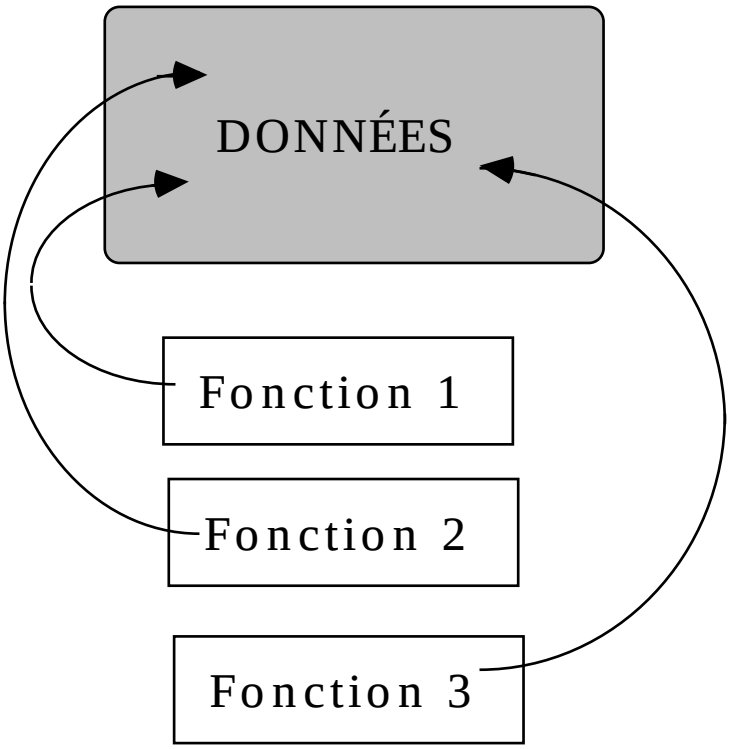
Approche Objet

En programmation procédurale (Algol, ...)

Un programme = suite d'instructions exécutées par une machine.
Son exécution = ces instructions agissent sur des données.



Les fonctions et procédures travaillent "à distance" sur les données.

Accent mis sur les actions. Il s'agit de répondre à la question: Que veut on faire ?

dissociation entre données et fonctions = > problème lorsqu'on change les structures de données.

En programmation procédurale (suite)

Les procédures s'appellent entre elles et peuvent modifier les mêmes données => problème lorsqu'on veut modifier une procédure: comment avait elle été appelée ?

Finalement conception plat de spaghettis dans les appels de procédures. Il serait bien de "responsabiliser" nos parties de programmes

