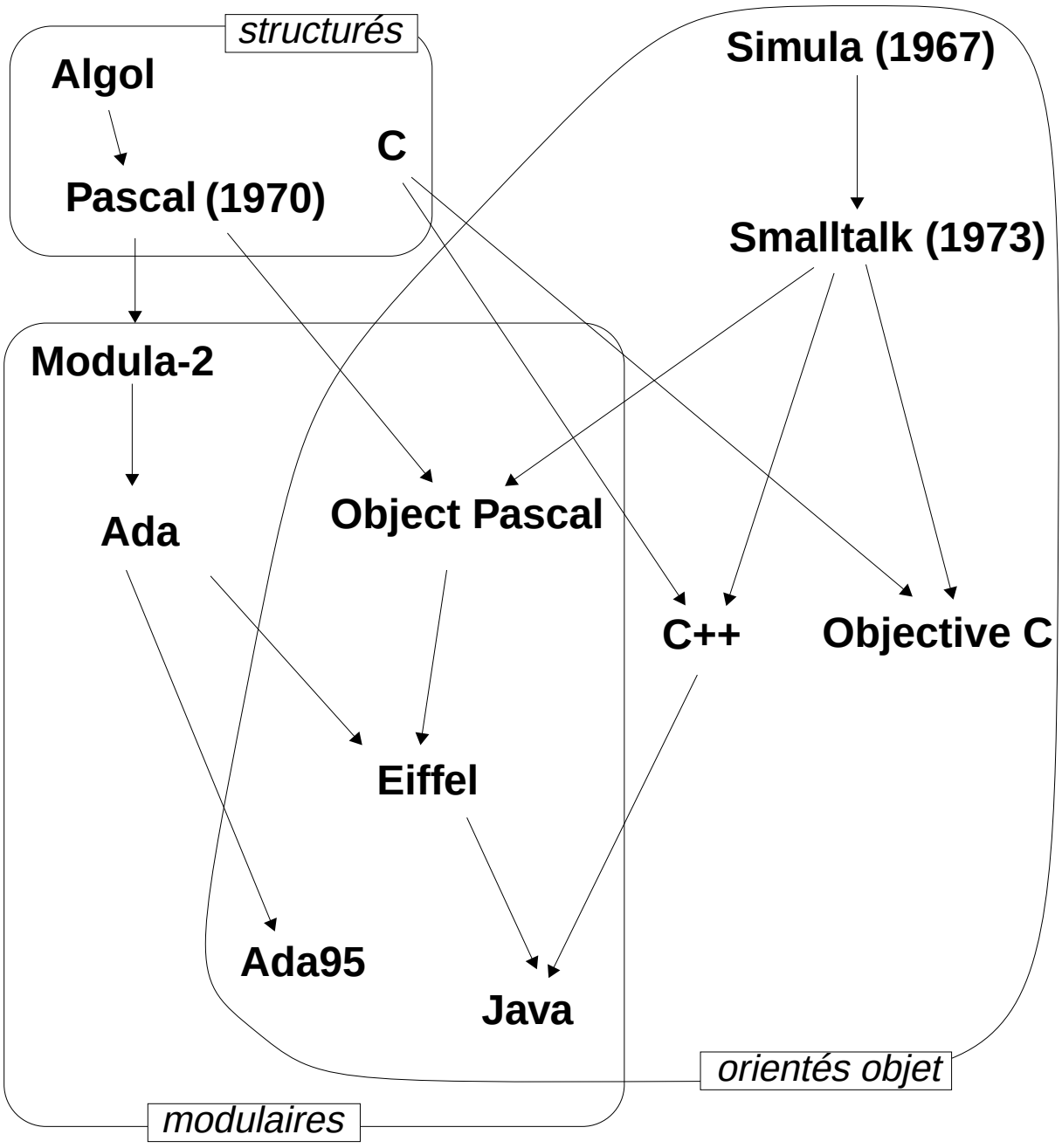
Objets et Programmation

- origine des langages orientés-objet
- modularité, encapsulation
- objets, classes, messages
- exemples en Java
- héritage, liaison dynamique

