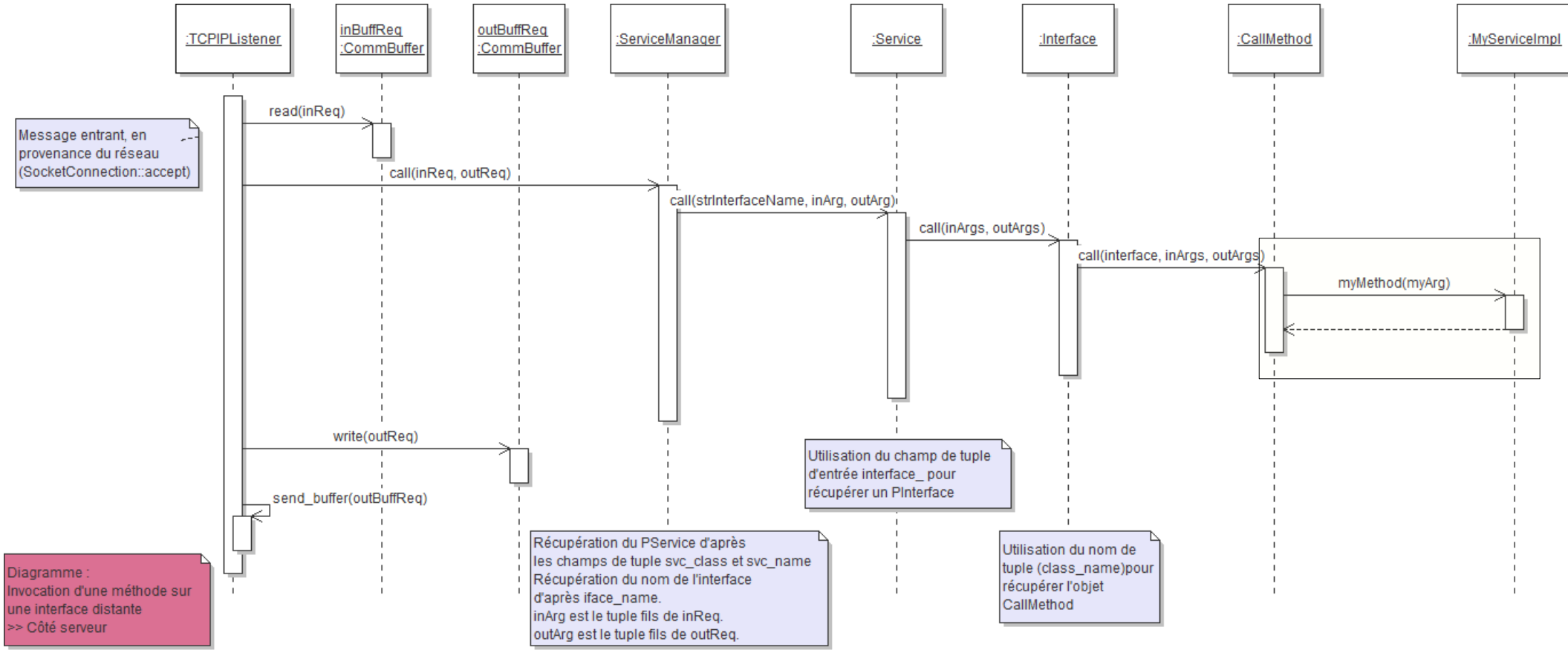
```
class Test {
        public static void main(...) {
                MyService service =
                        ServiceFactory.create(...);
                service.action1();
        }
}
```

Proxy : Remote (vue client)

- Il peut y avoir du chemin !



Proxy : Remote (vue server)



Smart Proxy

- Il y a des choses à faire, en plus :

```
Interface MyService {
        public void action1();
        ...
}
```

```
class MyServiceImpl implements MyService {
        public void action1() { ...}
        ...
}
```

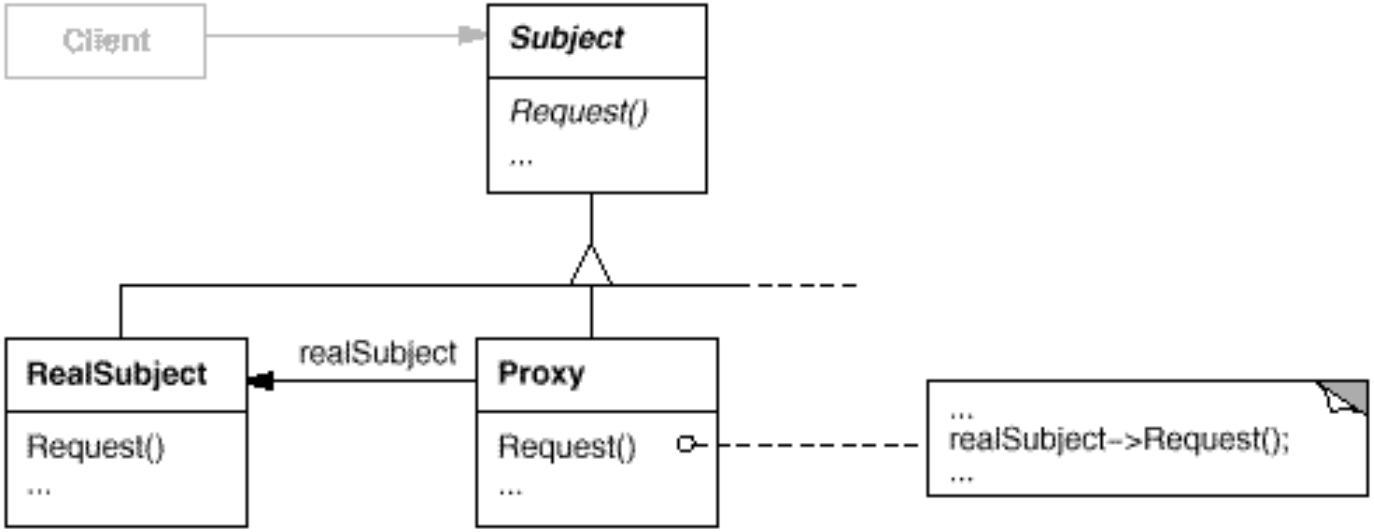
```
class MyServiceProxy implements MyService {
        public void action1() {
                log.finest(« … action1 ... »);
                realObject.action1();
        }
        MyServiceProxy(MyService o, Logger log)
                { realObject=o;}
        private MyService realObject;
}
```