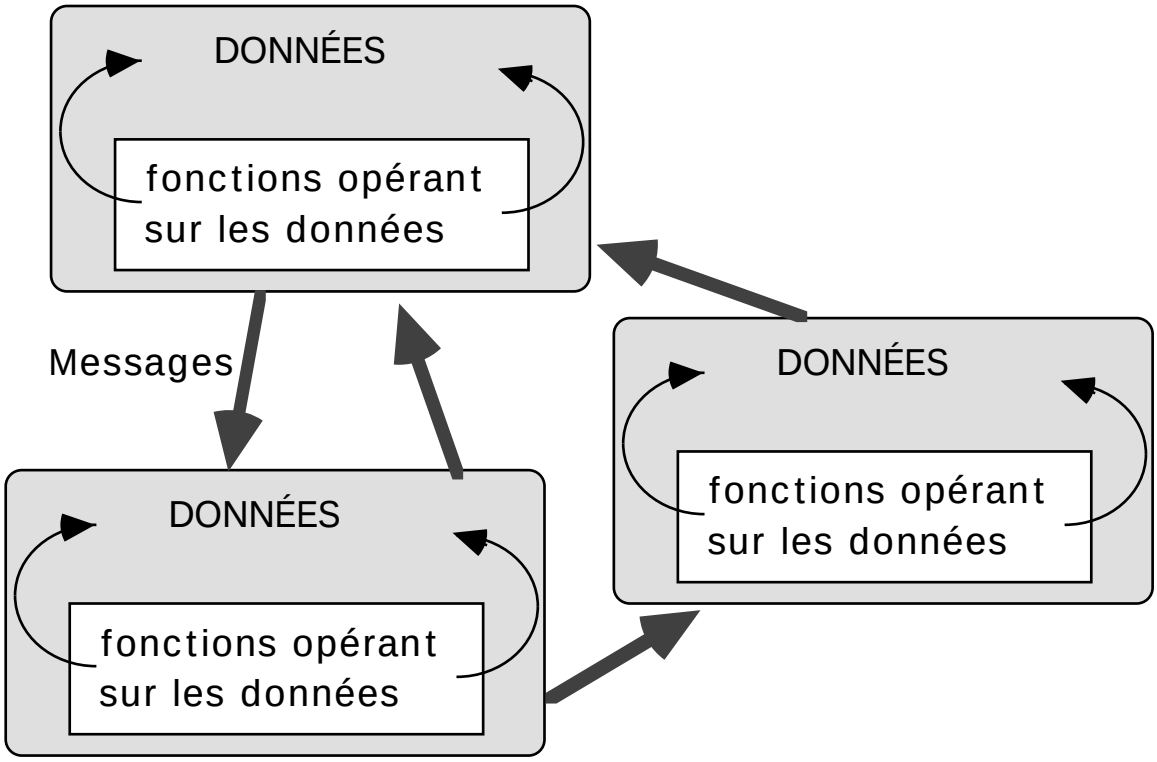
D'où une autre vision de la programmation

Programmation par objets

Un programme = une société d'entités

Son exécution : les entités collaborent pour résoudre le problème final en s'envoyant des messages.

une entité = un objet qui prend en compte sa propre gestion (objet responsable)



liaison inévitable entre données et procédures opérant sur ces données.

Les objets

La question est: De quoi parle t - on ?

Quelles sont les entités qui interviennent dans mon problème ?

exemple :
modéliser un logiciel de trafic routier

les entités sont :
- les feux tricolores
- les carrefours
- les véhicules
- les agents de la circulation.

Lorsqu'un feu tricolore passe au vert il envoie cette connaissance (= ce message) à l'agent posté à ce carrefour. L'agent prend une décision et en informe (envoi de messages) les chauffeurs des véhicules.

Notions manipulées dans le monde des objets

Objets

Encapsulation

Classe

Héritage

Polymorphisme

Objet

objet
=
données + opérations sur ces données
(méthodes)
=
variables de "type abstrait"
=
entité du domaine du problème

Un objet est composé de 2 parties :
- partie interface: opérations qu'on peut faire dessus (partie publique)
- partie interne (intime): données sensibles de l'objet (partie privée)

Les utilisateurs (i.e. les éléments extérieurs) de l'objet ne voient que la partie interface.

Ces entités doivent être indépendantes

Objet (suite)

exemple d'objets :
la renault R21 bleue immatriculée 2245 CDV 75 de mon chef de service est un objet

```
objet R21_de_mon_chef
    genre : Renault
    immatriculation : 2245 CDV 75
    NbPlaces : 5
    propriétaire: chef de service
    s_arreter()
    avancer()
fin objet
```

Un autre objet serait ma clio immatriculée 4357 NBG 93

```
objet ma_Clio
    genre : Renault
    immatriculation : 4357 NBG 93
    NbPlaces : 4
    propriétaire: Moi - même
    s_arreter()
    avancer()
fin objet
```

Encapsulation

Deux sens dans le monde des objets:

Encapsulation = regroupement de code et de données masquage d'information au monde extérieur (data hiding)

Avantages

meilleure modularité
l'unité de modularité est l'objet. Les communications entre modules sont traitées par les opérations d'interface.

meilleure sécurité
a) le code ne peut s'appliquer que sur des types de données bien précis et pas sur d'autres données.
b) certaines parties de l'objet sont inaccessibles pour certains (et n'ont d'ailleurs pas être connues)

meilleure conception dès le début
données et opérations sont spécifiées en même temps.

meilleure lisibilité
données et déclarations des opérations sont écrites au même endroit.