Origine des langages orientés objet



Modularité et Encapsulation

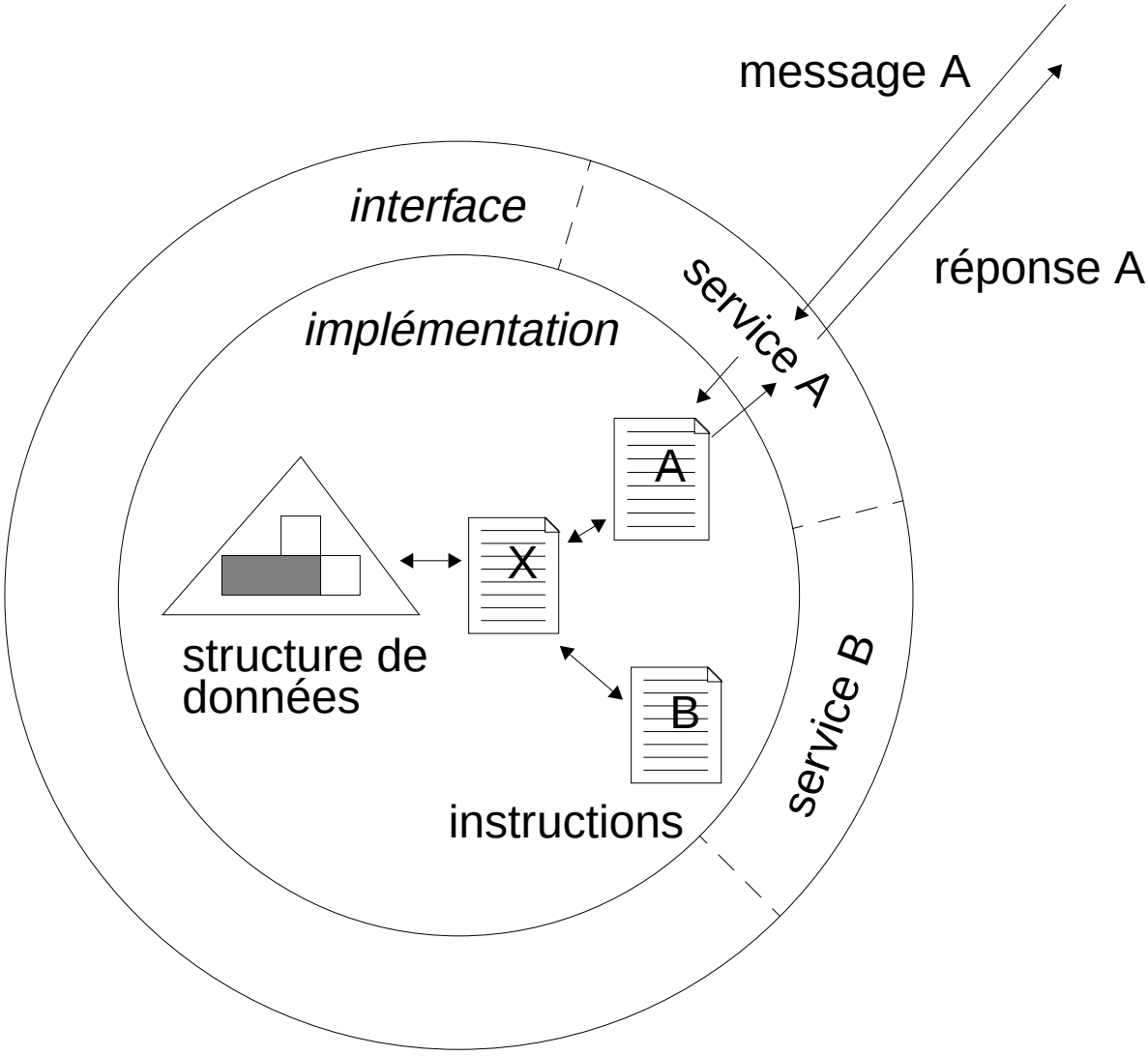
Modularité

- technique de *décomposition* de systèmes
- *réduire la complexité* d'un système par un assemblage de sous-systèmes plus simples
- réussite de la technique dépend du degré d'*indépendance* entre les sous-systèmes (ou degré de *couplage*)
- on appelle *module*, un sous-système dont le couplage avec les autres est relativement faible par rapport au couplage de ses propres parties