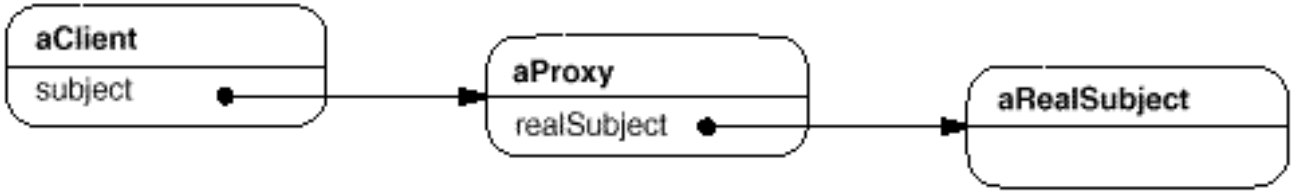
```
class Test {
        public static void main(...) {
            MyService service =
                ServiceFactory.create(...);
            service.action1();
        }
}
```

Proxy : check-list

- Identifier ce qui doit être implémenté avec un proxy
- Définir une interface commune au Proxy et à la classe réelle
- Une Factory permet en général d'encapsuler la décision de créer un proxy ou un objet réel
- Proxy implémente la même interface, en général avec une référence vers la classe réelle
- objet réel reçu à l'initialisation ou créé à la première utilisation
- Traitement délégué à l'objet réel, pré/post-traitement possible

Proxy


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...


aClient
subject
aProxy
realSubject
aRealSubject

Flyweight : quand

- La programmation objet, c'est très pratique, mais parfois coûteux en mémoire.
  - Un Integer prend plus de place qu'un int !
- Quand le nombre d'objets est important,
  - il faut donc économiser au maximum le nombre d'objets instanciés

Flyweight : comment

- Il faut partager les « caractéristiques communes » des objets en une seule et même instance
- Un objet est donc découpé en deux :
  - ses données « intrinsèques » qui seront partagées (l'objet flyweight)
  - ses données « extrinsèques » qui seront stockées dans une structure de donnée plus efficace, « à l'extérieur »
- Les méthodes de l'interface du flyweight prennent en paramètres les données extérieures, quand nécessaire

Flyweight : comment

- Il est en nécessaire d'avoir une factory pour créer/réutiliser les objets flyweight
- D'autres objets non flyweight peuvent avoir la même interface (pour utiliser le polymorphisme)
  - En général, ils n'utilisent pas les données externes, mais seulement leur état interne (accès plus rapide)

Flyweight : exemple

- Supposons que l'on veuille stocker un texte formaté en objet
- Un texte est un ensemble de lignes qui sont un ensemble de caractères qui ont chacun un certain nombre de caractéristiques (font/taille/ style/...)
- Les données intrinsèques de l'objet caractère : le caractère
- Les données extérieures : la font, la taille, le style
- Les méthodes de l'objet flyweight doivent donc prendre l'objet responsable du stockage des données extérieures en paramètre.

Flyweight : exemple

- La méthode de l'interface initiale pour dessiner est :
  draw(Graphics)
- Elle devient
  draw(Graphics,ExternalData)
- Une ligne respecte la même interface
- Une ligne peut :
  - Ne pas utiliser l'argument « externalData » (stocké dans la ligne)
  - L'utiliser partiellement et le calculer partiellement

```
interface Flyweight {
      public void draw(Graphics g, ExternalContext ec);
}
```

Flyweight : exemple

```
public class FlyweightChar implements Flyweight {
  private final char c;
  public FlyweightChar(char c) { this.c=c }
  public void draw(Graphics g,ExternalContext c) {
    …
  }
}
```

```
public class ExternalContext {
  private int position;
  public void setPosition(int position);
  public int getPosition();
  public Font getFont() { … } // computed from position
  public Style getStyle { … }
  public Size getSize { … }
}
```

Un composite !

```
public class Row implements Flyweight {
  private final List<FlyweightChar> characters;
  …
  public void draw(Graphics g,ExternalContext ec) {
    for(int i=0;i<characters.size();i++) {
      ec.setPosition(i);
      characters.get(i).draw(g, ec);
    }
  }
}
```

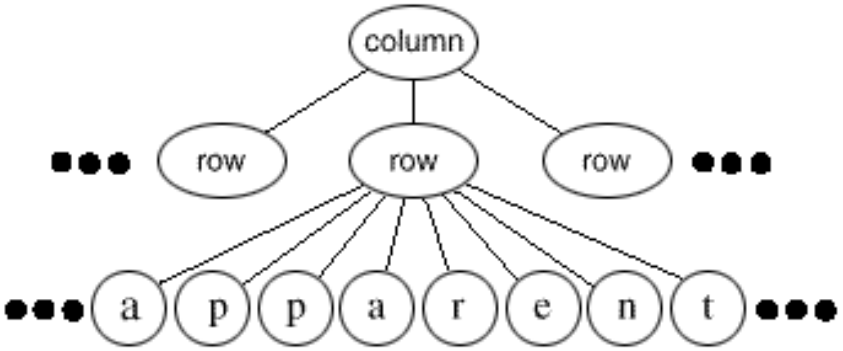
2 niveaux de calculs des données extrinsèques