Encapsulation

Avantages

simplicité apparente pour l'utilisateur
I'utilisateur ne connaît que ce qui lui est nécessaire. Il n'a pas connaissance du contenu interne (intime!!) de certaines données contenu qui peut être énorme.
meilleure portabilité
a) les parties masquées pourront être optimisées puis redonnées à l'utilisateur sans que celui ci ne change son code puisque ces parties n'ont pas été utilisées directement
b) Dans ces parties masquées on pourra mettre des points dépendant machines et implémenter ces points pour chaque machine (=> portage facilité: on sait quelle est la partie de code à porter)
vision homogène des objets
Quel que soit l'environnement, l'utilisateur a une même vision des choses.

Classe

classe = modèle décrivant le contenu et le comportement des futurs objets de la classe = ensemble d'objets

le contenu = les données
le comportement = les méthodes

Exemple:
la classe des véhicules, la classe des camions, des automobiles.
La classe des automobiles peut être décrite par

```
classe Automobile
    genre
    immatriculation
    NbPlaces
    propriétaire
    s_arreter()
    avancer()
fin classe
```

Un résumé : classe, objet, méthode et message.

Un exemple particulier d'une classe s'appelle une instance de la classe ou un objet de cette classe :

objet = instance de classe

En première approche, les objets sont à la programmation objet ce que sont les variables à la programmation procédurale. Les classes sont à la programmation objet ce que les types sont à la programmation procédurale.

Programmation procédurale	VARIABLE	TYPE
Programmation Orientée Objet	OBJET	CLASSE

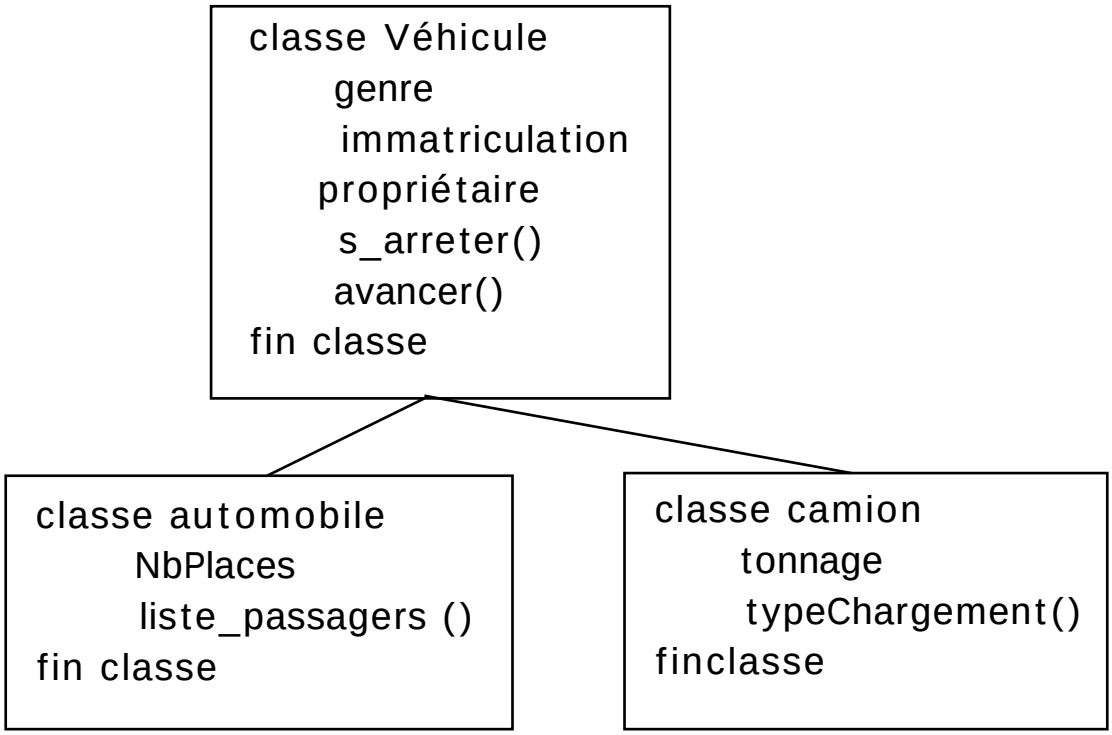
Envoyer un message à un objet c'est lui demander d'exécuter une de ses méthodes.