Encapsulation

- technique pour *favoriser la modularité* (l'indépendance) des sous-systèmes
- séparer l'interface d'un module de son implémentation,
- *interface* (partie publique): liste des services offerts (quoi)
- *implémentation* (partie privée): réalisation des services (comment)
  - structure de données
  - instructions et algorithmes
- protection des données par des règles (dans les instructions):
  les modules communiquent par *messages*, pas par accès aux données

Un module



(d'après G. Falquet 1997)

Types abstraits

Type “classique”

- ensembles de valeurs possibles

Type abstrait

- ensemble d’opérations applicables

TA rectangle

-- Opération de construction:

```
rect: entier X, entier Y, entier L, entier H --> rectangle;
translation: rectangle R, entier DX, entier DY--> rectangle;
agrandissement: rectangle R, entier DL, entier DH --> rectangle.
```

-- Opération d’accès

```
gauche: rectangle R --> entier;
haut: rectangle R --> entier;
largeur: rectangle R --> entier;
hauter: rectangle R --> entier;
bas: rectangle R --> entier;
droite: rectangle R --> entier.
```

Sémantique => équations

```
gauche(translation(R, DX, DY)) <==> gauche(R) + DX
```

Modules + Types abs. => Langage à objet

Un objet est un module

- privé: variables d'instance, code des méthodes
- public: nom et paramètres des méthodes
- communication: invocation de méthodes, retour de résultats

Classe = générateur d'objet

- génère des objets de même structure
- localise la définition de structure et action des objets
- définit la visibilité: `private`, `public`, etc.

Classe --> type abstrait

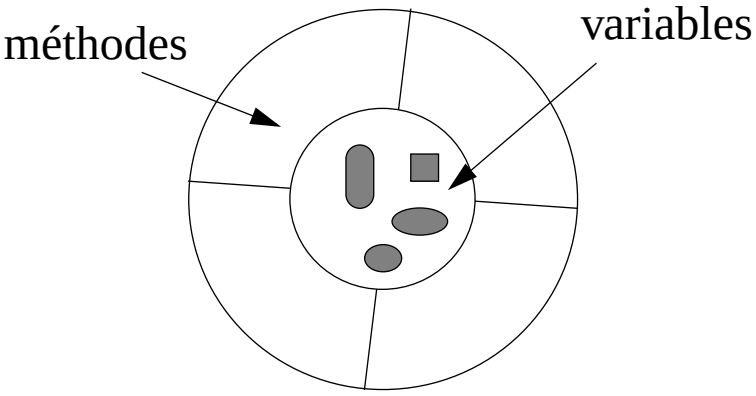
- partie publique --> spécif. (partielle) TA

Classe --> structuration du logiciel

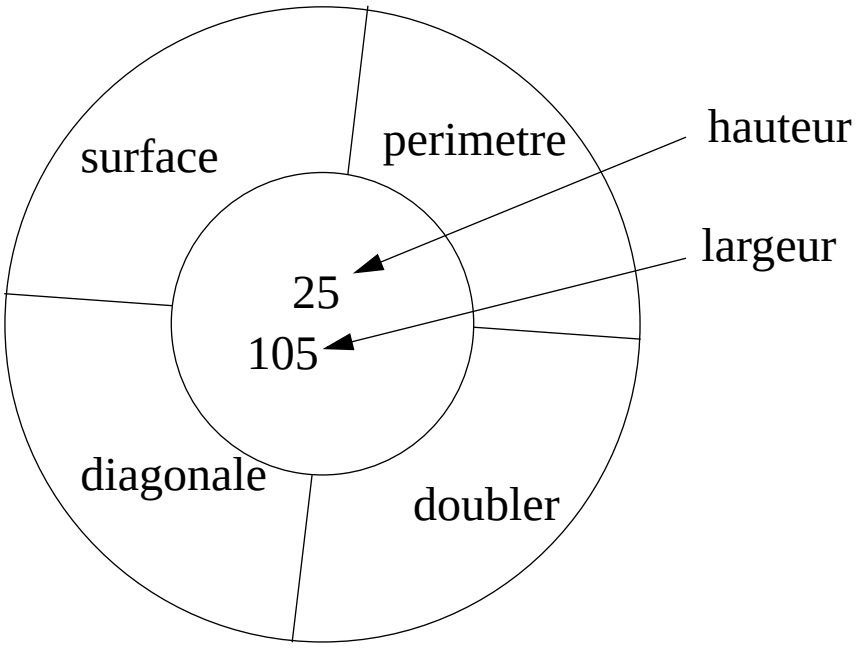
- liens client/serveur
  -> statiques (variables)
  -> dynamiques (méthodes)
- découplage spécification / réalisation
  -> différentes versions de la réalisation