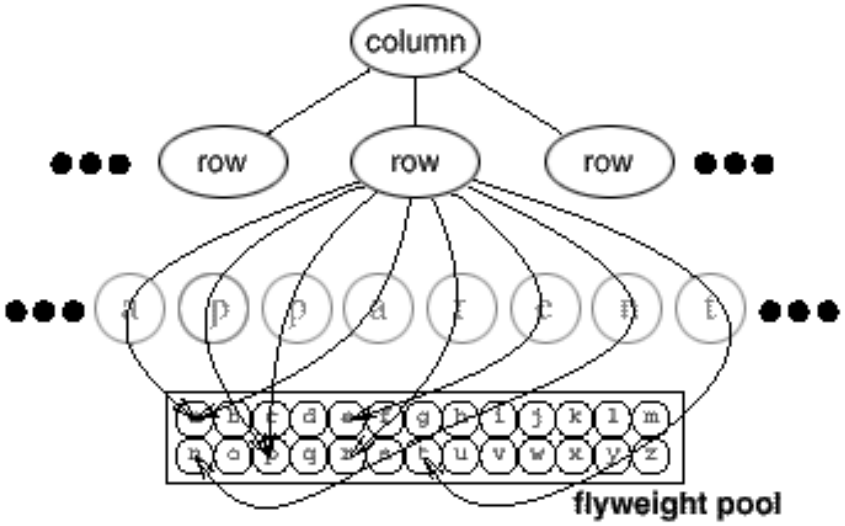
Rq: ici, même fonte sur toute la ligne

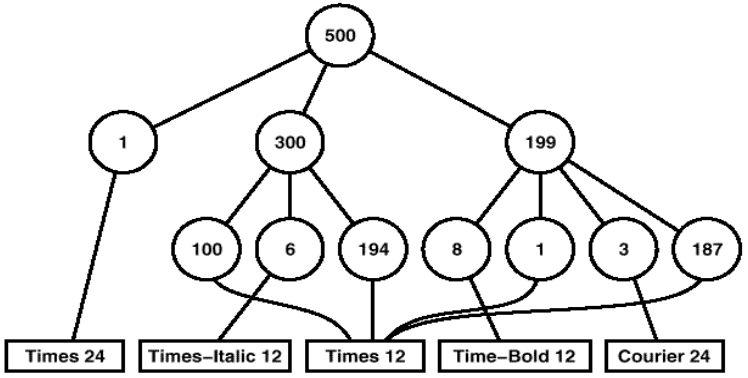
Flyweight : exemple

```
class FlyweightFactory {
      private final Map<Character, FlyweightChar> flyweightPool =
            new HashMap<Character, FlyweightChar>();

      public FlyweightChar instance(char c) {
            FlyweightChar fly = flyweightPool.get(c);
            if (fly == null) {
                  fly = new FlyweightChar(c);
                  flyweightPool.put(c, fly);
            }
            return fly;
      }
}
```

Flyweight : exemple

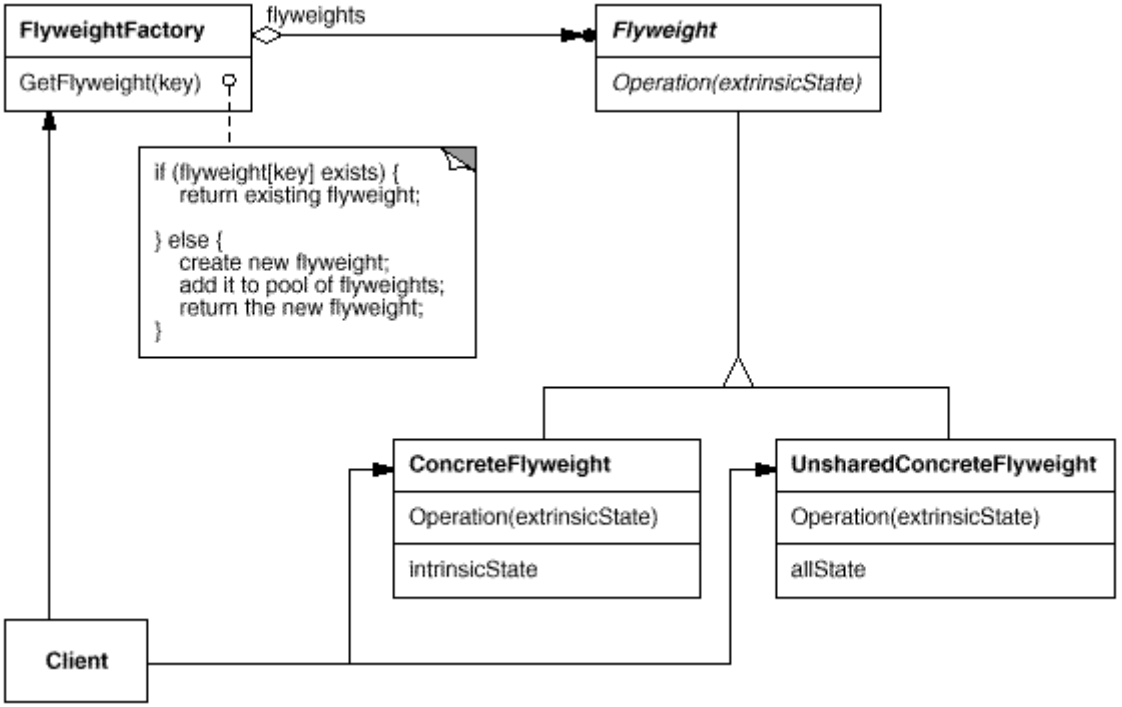






Gestion de ExternContext...

Flyweight


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Flyweight : check-list

- Il y a un problème ?
  - Le client pourra absorber la responsabilité supplémentaire ? (on lui complique la vie !)
- Diviser l'état entre partageable (intrinsèque) et , non-partageable (extrinsèque)
- La partie non-partageable est supprimée de la classe et ajoutée en argument de certaines méthodes
- Créer une Factory qui réutilise les instances des fly-weights
  - Le client doit utiliser cette Factory
- Le client (ou un tiers) doit *stocker ou calculer* les données non-partageables et les fournir à certaines méthodes

Bridge

- GoF: "*Decouple an abstraction from its implementation so that the two can vary independently*"
- Élément clé : dimension**S** de variabilité
  - Si une seule dimension, simple polymorphisme basé sur héritage ou interface
  - Par exemple, différentes « formes » (Shape)
- Si plusieurs dimension**S** ?
  - Abstraction « métier » / Plate-forme
  - domaine/infrastructure
  - front-end/back-end

