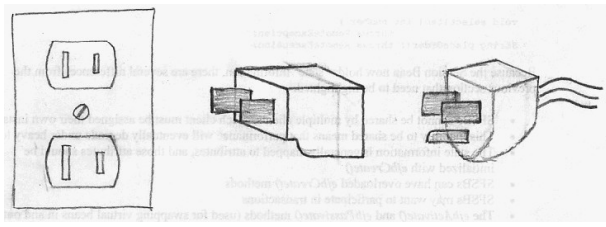


Patrons structurels

- On a des classes, il faut les organiser pour qu'elles fonctionnent ensemble
- Lise encadre les dessins de Marc
- Lise dessine un trait, Marc un ensemble de traits
- Lise fournit un `List<T>`
 - Marc veut un `T[]`
- Lise veut juste regarder un DVD, Marc lui demande de trouver les 3 télécommandes, d'allumer la TV, le lecteur DVD, l'ampli, ...
- Il y a tellement de cases dans mon monde, je n'ai pas assez de mémoire. Et pourtant elles se ressemblent beaucoup.
- Lise manipule un ensemble de Portes/Fenêtres/Pièces
 - Marc préfère un ensemble de Cloison/Mur/Ouverture
- Marc demande à Lise pour la 40e fois « quel âge as-tu ? »

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, un peu de mémoire !



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 = adapted.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 adapted.get();  
        }  
        catch (NoSuchElementException e) { return null; }  
    }  
}
```

Adapter : exemple

*line from (10,20) to (30,60)
rectangle at (10,20) with width 20 and height 40*

```
class LegacyLine {
    public void draw( int x1, int y1, int x2, int y2 ) {
        System.out.println( "line from (" + x1 + ',' + y1
            + ") to (" + x2 + ',' + y2 + ')' );
    } }
class LegacyRectangle {
    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 AdapterDemo {
    public static void main( String[] args ) {
        Object[] shapes = { new LegacyLine(),
            new LegacyRectangle() };
        // 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)
            if (shapes[i].getClass().getName()
                .equals("LegacyLine"))
                ((LegacyLine)shapes[i]).draw( x1, y1, x2, y2 );
            else if (shapes[i].getClass().getName()
                .equals("LegacyRectangle"))
                ((LegacyRectangle)shapes[i]).draw(
                    Math.min(x1,x2), Math.min(y1,y2),
                    Math.abs(x2-x1), Math.abs(y2-y1) );
    } }
```

```
class LegacyLine { ... }
class LegacyRectangle { ... }

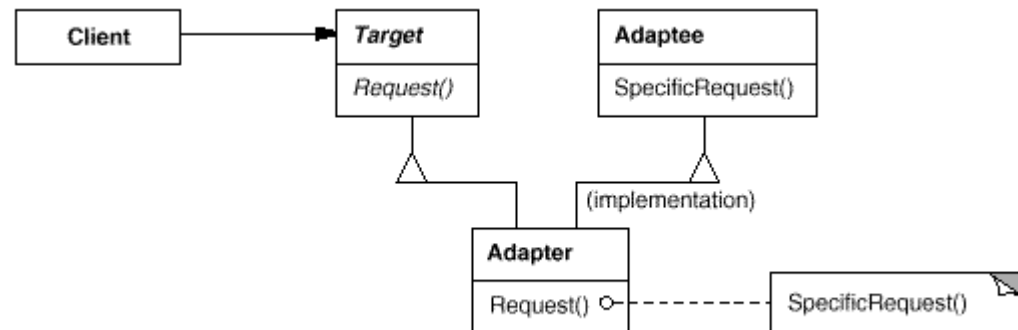
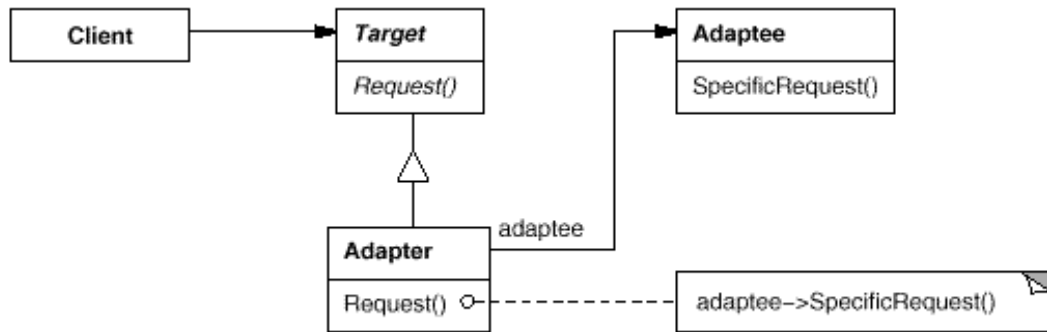
interface Shape {
    void draw( int x1, int y1, int x2, int y2 );
}
class Line implements Shape {
    private LegacyLine adaptee = new LegacyLine();
    public void draw( int x1, int y1, int x2, int y2 ) {
        adaptee.draw( x1, y1, x2, y2 );
    } }
class Rectangle implements Shape {
    private LegacyRectangle adaptee = new LegacyRectangle();
    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 AdapterDemo {
    public static void main( String[] args ) {
        Shape[] shapes = { new Line(), new Rectangle() };
        // 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 );
    } }
```

© Vince Huston

}}
POO

7

Adapter's



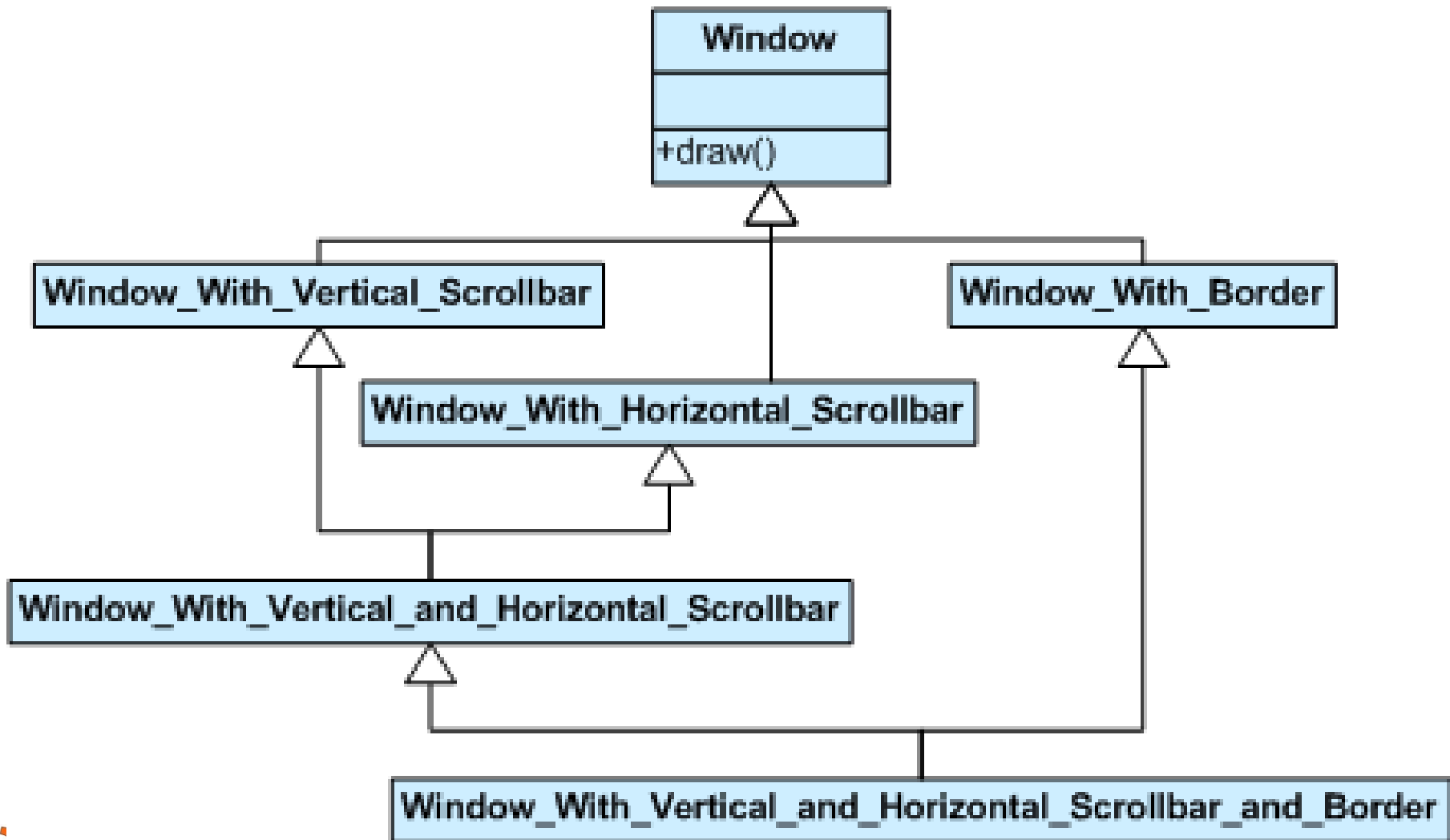
Decorator : quand

- 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, on lit les données avec l'objet initial, on les 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

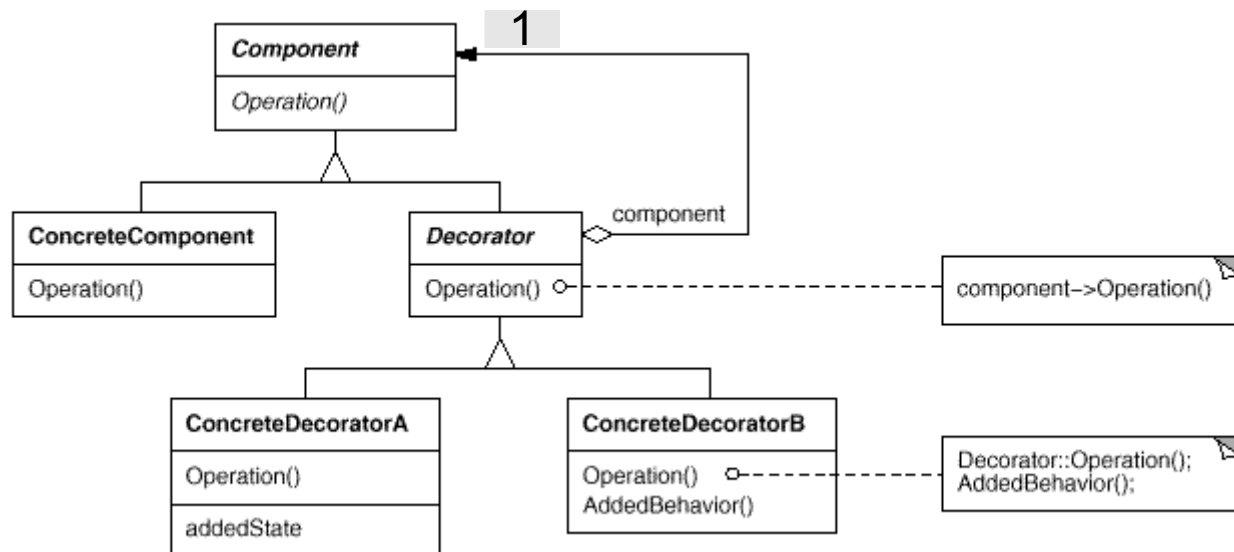


Decorator : exemple