Objet

Une entité contenant des données (état) et des procédures associées (comportement)



Exemple: un objet *rectangle*



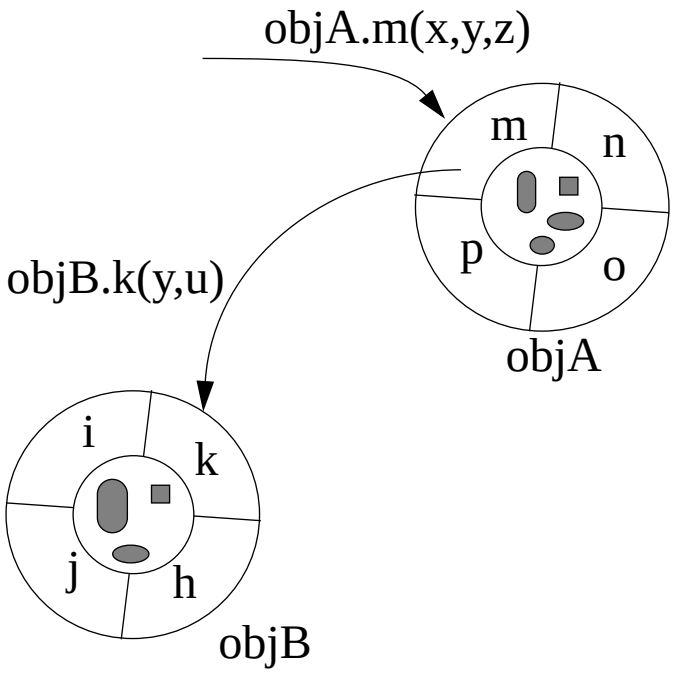
Un objet possède une *identité* unique et invariable.

Messages

Pour utiliser un objet on lui envoie des messages

Un message déclenche l'exécution d'une méthode

Une méthode peut envoyer des messages à d'autres objets



en Java :

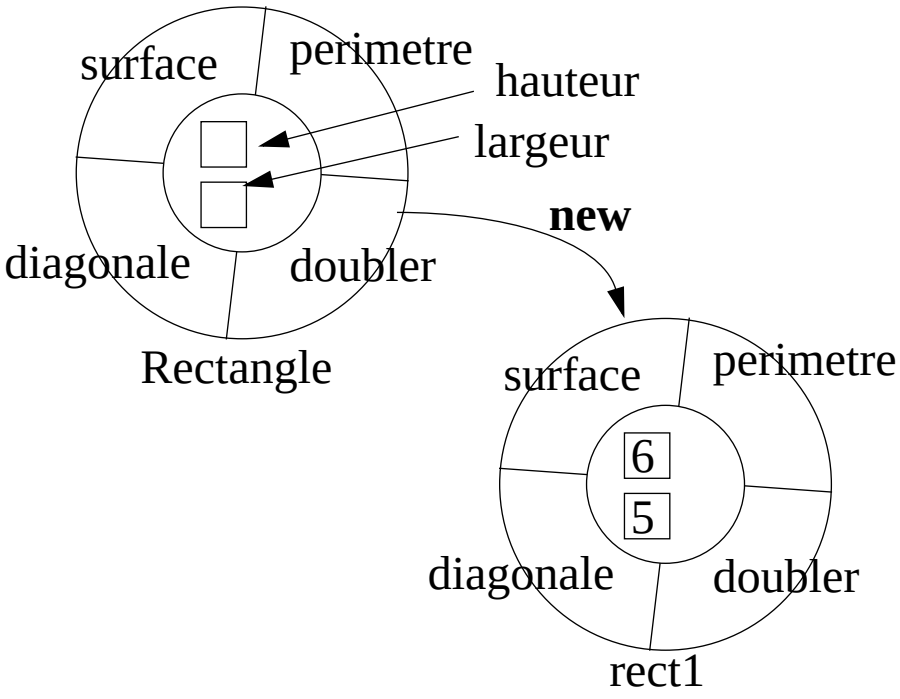
```
rect1.doubler();
d = rect1.diagonale();
System.out.println("Hello");
"Longtemps je me suis levé de bonne heure".size();
uneListe.insertAt(12, "bien");
z = Math.cos(2.45);
```

Un système est constitué d'objets qui communiquent entre eux.