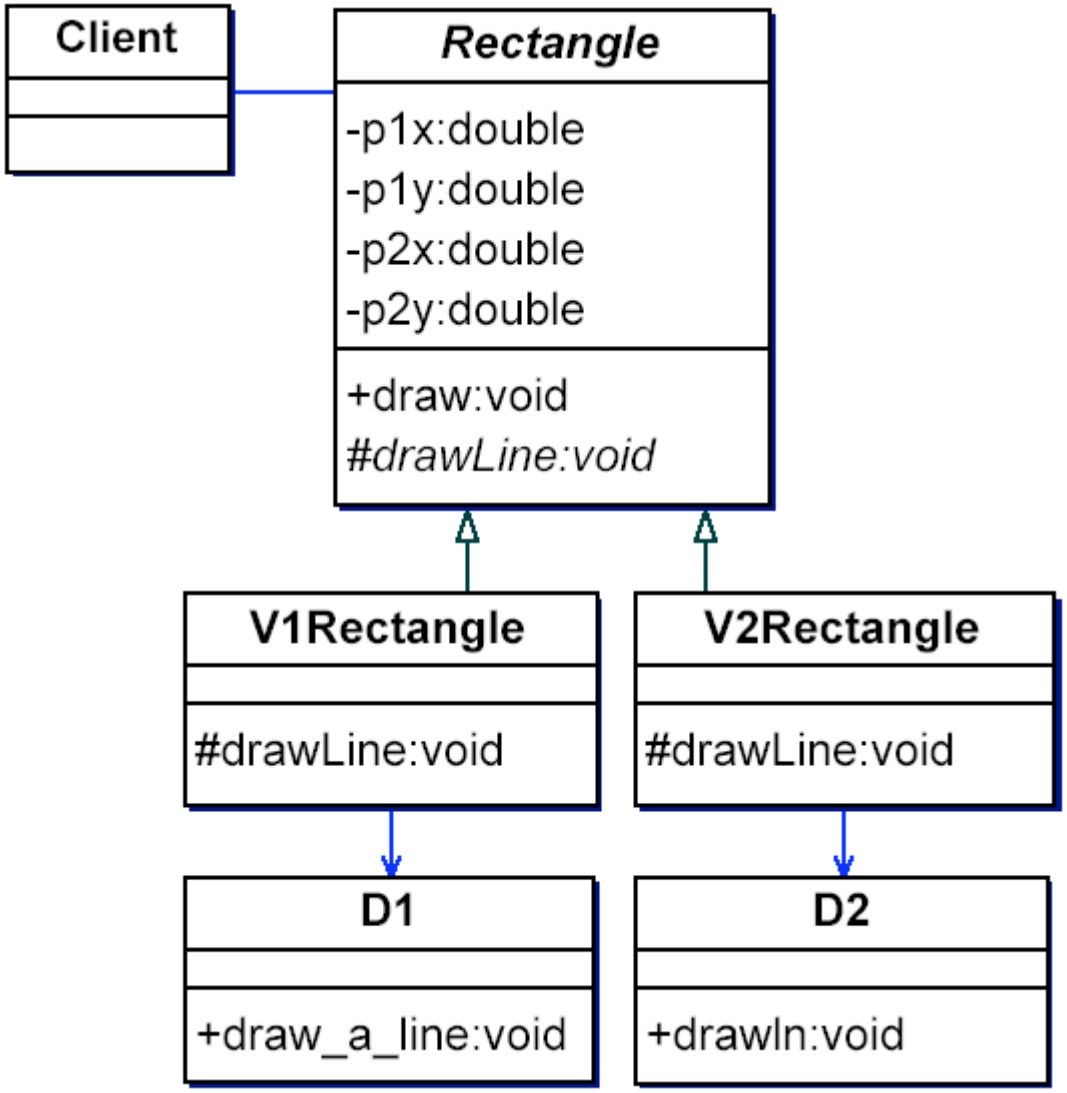
  - Les deux dimensions:
    - Ce que veut le client (abstraction)
    - ce que le système fournit

Bridge : pourquoi ?

- Un programme qui dessine des rectangles
  - Deux classes utilitaires D1 et D2 (pour raisons historiques, rendu différent, ...)
  - Chaque rectangle utilise uniquement une des deux classes
    - Donnée à la création du rectangle

- D1 :
  - draw_a_line(x1,y1,x2,y2)
  - draw_a_circle(x,y,r) – center and radius
- D2 :
  - drawln(x1,x2,y1,y2)
  - drawcr(x,y,r)

Design 1


Client
Rectangle
-p1x:double
-p1y:double
-p2x:double
-p2y:double
+draw:void
#drawLine:void
V1Rectangle
#drawLine:void
V2Rectangle
#drawLine:void
D1
+draw_a_line:void
D2
+drawln:void

Design 1

```
abstract class Rectangle {
public void draw() {
  drawLine(p1x,p1y,p2x,p1y);
  drawLine(p1x,p1y,p1x,p2y);
  drawLine(p2x,p2y,p2x,p1y);
  drawLine(p2x,p2y,p1x,p2y); }
protected abstract void drawLine
          (double,double,double,double);
private double p1x, p1y, p2x, p2y;
// constructor not shown }
```

```
class V1Rectangle extends Rectangle {
public void drawLine(double x1,double y1,
                     double x2,double y2)
   { d1.draw_a_line(x1,y1,x2,y2); }
public V1Rectangle(double x1,double y1,
          double x2,double y1,D1 d)
    { super(x1,y1,x2,y2); d1 = d; }
private D1 d1; }

class V2Rectangle extends Rectangle {
… d2.drawln(x1,x2,y1,y2) …}
```