Héritage

Notion rattachée aux classes

Héritage = construire une classe à partir d'une (d') autre(s)

héritage simple



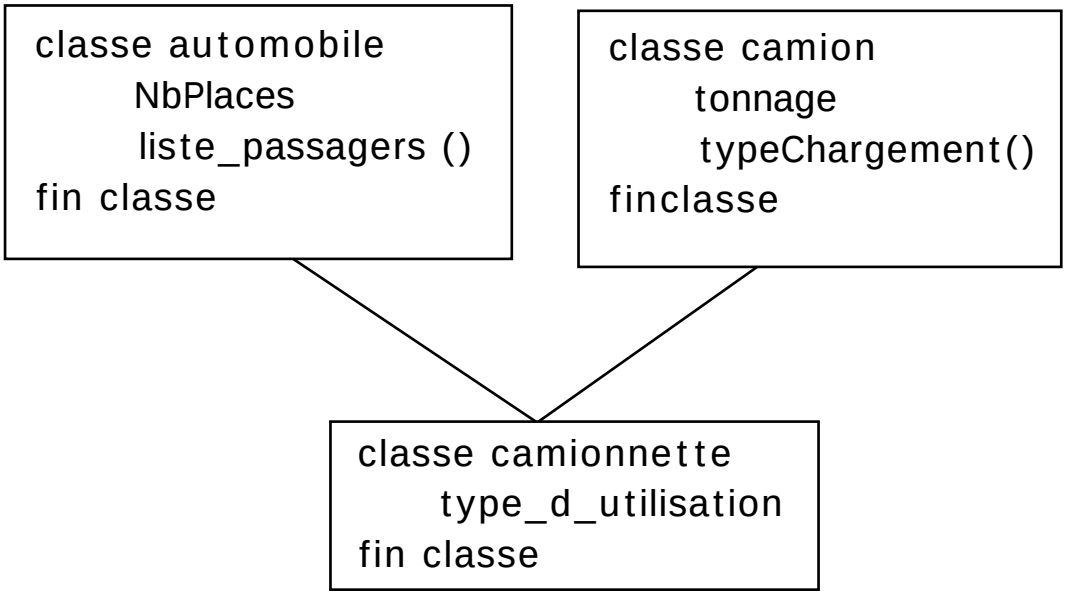
automobile et camion hérite (ou dérive) de Véhicule.

héritage = dérivation

La classe dont on dérive est dite classe de base. Les classes obtenues par dérivation sont dites classes dérivées.

Héritage (suite)

héritage multiple



Héritage (fin)

L'héritage est la possibilité de pouvoir reprendre intégralement tout ce qui a déjà été fait et de pouvoir l'enrichir : vision descendante.
L'héritage est la possibilité de regrouper en un seul endroit ce qui est commun à plusieurs : les modifications des éléments communs ne se font qu'à un seul endroit : vision ascendante.