```
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



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

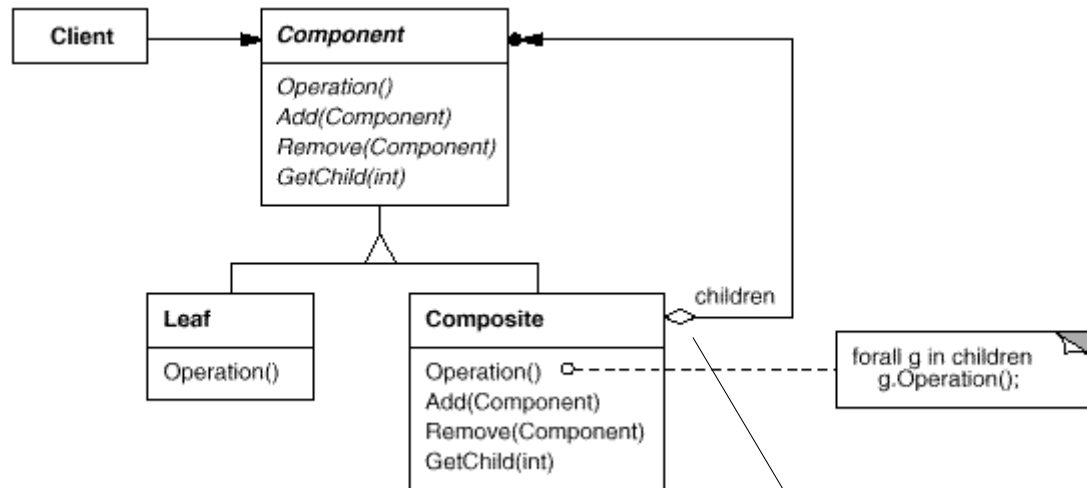
```
public interface Drawable {  
    void draw(Graphics g);  
}
```

```
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 ?

Agrégation ou Composition

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 ...

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

```
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();
    }
}
```

```
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( Object obj ) {m_files.add( obj ); }
    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();
    }
}
```

Facade : quand

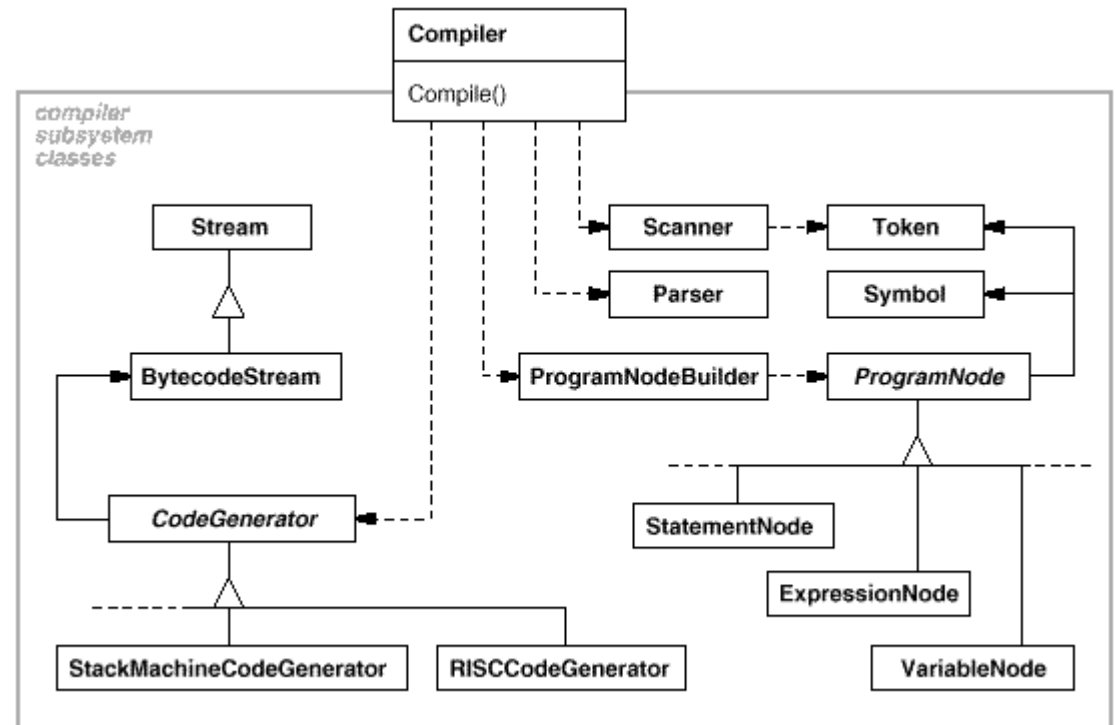
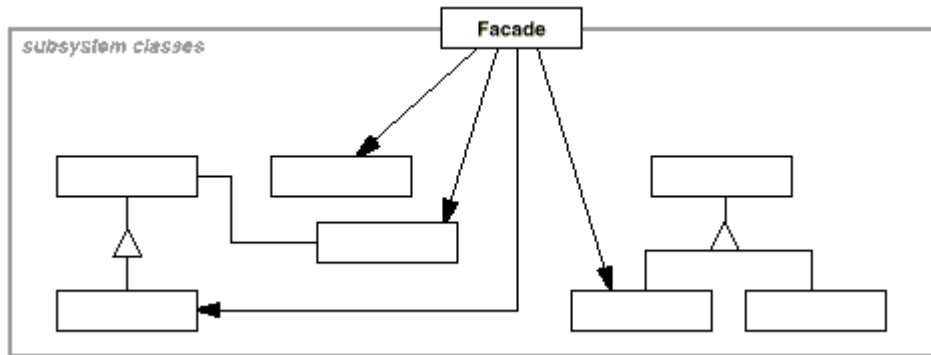
- L'interface d'un paquetage est complexe, *trop* pour son besoin