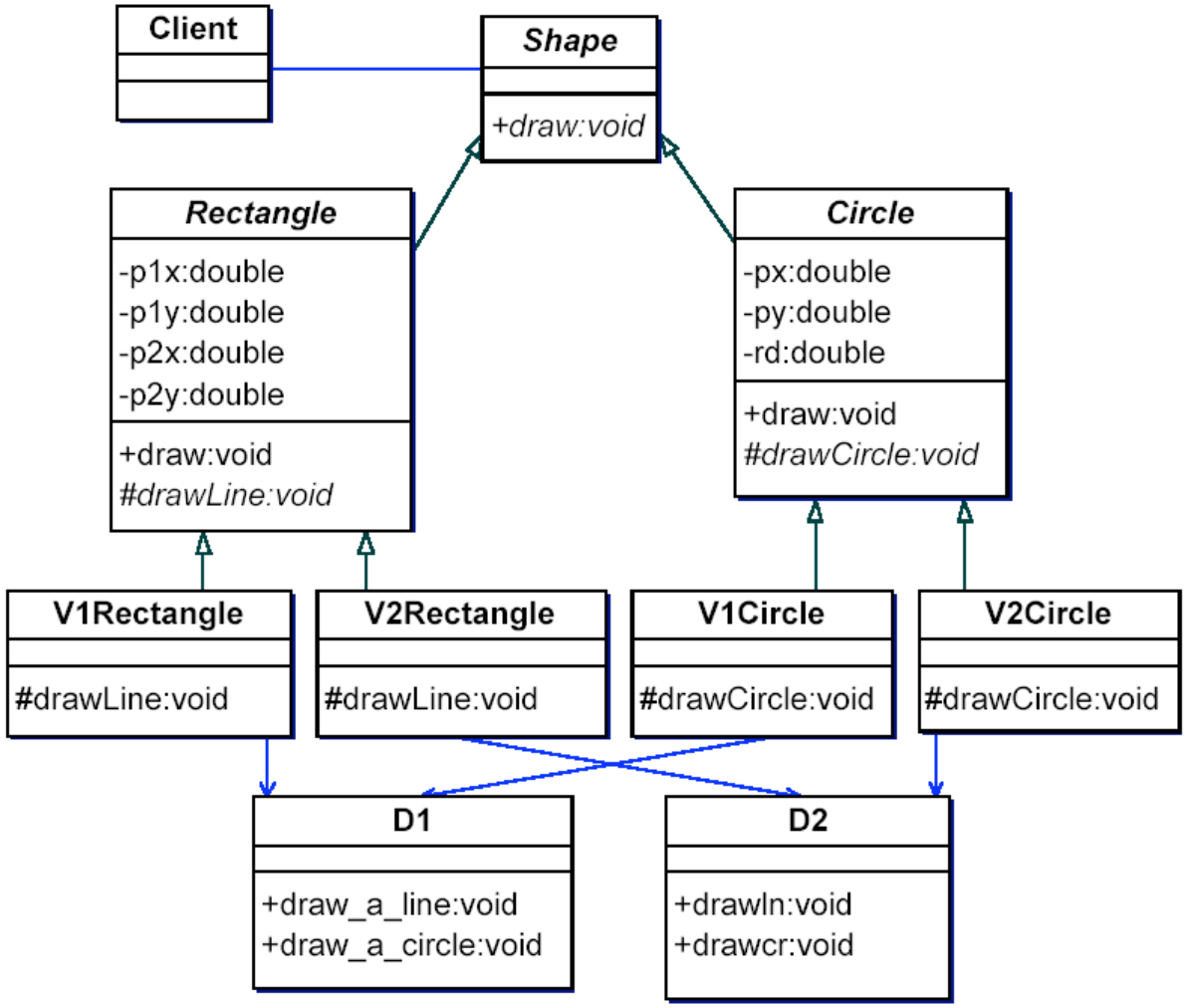
Nouveau besoin ...

- Le chef veut des cercles !
- Le code Client doit pouvoir manipuler des Cercles et des rectangles de manière uniforme (« dessines-moi toutes mes formes »)

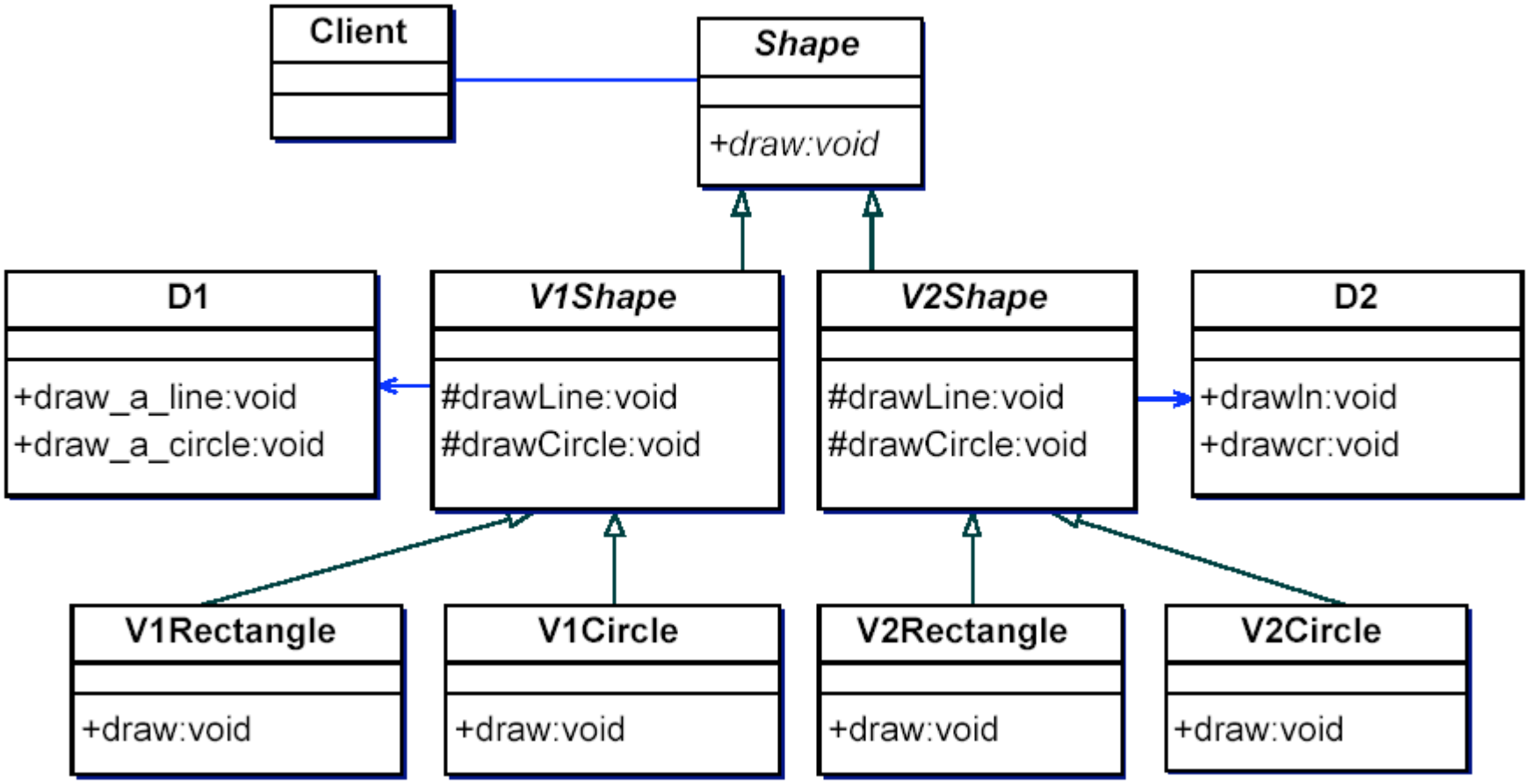
- Solution: superclass Shape + subclasses Circle et Rectangle