Il s'utilise dans "les deux sens":
vers le haut
surtout lors de l'analyse O.O: on regroupe dans une classe ce qui est commun à plusieurs classes.
exemple:
dans la classe véhicule on regroupe les caractéristiques communes aux camions et aux automobiles
vers le bas
surtout lors de la réutilisabilité.
La classe véhicule étant définie, on peut la reprendre intégralement pour construire la classe bicyclette

Classe abstraite, classe concrète

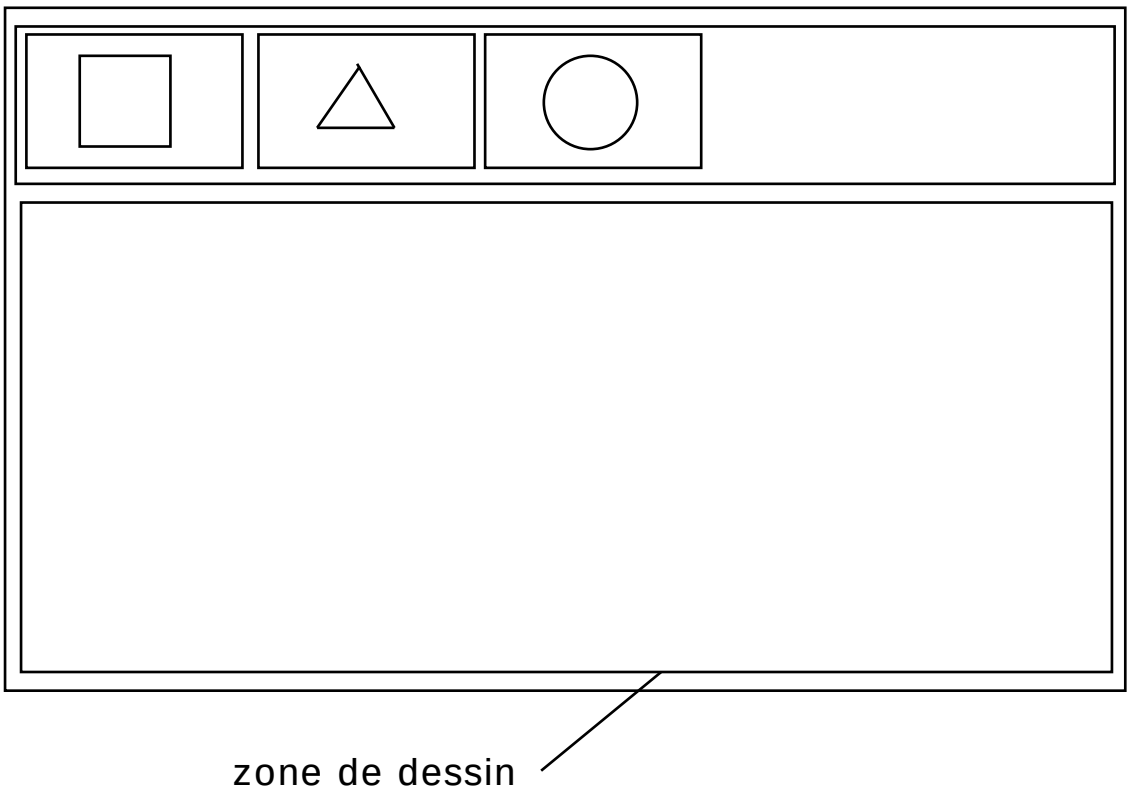
Pour les besoins d'une bonne analyse, on est amené parfois à créer des classes qui finalement ne donneront jamais d'objet.
Ces classes sont dites classes abstraites.

Par opposition les classes à partir desquelles on fabrique des objets sont dites classes concrètes.

Certains langages O.O renforcent cette notion de classe abstraite : le compilateur vérifie qu'on ne crée jamais d'objets de ces classes.