Classes

Une classe est un moule pour fabriquer des objets de même structure et de même comportement.

Un objet est une instance d'une classe



Une classe C définit un type C.

Une variable de type C peut faire référence à un objet de la classe C.

Déclaration d'une classe en Java

```
class Rectangle {
     // variables d'instance
     int largeur, hauteur;

     // constructeur
     Rectangle(int initialL, int initialH) {
          largeur = initialL;
          hauteur = initialH;
     }
     // méthodes
     int perimetre() {
          return 2 * (largeur+hauteur);
     }
     int surface() {
          return largeur*hauteur;
     }
     void retaille(double facteur) {
          largeur = (int)(largeur*facteur);
          hauteur = (int)(hauteur*facteur);
     }
}
```

Utilisation d’une classe Java

Déclaration de variables

```
Rectangle r1, r2;
```

Création d’objets (instantiation)

```
r1 = new Rectangle(50, 100);
r2 = new Rectangle(32, 150);
```

Envoi de messages (utilisation)

```
r2.retaille(1.25);
System.out.println( r2.perimetre() );
```

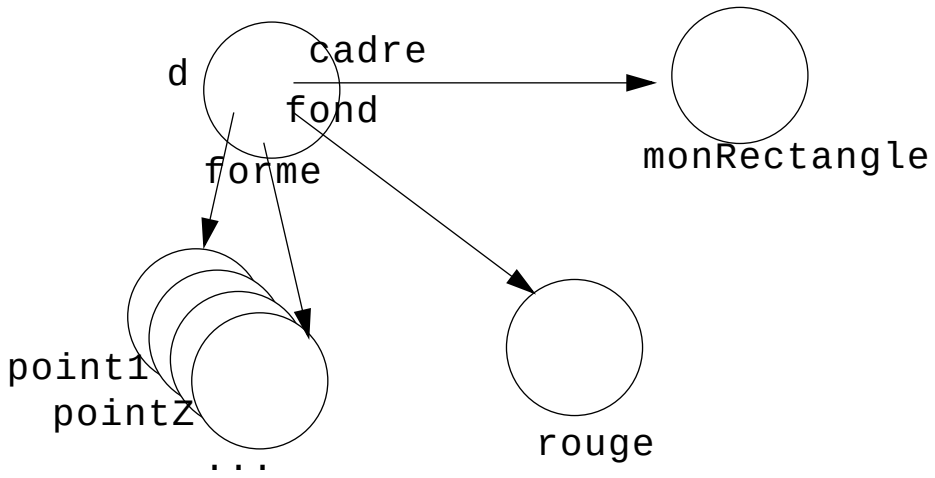
Références aux objets

Une variable de type C fait *référence* à une objet de la classe C.

```
Rectangle ra, rb;
ra = new Rectangle(32, 125);
rb = ra // les deux variables font référence au même
obj.
```

Les variables d'instance établissent des relations entre objets.

```
class Dessin {
     Rectangle cadre;
     Couleur fond;
     Point[] forme;
}
...
d = new Dessin();
d.cadre = monRectangle; d.fond = rouge;
d.form[0] = point1; d.forme[1] = pointZ; ...
```



Identité et Egalité

Une des difficultés de la programmation O-O:

- distinction objet / valeur ==> identité ≠ égalité

```
r1 = new Rectangle(128, 256);
r2 = new Rectangle(256, 128);
r3 = new Rectangle(128, 256);

(r1 == r2) ==> false
(r1 == r3) ==> false

r1.equals(r2) ==> false
r1.equals(r3) ==> true.
```

Test d'égalité

- (simple) comparaison des variables d'instance
- (complexe) calcul sur les variables d'instance

```
class Fraction {
     private int numerateur, denominateur;
     public boolean equals(Fraction f) {
          return (this.numerateur * f.denominateur =
                 this.denominateur * f.numerateur); }
```