→ Réécriture du code client pour utiliser des Shape

Design 2


Client
Shape
+draw:void
Rectangle
-p1x:double
-p1y:double
-p2x:double
-p2y:double
+draw:void
#drawLine:void
Circle
-px:double
-py:double
-rd:double
+draw:void
#drawCircle:void
V1Rectangle
#drawLine:void
V2Rectangle
#drawLine:void
V1Circle
#drawCircle:void
V2Circle
#drawCircle:void
D1
+draw_a_line:void
+draw_a_circle:void
D2
+drawln:void
+drawcr:void

Design 3


Client
Shape
+draw:void
D1
+draw_a_line:void
+draw_a_circle:void
V1Shape
#drawLine:void
#drawCircle:void
V2Shape
#drawLine:void
#drawCircle:void
D2
+drawln:void
+drawcr:void
V1Rectangle
+draw:void
V1Circle
+draw:void
V2Rectangle
+draw:void
V2Circle
+draw:void

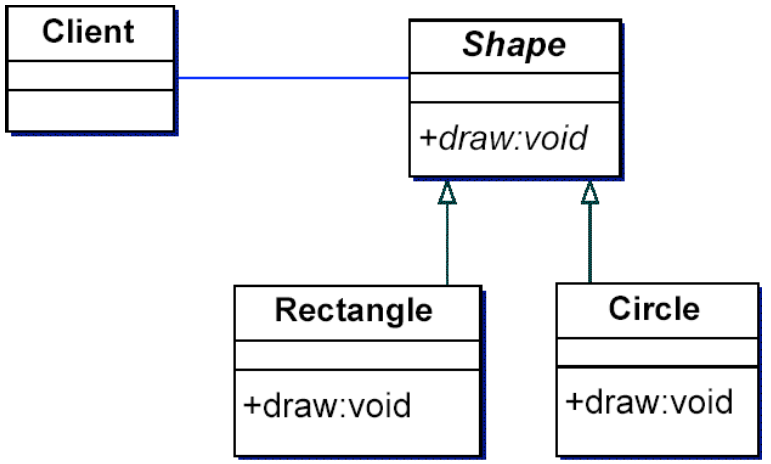
Compliqué ?

- Pourquoi ces problèmes ?
  - Utilisation abusive de l'héritage !
- Il y a deux dimensions de variabilité
  - Type de formes
  - Implémentations pour le type de rendu d'affichage
- Erreur classique :
  - Utiliser l'héritage quand la composition rend plus flexible

Bridge pattern

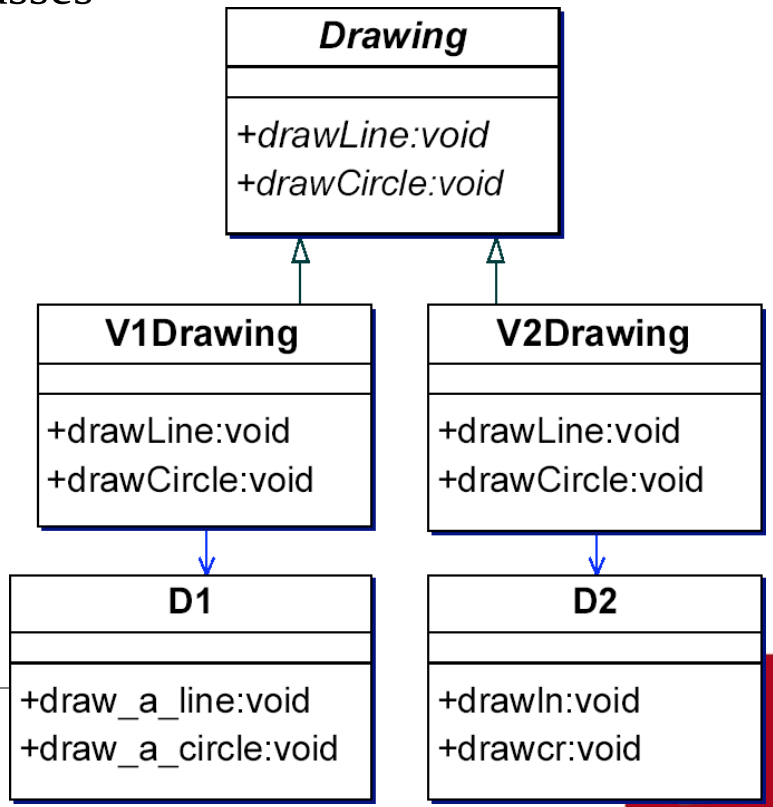
- Considérer les deux dimensions de manières in-dépendantes
  - Les connecter (par un pont) via la composition

  - 1ère dimension : Type de formes
    - Shape est une abstraction
      - Abstract class
    - Sous-classes Rectangle et Circle
    - Chaque Shape est reponsable de se dessiner
      - Abstract draw dans Shape
      - Impl. draw dans les sous-classes