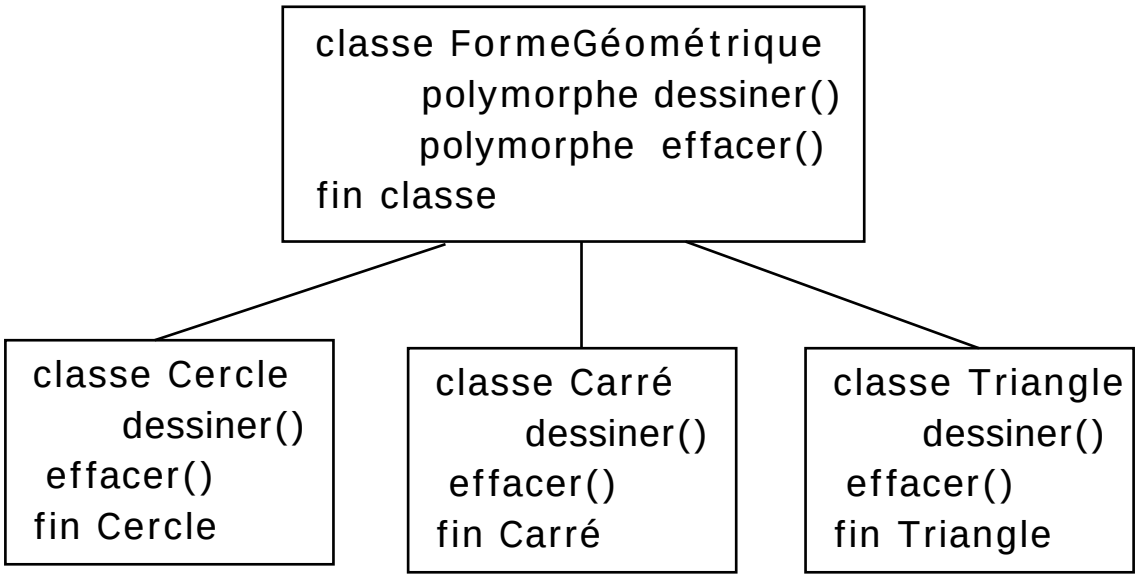
Polymorphisme

On veut créer un paint :



On fait une analyse O.O du problème. On dégage les classes `Cercle`, `Carré`, `Triangle`. Ces classes ont des points communs qu'on regroupe dans la classe `FormeGéométrique`.

Polymorphisme (suite)



`dessiner()` et `effacer()` sont polymorphes. Leur nom est similaire dans les classes `Carré`, `Cercle` et `Triangle` mais dessiner un cercle != dessiner un triangle => plusieurs formes.

Polymorphisme = un même nom, plusieurs implantations

Certains langages demandent de préciser si une fonction doit être polymorphe ou pas (C++, ...). On le précise alors dans la classe de base.

D'autres langages implantent systématiquement le polymorphisme (Smalltalk, Java, ...).

Polymorphisme (suite)

Si on a un ensemble de formes géométriques et qu'on veuille écrire: `rafraichir()`

```
Pour toute forme dans zone de dessin
        effacer()
Pour toute forme dans zone de dessin
        dessiner()
```

A chaque fois il faut appeler la fonction `dessiner()` ou `effacer()` associée à la forme géométrique repérée. Ce choix ne peut être fait qu'à l'EXÉCUTION. Donc

Polymorphisme = liaison dynamique

remarque
A l'aide du polymorphisme, la détermination de la bonne fonction `effacer()` et `dessiner()` est faite automatiquement au moment de l'EXÉCUTION. On ne s'en soucie pas au moment du codage.

Polymorphisme (suite)

Comparaison programmation procédurale vs P.O.O

En programmation procédurale on aurait écrit :

```
dessiner(type_dessin)
{
    switch (type_dessin)
    case CARRE:
    •••
    break ;
    case CERCLE:
    •••
    break ;
    case TRIANGLE:
    •••
    break ;
}
```

En programmation orientée objet on écrit :

```
Classe CARRE {
    dessiner();
}

Classe CERCLE {
    dessiner();
}

Classe TRIANGLE {
    dessiner();
}
```

même libellé de la fonction dessiner()

"le switch est le goto de la P.O.O"

Polymorphisme (fin)

Avantages

Les fonctions ayant la même sémantique ont même nom.