```
Iterator<Lexem> lexems = Lexer.lexicalAnalysis(input);
AST tree = Parser.syntaxAnalysis(lexems);
SymbolTable table = new SymbolTable();
Semantics.semanticAnalysis(table, tree);
ByteCode code =
Compiler.intermediateCode(table, ast);
CodeCompiler compiler = CompilerFactory.getByName("i386");
compiler.compile(code);
compiler.writeOut(new File("a.out"));
```

Facade : comment

- On crée une classe particulière responsable de réaliser l'interface entre le paquetage et ses clients
- Elle sera le seul point d'entrée du paquetage
- Pour l'exemple, on créera un classe Compiler
 - avec les différentes méthodes à appeler
 - avec une méthode qui effectue la compilation complète

Façade



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

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 ExternContext {  
    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 ArrayList<FlyweightChar> characters;  
    ...  
    public void draw(Graphics g, ExternalContext c) {  
        for(int i=0; i<characters.size(); i++) {  
            c.setPosition(i);  
            characters.get(i).draw(g, c);  
        }  
    }  
}
```

2 niveaux de
calculs des
données
extrinsèques

Rq: ici, même fonte
sur toute la ligne

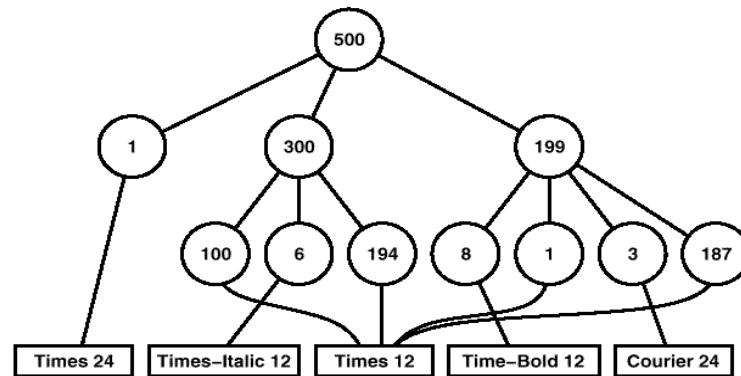
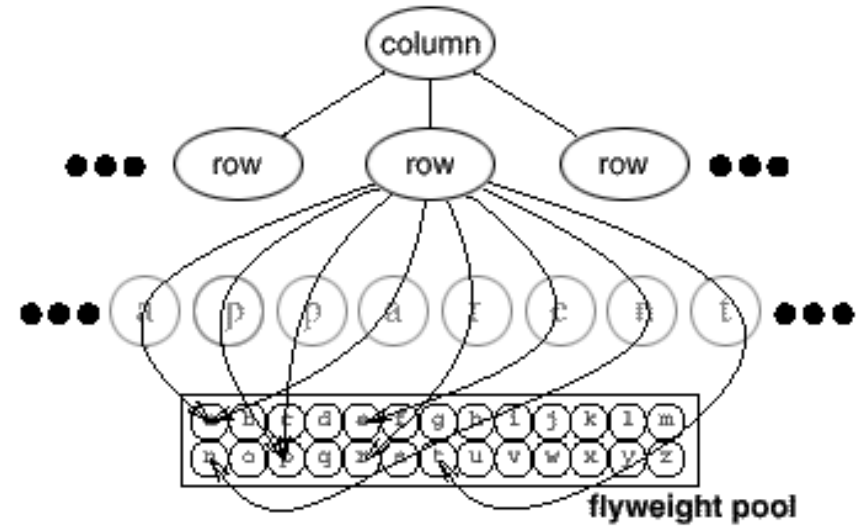
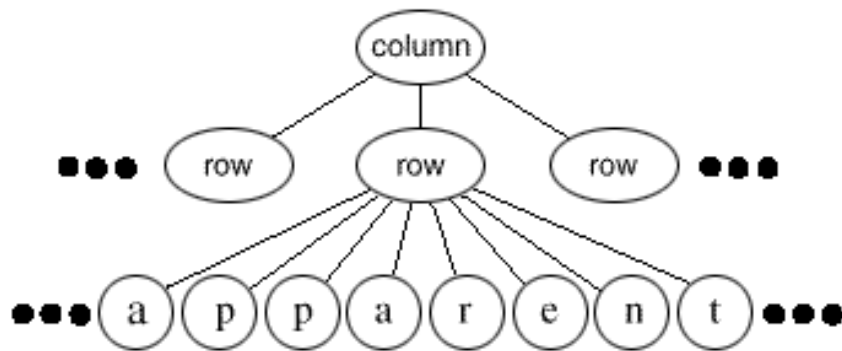
Flyweight : exemple

```
import gnu.trove.TIntObjectHashMap;
```

Optimisée avec des clés « int »

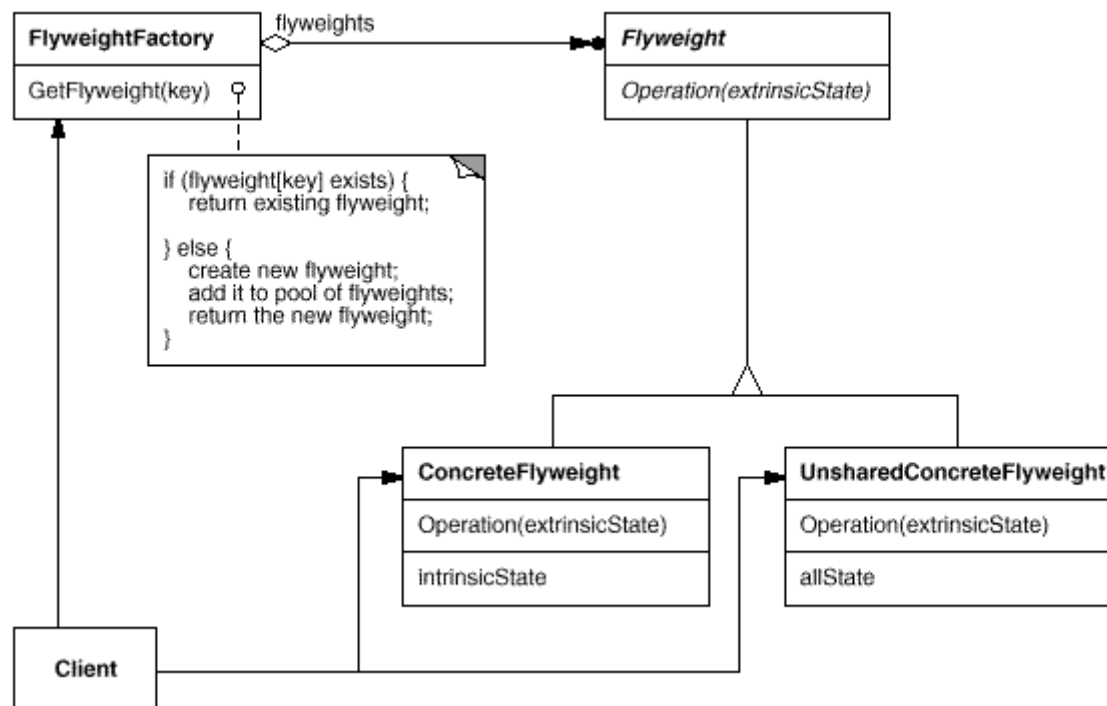
```
public class FlyweightFactory {  
    private final TIntObjectHashMap<FlyweightChar> flyweightPool =  
        new TIntObjectHashMap<FlyweightChar>();  
    public FlyweightChar instance(char c) {  
        FlyweightChar fly = flyweightPool.get(c);  
        if (fly == null) {  
            fly = new FlyweightChar(c);  
            flyweightPool.put(c,fly);  
        }  
        return fly;  
    }  
}
```

Flyweight : exemple



Gestion de ExternContext...