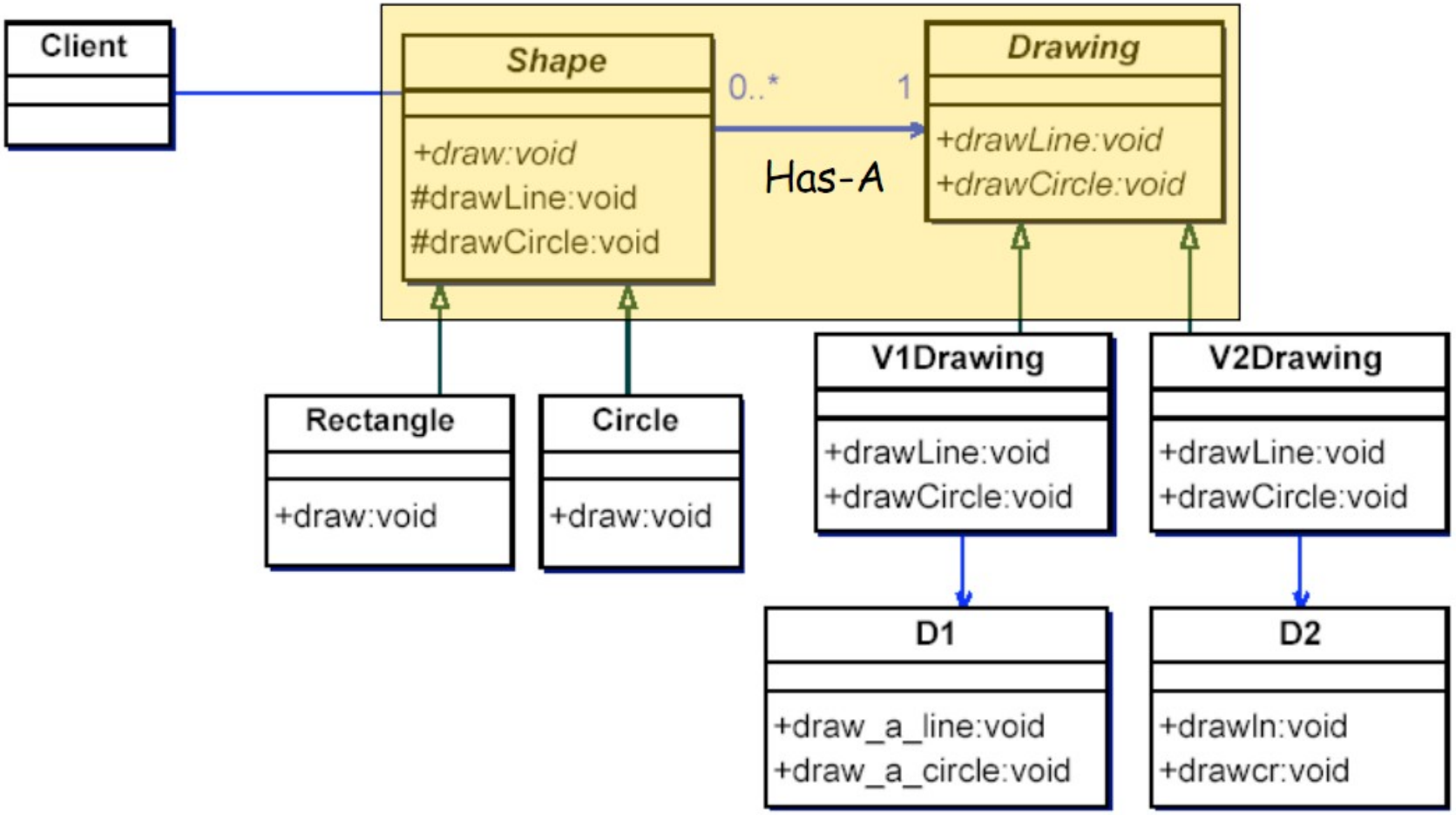


Bridge pattern

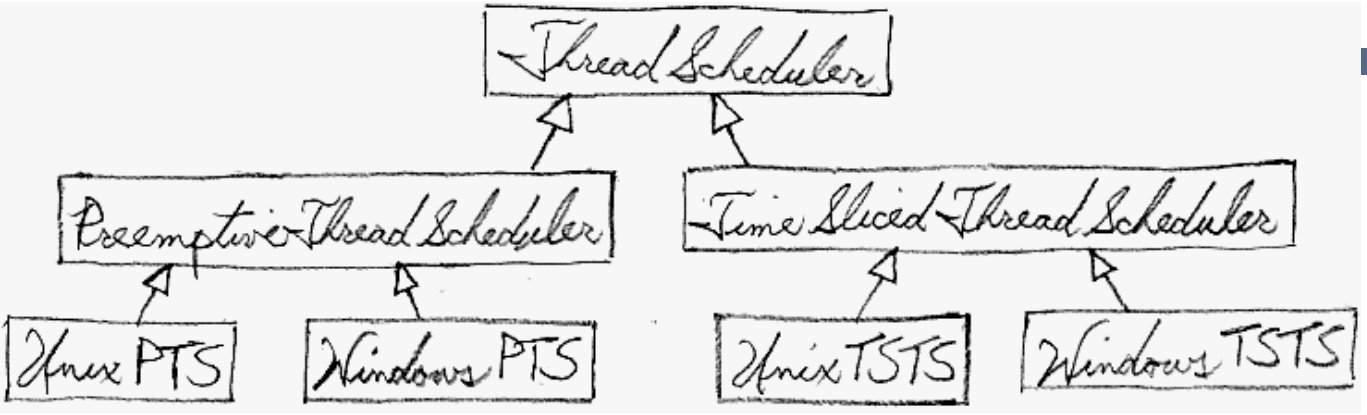
- 2ème dimension : Type de dessin
  - Abstract class : Drawing
  - Sous-classes V1Drawing et V2Drawing
  - Chaque classe est reponsable de savoir comment dessiner des lignes ou des cercles
    - Abstract methods drawLine() et drawCircle() dans Drawing
    - Impl. DrawLine et drawCircle dans les sous-classes
  - Association avec les classes D1/D2