Programmation plus souple : si on veut ajouter une classe `Rectangle`, il suffit de le faire (!!) et d'implanter la méthode `dessiner()` dans cette classe. En programmation procédurale, il faut reprendre le code de `dessiner()` (encore faut il l'avoir !!) et l'enrichir (sans le détériorer !!).

Au moment d'écrire `rafraichir()`, le programmeur n'a pas à connaître tous les types d'objets : l'adjonction d'une forme géométrique (`Rectangle`) se fait sans modification de la fonction `rafraichir()` i.e. la fonction `rafraichir()` fonctionne même sur du code qui sera implanté plus tard !!

Programmation Orientée Objet

Programmation O.O = programmation dans laquelle les programmes sont organisés comme des ensembles d'objets coopérants. Chaque objet représente une instance d'une classe.
Les classes appartiennent à une hiérarchie suivant la relation d'héritage

Remarques sur les langages O.O

"Un langage est orienté objet s'il possède les mécanismes supportant le style de programmation O.O i.e. s'il procure les facilités qui rendent pratique l'usage de ce style. Un langage ne supporte pas une style s'il faut un effort où une adresse exceptionnelle pour écrire des programmes dans ce style (Bjarne Stroustrup)

exemple:
on peut écrire des programmes O.O en COBOL ou assembleur mais ces langages ne supportent (i.e. n'aident pas à) ce style.
Smalltalk, C++, Java sont des langages O.O

Analyse et Conception Orientée Objet

Dijkstra: "la technique à appliquer pour maîtriser la complexité du logiciel est connue depuis très longtemps : divide et impera (diviser pour régner)"

Descartes (Le discours de la méthode) : "Diviser chacune des difficultés que j'examinerai en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre"

Donc décomposer en parties de plus en plus petites chacune d'elles pouvant être affinées. Pour comprendre un niveau donné d'un système il ne faut avoir qu'à appréhender quelques parties (et non pas toutes) du système.

Décomposition algorithmique :
L'accent est mis sur les actions à effectuer pour résoudre le problème

L'analyse orientée objet est une méthode d'analyse qui examine les besoins d'après la perspective des classes et objets trouvés dans le vocabulaire du domaine du problème

Notions "est une" et "a une" : Héritage ou Inclusion

Le problème : lorsqu'une classe est créée, faut il la réutiliser en fabriquant un objet de cette classe à l'intérieur d'une autre classe (inclusion) ou en hériter ?

Théorèmes

Lorsqu'une nouvelle classe est une spécialisation d'une autre classe, est "une sorte" d'autre classe, on hérite : notion "est une"

Lorsqu'une classe possède un élément d'une autre classe, on crée un champ de cette autre classe comme composante de la nouvelle classe créée. On inclue : notion "a une".

Exemple

Une voiture comme un camion est un véhicule => les classes voiture et camion hérite de véhicule.

Un véhicule possède un châssis => les objets de la classe véhicule ont un champ châssis.

Un processus d'analyse orientée objet

Dans cet ordre :
1°) répertorier les entités du domaine du problème et leur comportement

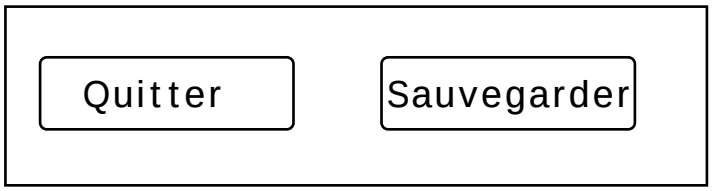
2°) En déduire les classes auxquelles ces entités appartiennent.

3°) Architecturer l'ensemble des classes en regroupant les données ou procédures communes à certaines classes dans des classes dont on hérite.

Notions O.O et interfaces graphiques

Les notions d'objets, de classes, d'héritage sont grandement utilisées.