Flyweight



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

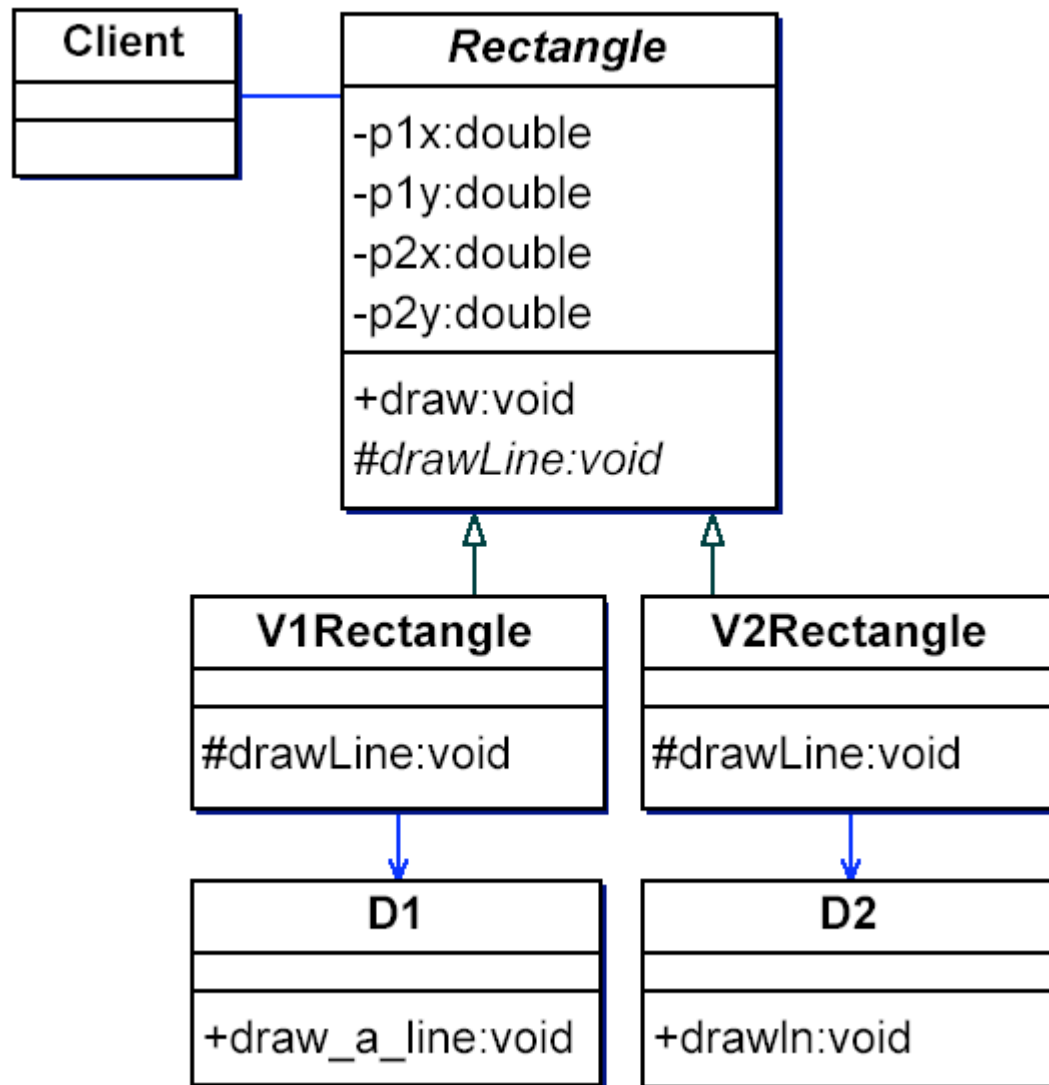
- GoF: *“Decouple an abstraction from its implementation so that the two can vary independently”*
- Élément clé : dimensionS de variabilité
 - Si une seule dimension, simple polymorphisme basé sur héritage ou interface
 - Par exemple, différentes « formes » (Shape)
- Si plusieurs dimensionS ?
 - Abstraction « métier » / Plate-forme
 - domaine/infrastructure
 - front-end/back-end
- Les deux dimensions:
 - Ce que veut le client (abstraction)
 - ce que le système fournit

© BG Ryder/A Rountev

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



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 }
```

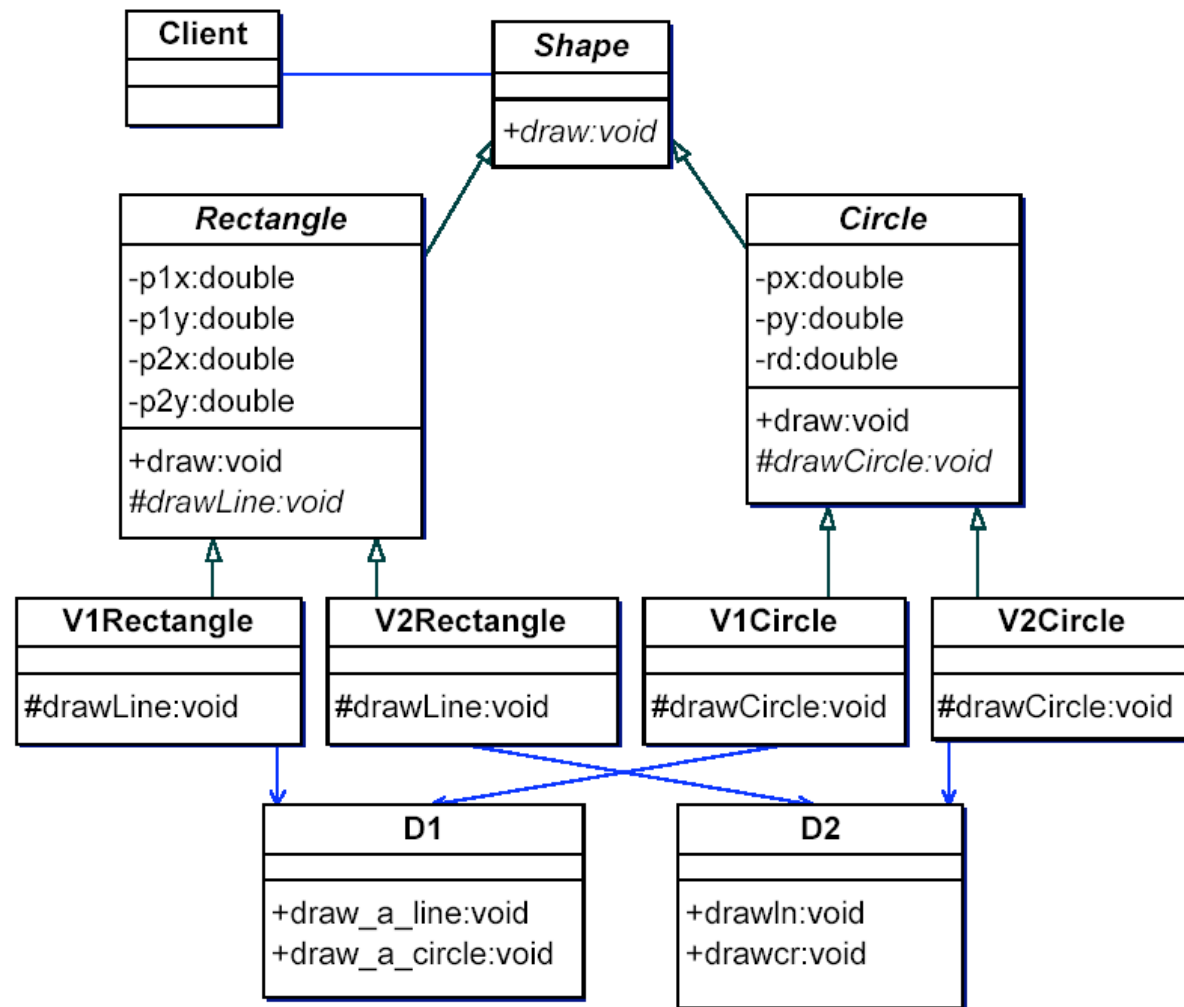
```
class V1Rectangle extends Rectangle {  
    public void drawLine(double x1,double y1,  
        double x2,double y2)  
        { d1.draw_a_line(x1,y1,x2,y2); }  
    public V1Rectangle(double x1,double y1,  
        double x2,double y1,D1 d)  
        { super(x1,y1,x2,y2); d1 = d; }  
    private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