Structures de données complexes

Tableaux : dans le langage

```
Point[] monPolygone = new Point[12];
```

Tuples / Records / Struct : classe avec variables d'instance

```
class Adresse { int no; String rue, ville, npostal;}
```

Ensembles/Listes/Piles/Files : classes de l'environnement

```
Vector participants = new Vector();
participants.addElement(joeCool);
s = participants.size();
```

Fonctions/Dictionnaires : classes de l'environnement

```
HashTable localisation = new HashTable();
localisation.put(cnTower, toronto);
localisation.get(tourEiffel);
```

Itérateurs:

```
Enumeration e = localisation.keys();
while (e.hasMoreElements()) {
    k = e.nextElement();
    ...
}
```

Egalité profonde et de surface

```
class ListeRect {
     private Reactangle[] lesRectangles;
     private int nbRectangles;
```

Egalité de surface si

```
a.lesRectangles[0] == b.lesRectangles[0] et
a.lesRectangles[1] == b.lesRectangles[1] et
etc.
```

Egalité profonde (ou récursive)

```
a.lesRectangles[0].equals (b.lesRectangles[0]) et
a.lesRectangles[1].equals (b.lesRectangles[1]) et
etc.
```

Propriétés à maintenir

- Symétrique: a.equals(b) et b.equals(a) doivent toujours donner le même résultat;
- Réflexive: a.equals(a) doit toujour donner true;
- Transitive: si a.equals(b) et b.equals(c) donnent true alors a.equals(c) doit aussi donner true.

La copie d'objets

Même genre de questions que l'égalité

- copie de surface

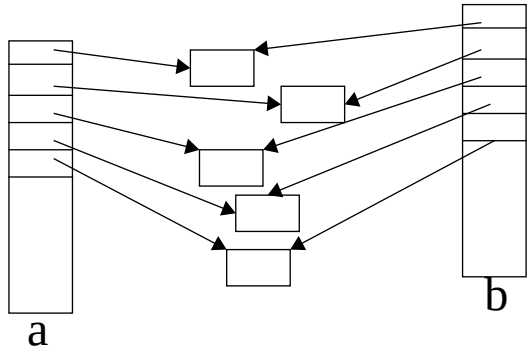
```
for (i=0; i<nbRectangles; i++)
{b.lesRectangles[i] = a.lesRectangles[i]; }
```

- copie profonde

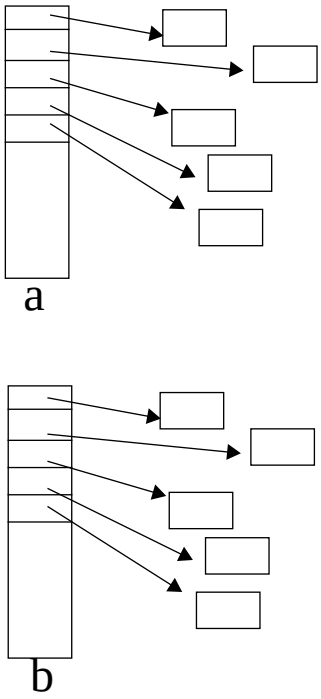
```
for (i=0; i<nbRectangles; i++)
{b.lesRectangles[i] = a.lesRectangles[i].clone(); }
```

La copie de surface introduit du partage de structure !

copie de surface



copie profonde



Sous-classes

- Technique de réutilisation

```
class Point {
     int x, y;
     public float distanceOrigine() {...}
     public float distance(Point pt) {...}
}
class PointTopographique extends Point {
     //hérité: int x, y;
     //hérité: public float distanceOrigine() {...}
     //hérité: public float distance(Point pt) {...}
     int alt;
     public int altitude() {...}
     public void ajusterAltitude(int a) {...}
}
```

- Réutilisation souple (≠ tout ou rien)

On peut redéfinir des méthodes

```
     public float distance(Point p) {...}
```

ou les surcharger

```
     public float distance(PointTopographique p)
{...}
```

=> technique de résolution des message (late binding)

Suppression des tests de typage

```
class Piece {
     String nom;
     boolean estComposee; //la pièce est-elle
composée ?
     int poids;
     int nbComposantes;
     Piece [] composantes;
   ...
   int poidsTotal() {
     if (estComposee) {
          int pt = 0;
          for (int i = 0; i < nbComposantes; i++)
               pt += composantes[i].poidsTotal();
          return pt;
     } else return poids;
   }
   ...
```