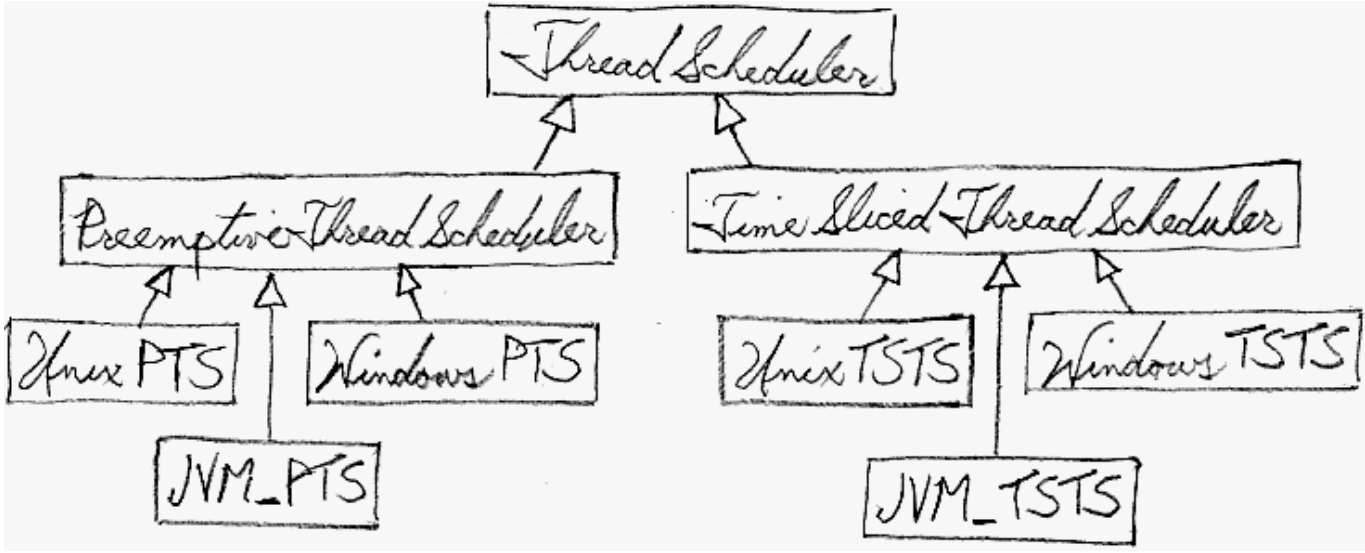


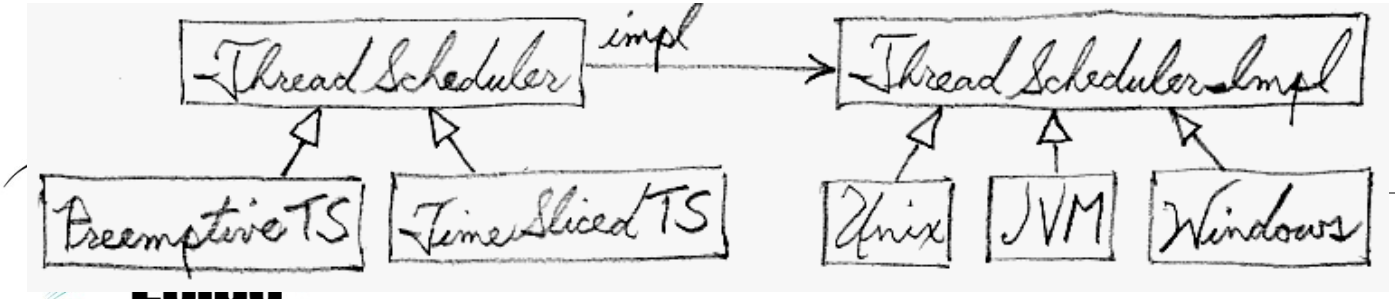
Bridge : Solution !



Bridge : exemple



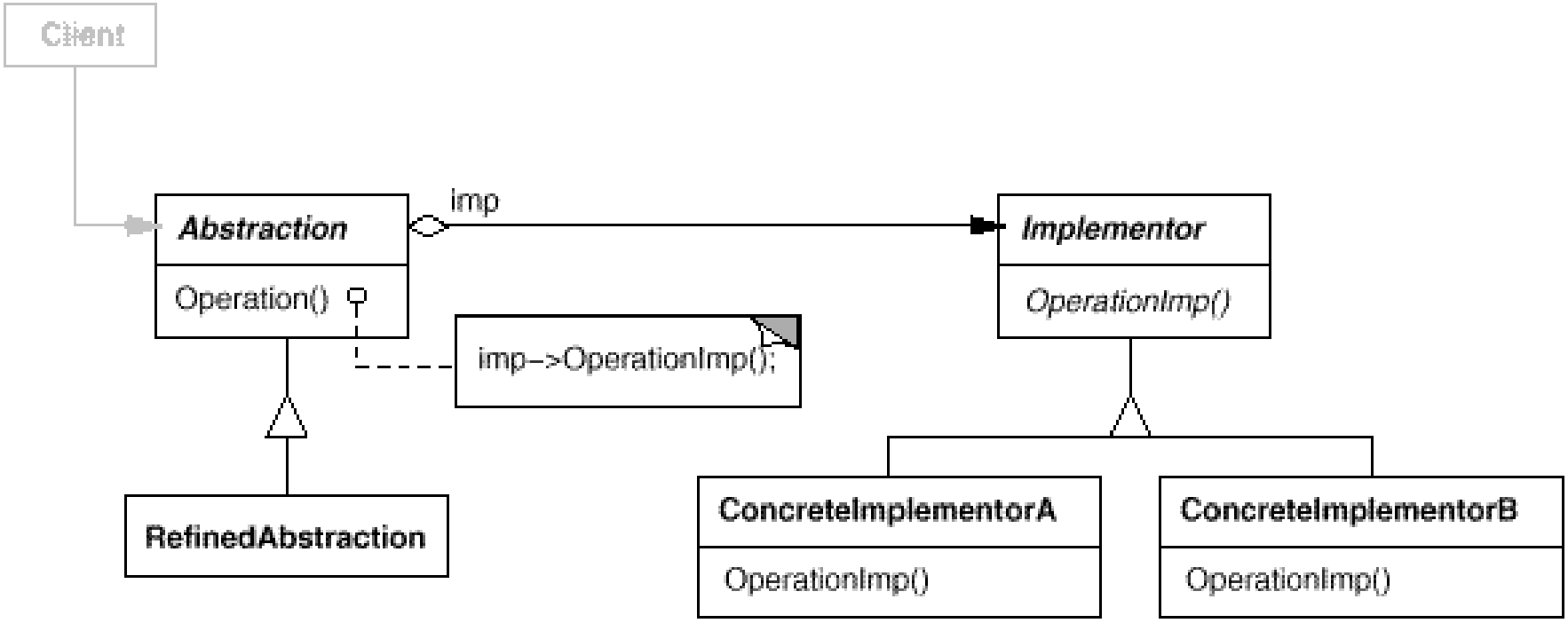




Bridge : check-list

- Des dimensions orthogonales ?
- Séparer :
  - Que veut le client ?
  - Qu'est-ce que la plate-forme impose/fournit ?
- Concevoir l'interface « orientée plate-forme » (imple-mentor) de manière minimale, nécessaire et suffisante !
  - Objectif : découpler l'abstraction de la plateforme
- Definir la hiérarchie de classe / plateforme
- Définir l'abstraction qui A UNE plateforme et lui délègue ce qui est dépendant de la plateforme
- Definir les spécialisations de cette abstraction
- En général, utilisation de Factories

Bridge


Client
Abstraction
Operation()
imp
Implementor
OperationImp()
imp->OperationImp();
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

DP structurels: synthèse

- Comment les classes et les objets collaborent pour créer des structures plus importantes
  - Adapter: 2 interfaces incompatibles
  - Composite: composition récursive en définissant une abstraction commune pour une composition et un composant.
  - Decorator: associer dynamiquement de nouvelles responsabilités à un objet. Alternative flexible à l'héritage.
  - Facade: une interface plus simple
  - Flyweight: utiliser le partage pour optimiser l'usage d'un grand nombre de petits objets
  - Bridge: indépendance entre l'abstraction et certains aspects de l'implémentation
  - Proxy: utiliser un niveau supplémentaire d'indirection pour supporter un accès contrôlé, distant, plus généralement « - techniquement intelligent »