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

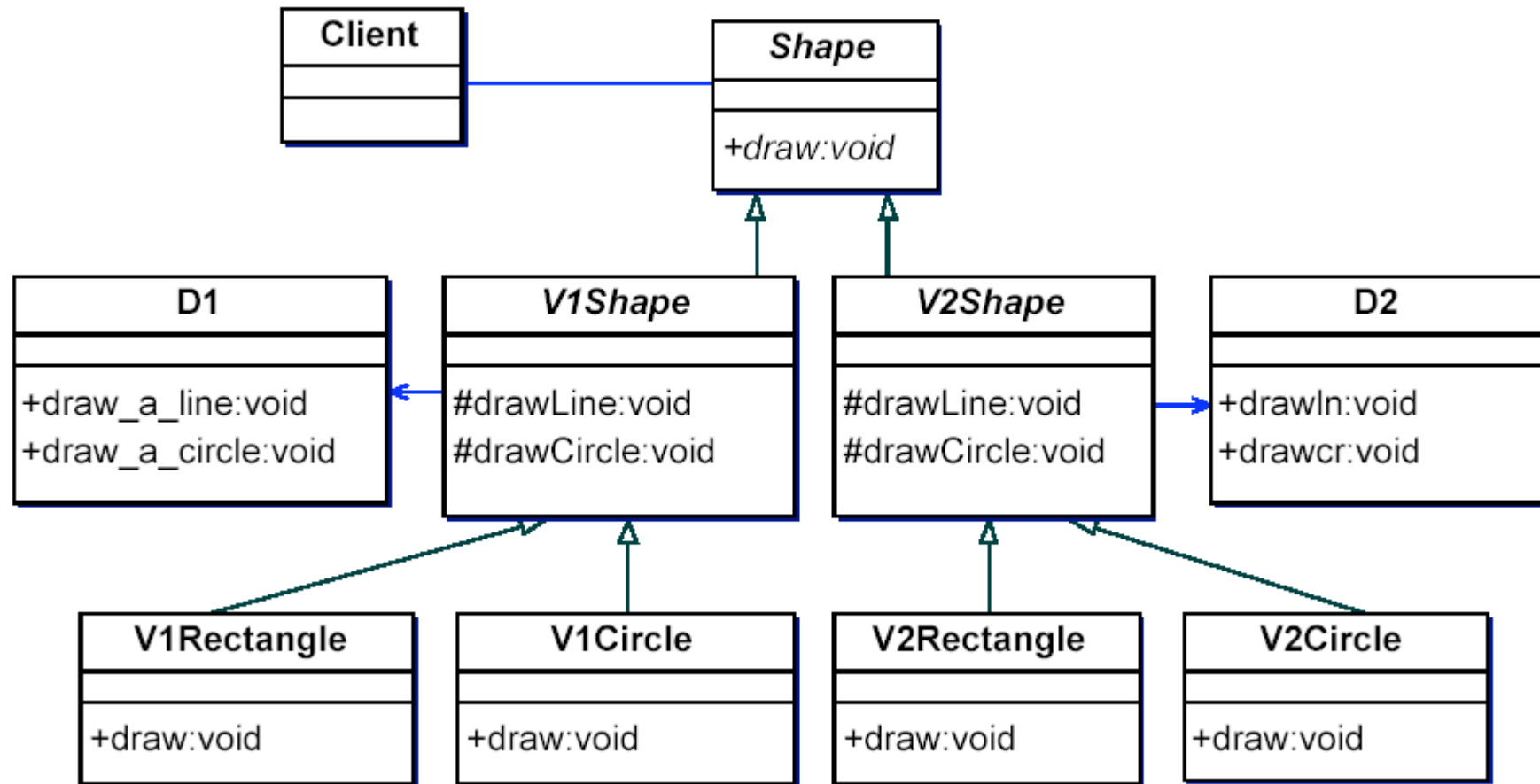
Nouveau besoin ...

- Le chef veut des cercles !
- Le code Client doit pouvoir manipuler des Cercles et des rectangles de manière uniforme (« dessi-
nes-moi toutes mes formes »)
- Solution: superclass Shape + subclasses Circle et Rectangle
→ Réécriture du code client pour utiliser des Shape

Design 2



Design 3

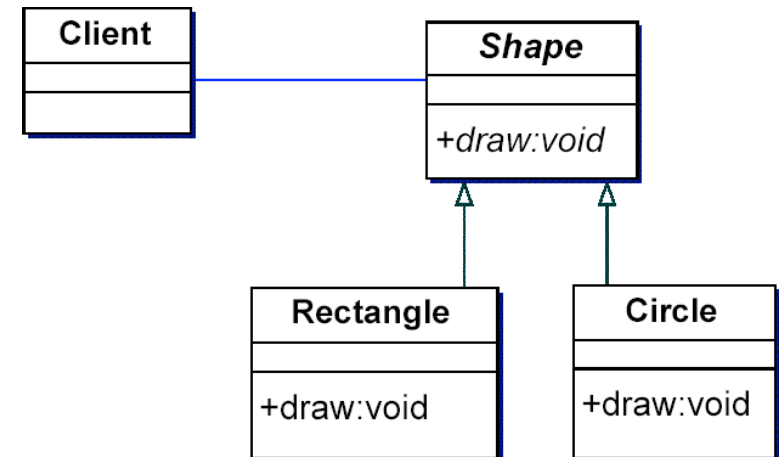


Complicqué ?

- 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

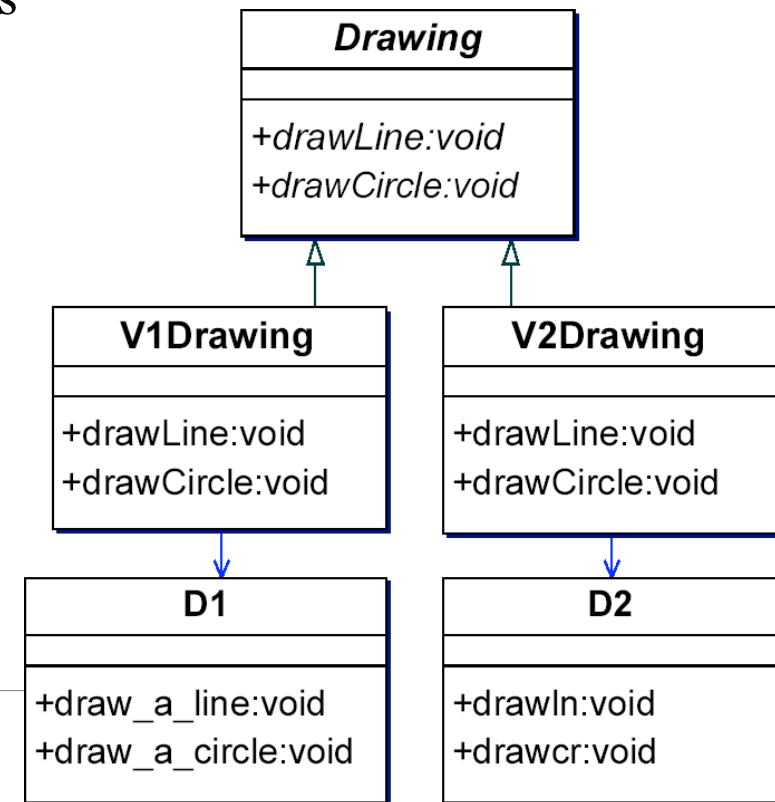
Bridge pattern

- Considérer les deux dimensions de manières indépendantes
- 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 responsable de se dessiner
 - Abstract draw dans Shape
 - Impl. draw dans les sous-classes