* Variables inutilisées

* Code avec des tests de cas

Taxonomie et factorisation

```
abstract class Piece {
     String nom;   // FACTORISATION
     abstract int poidsTotal();
     void affiche() {   // FACTORISATION
          System.out.println("nom:" + nom + " poids:
"
                              + this.poidsTotal());
     }
}
class PieceSimple extends Piece {
     int poids;
     int poidsTotal() {return poids;}
     void affiche() {
          System.out.println("Pièce simple: ");
          super.affiche();
     }
}
class PieceComposee extends Piece {
     int nbComposantes;
     Piece [] composantes;
     int poidsTotal() {
          int pt = 0;
          for (int i = 0; i < nbComposantes; i++)
               pt += composantes[i].poidsTotal();
          return pt;
     }

}
```

Mécanisme de liaison dynamique

Une variable ou un paramètre de type C peut faire référence à un objet de C ou d'une sous-classe de C.

Le choix de la méthode à exécuter se fait dynamiquement lors de l'envoi d'un mesage.

p.poidsTotal()

si *p* fait référence à une *PieceSimple*

exécute *poidsTotal()* de *PieceSimple*.

si *p* fait référence à une *PieceComposée*

exécute *poidsTotal()* de *PieceComposée*.

etc.

La gestion du typage est laissée au système d'exécution.

Construction d’une hiérarchie de classe

Conservation de la sémantique des méthodes

des méthodes de même nom doivent faire “la même chose”

```
class Rectangle {
     double perimetre() {return 2*(largeur+hauteur);}
...
class RectangleMobile {
*    double perimetre() {return largeur*hauteur;} //
!!!
```

Créer des abstractions pour factoriser

```
abstract class Personne {...}
class PersonneMorale extends Personne {...}
class PersonnePhysique extends Personne {...}
```

Eviter les héritages “à l’envers” de la spécialisation

```
class Point {x  y  ...}
* class Carre extends Points {x y c...}
* class Rectangle extends Carre {x y c h...}
```

Un carré n’est pas un point, un rectangle n’est pas un carré !

Un carré est un rectangle particulier, un point est un carré de taille 0.

Interfaces

- Comme une classe abstraite mais sans aucun code de méthodes
- Définissent des propriétés communes transversalement à la hiérarchie des classes
- Hétérarchie d'interfaces (héritage multiple)
- Définissent des types

Une classe peut *implémenter* une interface

Pour cela elle doit définir toutes les méthodes de l'interface.

```
interface Vendable {
     double prix();
     double rabais();
     void acheter(); }

class PieceSimple extends Piece implements Vendable {
     void affiche(){...}
     double prix(){return 7.5 * poidsTotal();}
     double rabais(){return 0.12 * prix();}
     void acheter(){...}
}
...
Vendable aVendre;
aVendre = maPieceSimple12;
x = aVendre.prix();
```

Outil pour la généricité

```
// l'interface est un contrat: s'engager à être
comparable
interface Comparable {
     boolean inferieur(Object c);
}

// classe capable de traiter tout objet respectant le
// contrat "Comparable".
class Tri {

     static void parEchange(Comparable[] x) {
          Comparable tmp;
          for (int i = 0; i<x.length - 1; i++) {
               for (int j = i+1; j<x.length; j++) {
                    if (x[j].inferieur(x[i])) {
                         tmp = x[i];
                         x[i] = x[j];
                         x[j] = tmp;
                         }
                    }
               }
          }
}
```

```
class MotoTriable   extends Moto
                    implements Comparable {

     public boolean inferieur(Object c)
       { return this.prix() <= ((Moto)c).prix(); }
}

// Et maintenant trions des motos(triables):
public static void main(String[] args) {

     MotoTriable motos[] =
          {new MotoTriable(“Honda”, 900, 18350),
           new MotoTriable(“Kawasaki”, 750, 14500),
           new MotoTriable(“Gagiva”, 650, 28300)};

           Tri.parEchange(motos);
           for (int i = 0; i<motos.length; i++)
              {System.out.println(motos[i].label());};
           }
```