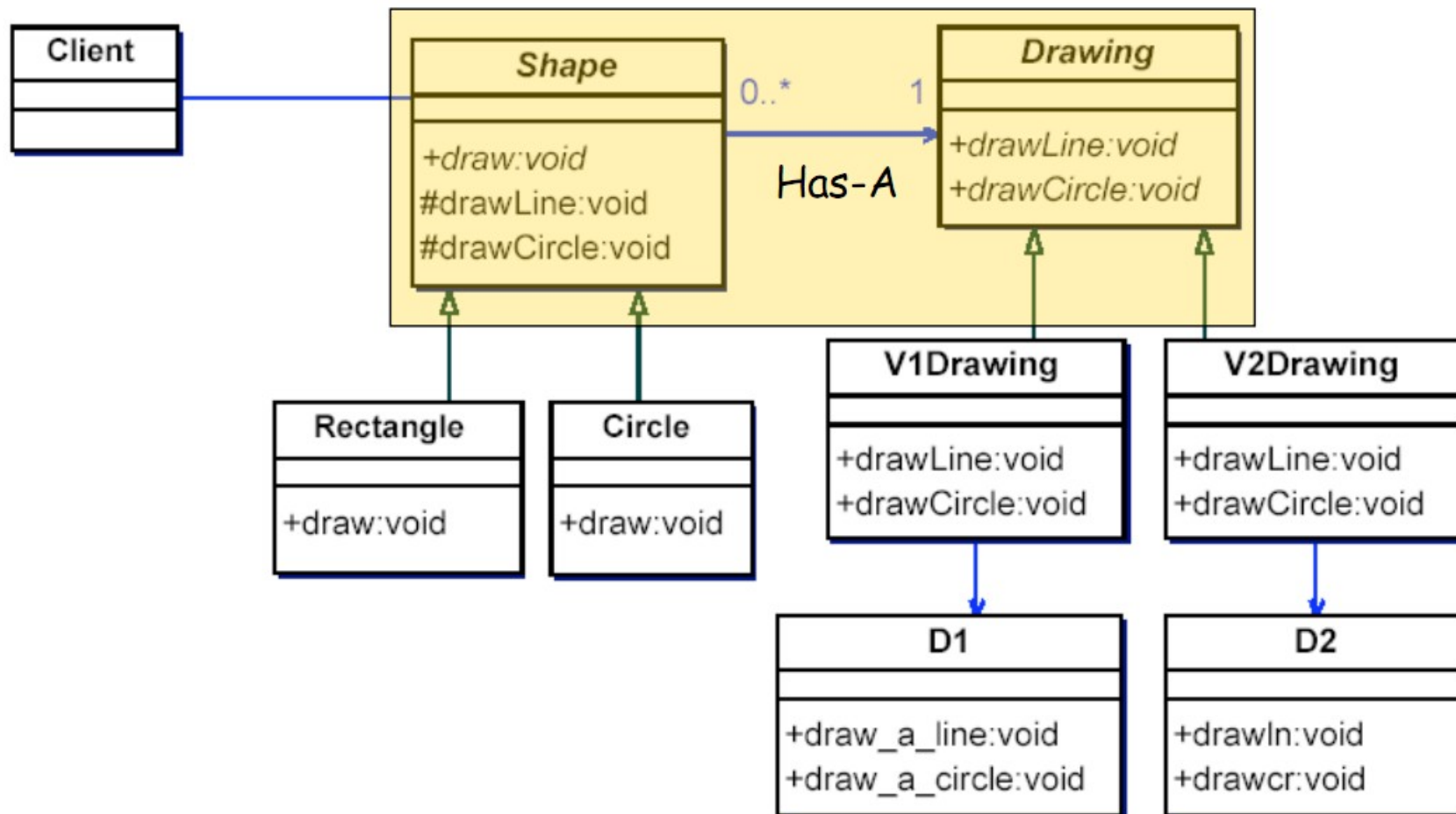


Bridge pattern

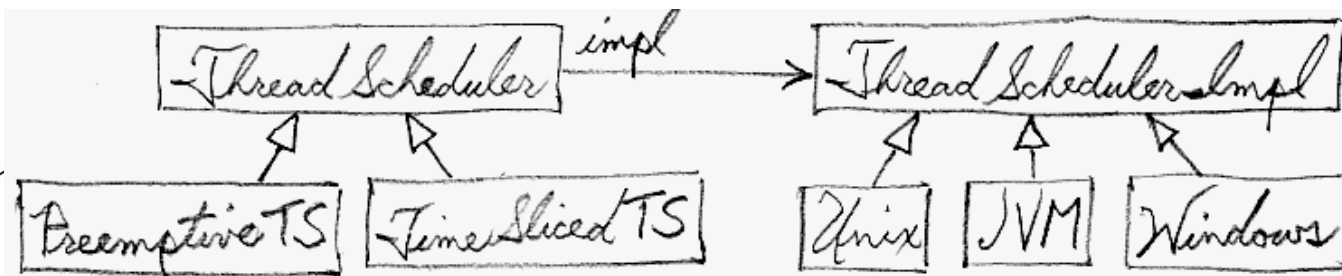
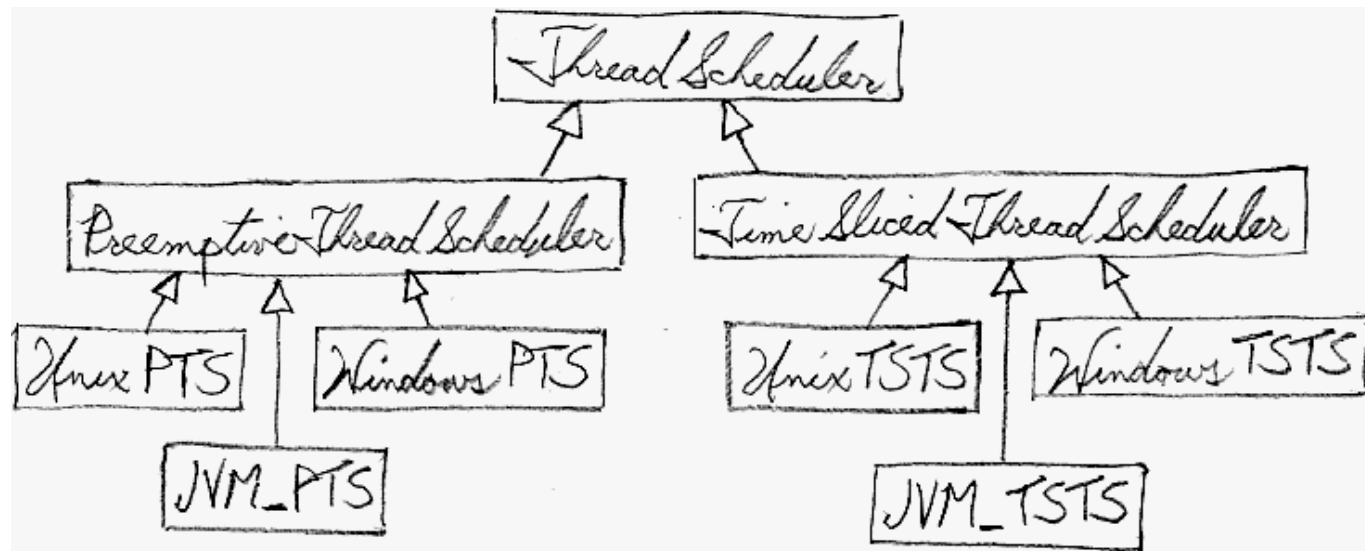
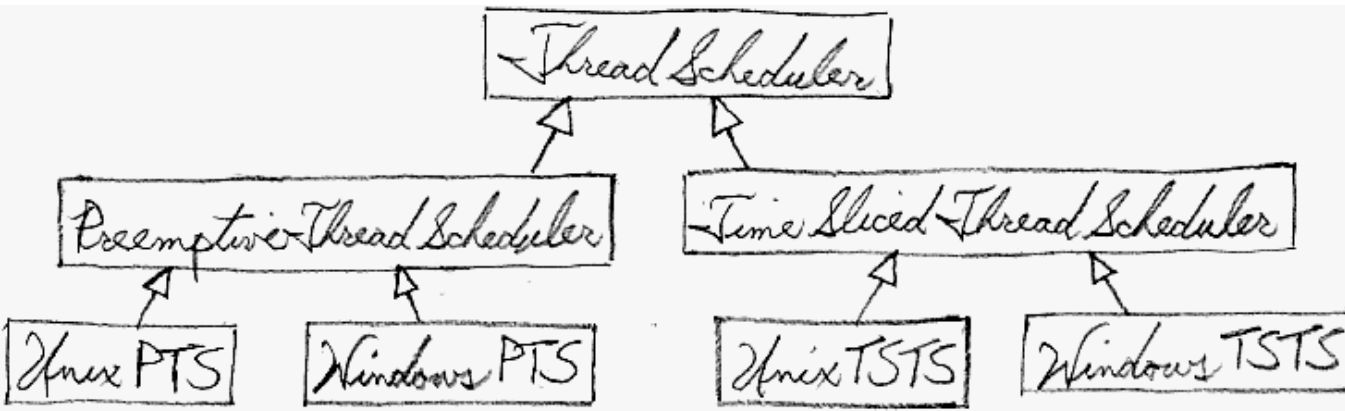
- 2ème dimension : Type de dessin
 - Abstract class : Drawing
 - Sous-classes V1Drawing et V2Drawing
 - Chaque classe est responsable de savoir comment dessiner des lignes ou des cercles
 - Abstract methods drawLine() et drawCircle() dans Drawing
 - Impl. DrawLine et drawCircle dans les sous-classes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



Bridge : Solution !



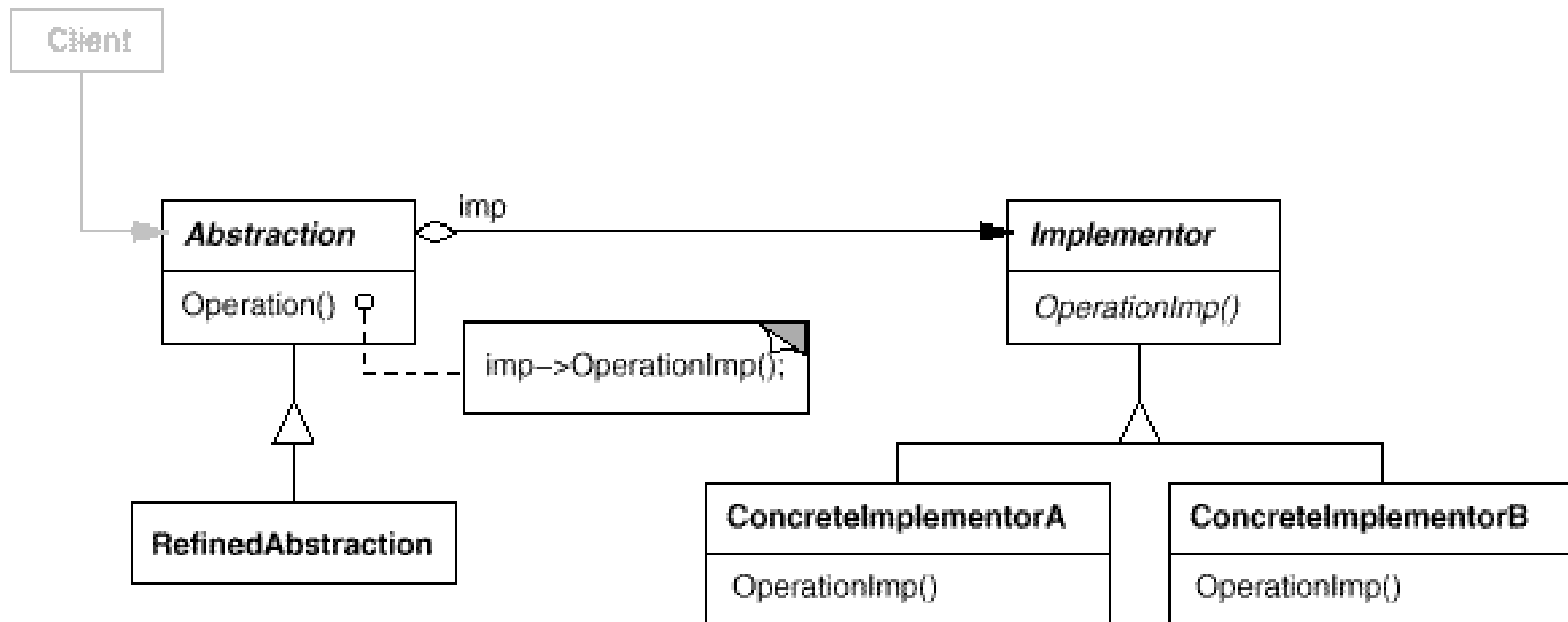
Bridge : exemple



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 » (implementor) de manière minimale, nécessaire et suffisante !
 - Objectif : découpler l'abstraction de la plateforme
- Définir la hiérarchie de classe / plateforme
- Définir l'abstraction qui A UNE plateforme et lui délègue ce qui est dépendant de la plateforme
- Définir les spécialisations de cette abstraction
- En général, utilisation de Factories

Bridge



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

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

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

Dépendance :
peu
acceptable

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();  
    ...  
}
```

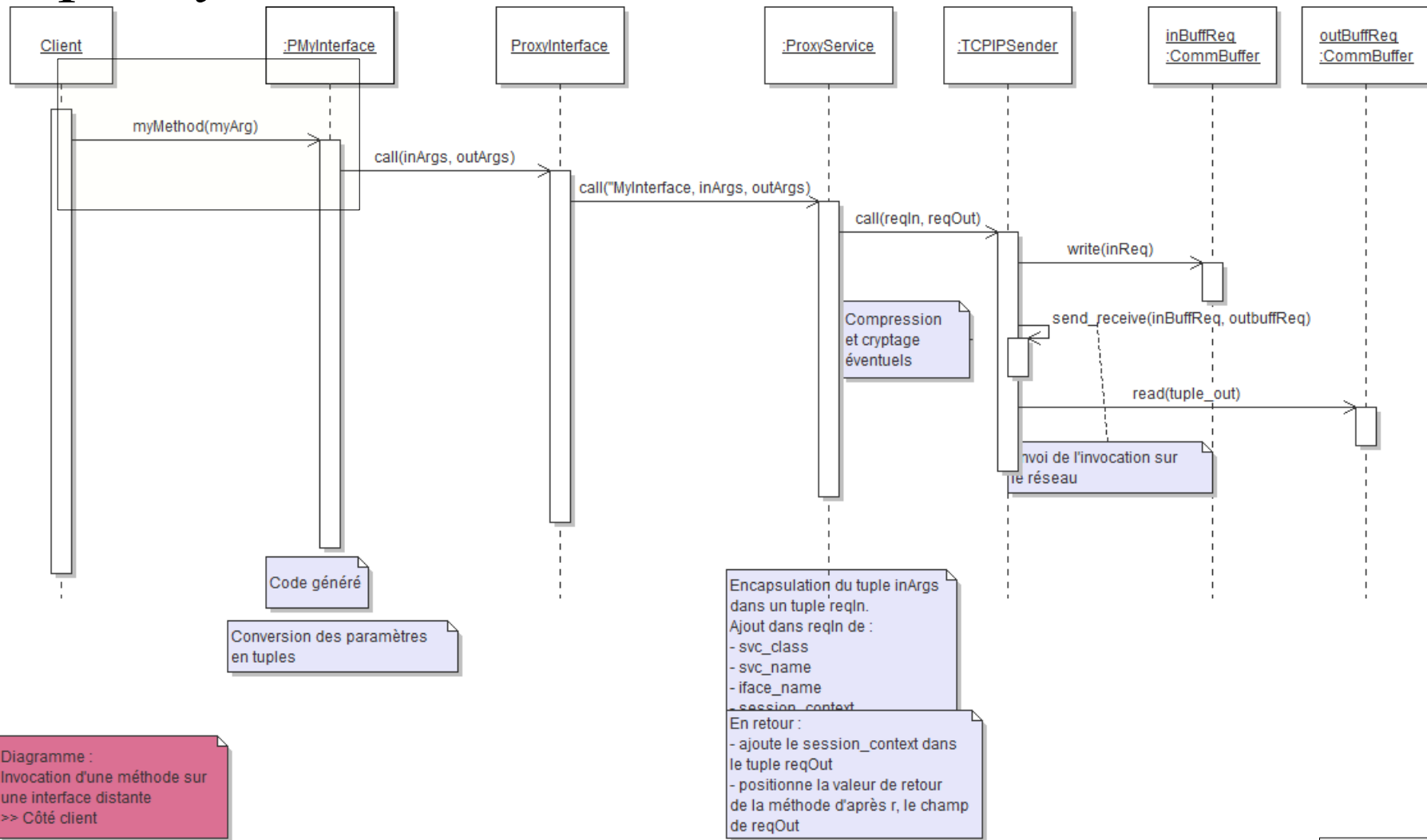
```
class MyServiceImpl implements MyService {  
    public void action1() { ...}  
    ...  
}
```

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

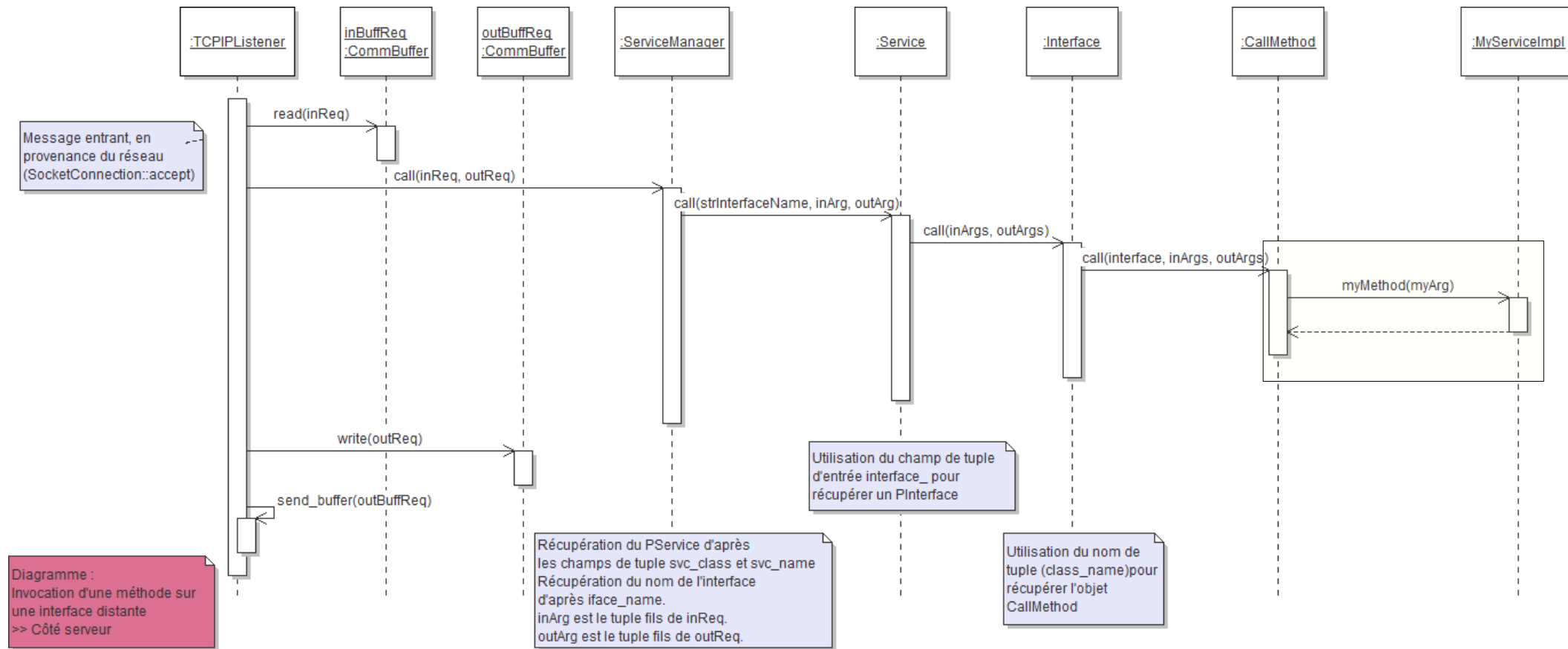
```
class Test {  
    public static void main(...) {  
        MyService service =  
            ServiceFactory.create(...);  
        service.action1();  
    }  
}
```

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() { ...}  
    ...  
}
```

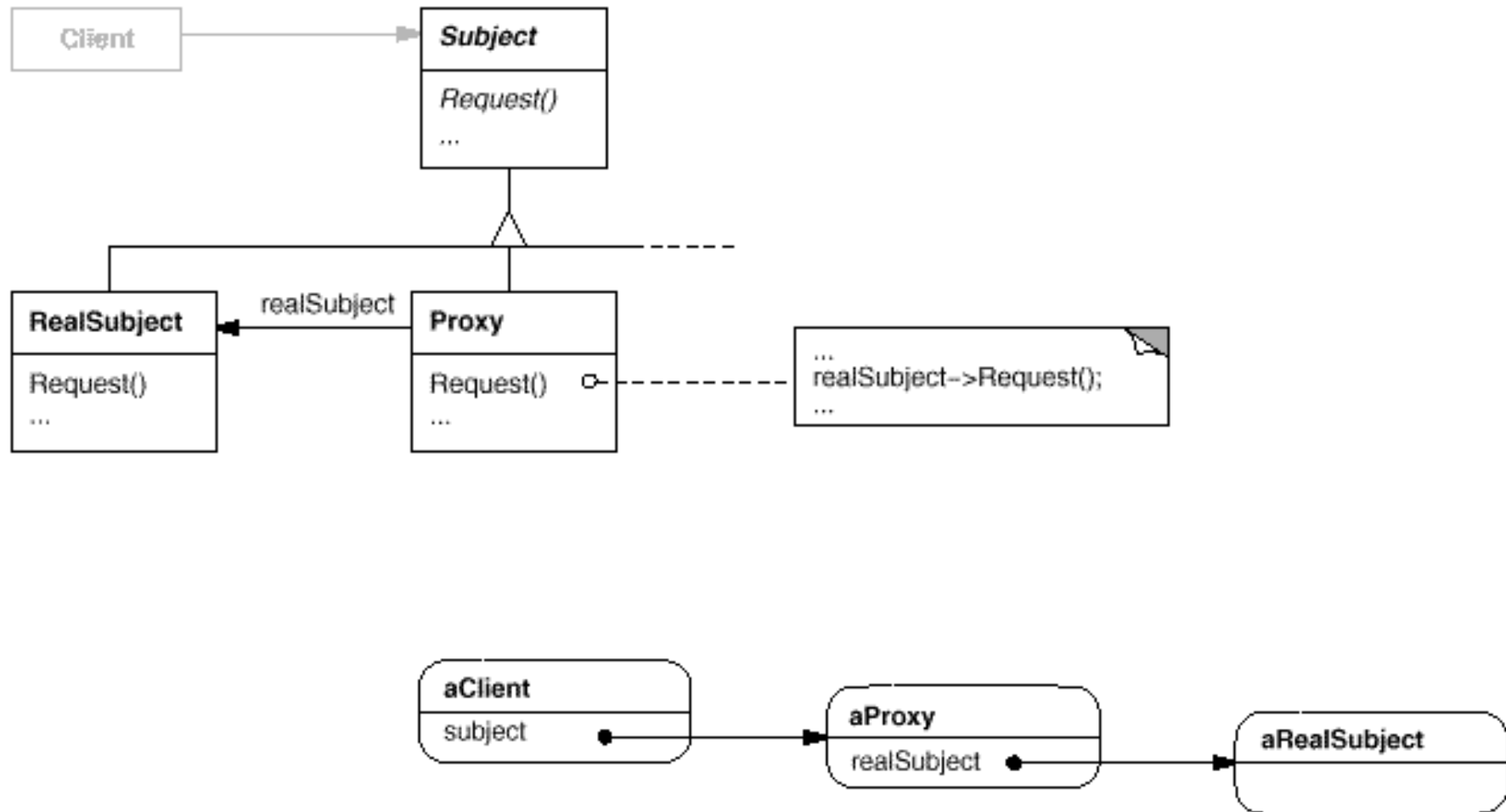
```
class MyServiceProxy implements MyService {  
    public void action1() {  
        log.finest(« ... action1 ... »);  
        realObject.action1();  
    }  
    MyServiceProxy(MyService o, Logger log)  
        { realObject=o;}  
    private MyService realObject;  
}
```

```
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
- Construction de l'objet réel à l'initialisation ou à la première utilisation
- Traitement délégué à l'objet réel, pré/post-traitement possible

Proxy



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 »