Dans une interface, on dispose de "rectangles" "sensibles" et donc vont exécuter du code = objets.
exemple : 2 boutons poussoir



On définit donc des classes d'objets graphiques
exemple : la classe des bouton poussoir

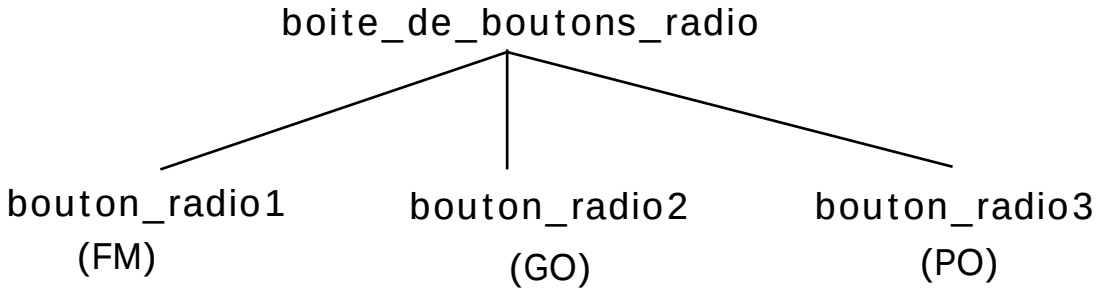
```
classe BoutonPoussoir
largeur, hauteur
code_lors_d_un_clic()
fin classe
```

Notions O.O et interfaces graphiques (suite)

disposition des objets et envoi de messages

Les objets sont rangés dans d'autres.
exemple :

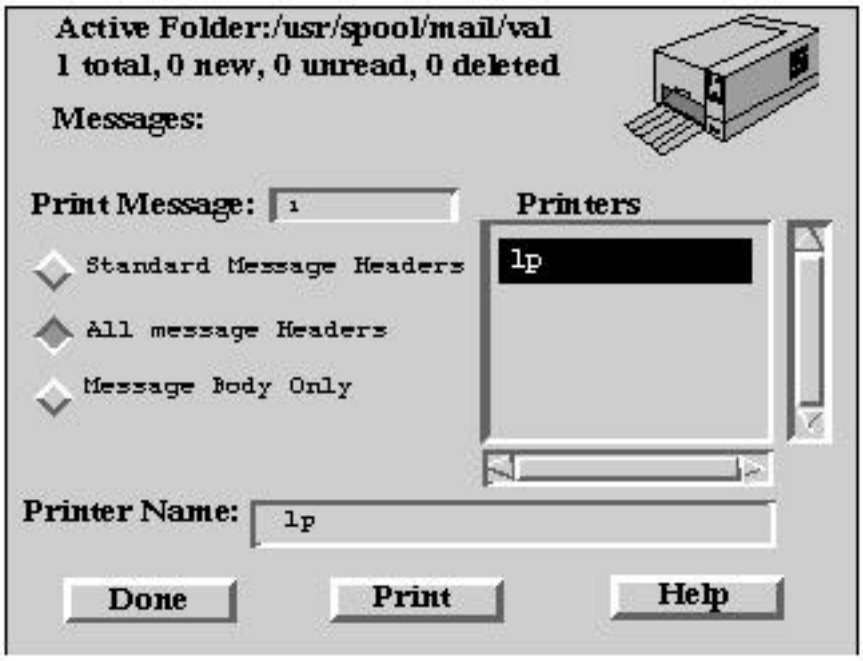
Cette représentation graphique est notée :



l'objet graphique `boite_de_bouton_radio` contient trois autres objets `bouton_diamant`. Cette `boite_de_bouton_radio` les gère de sorte à ne pouvoir en sélectionner qu'un seul à la fois => passage de renseignements d'un objet contenant à ses objets contenus : envoi de messages
Même idée avec des objets de positionnement (ligne_colonne, d'attachement, etc.)

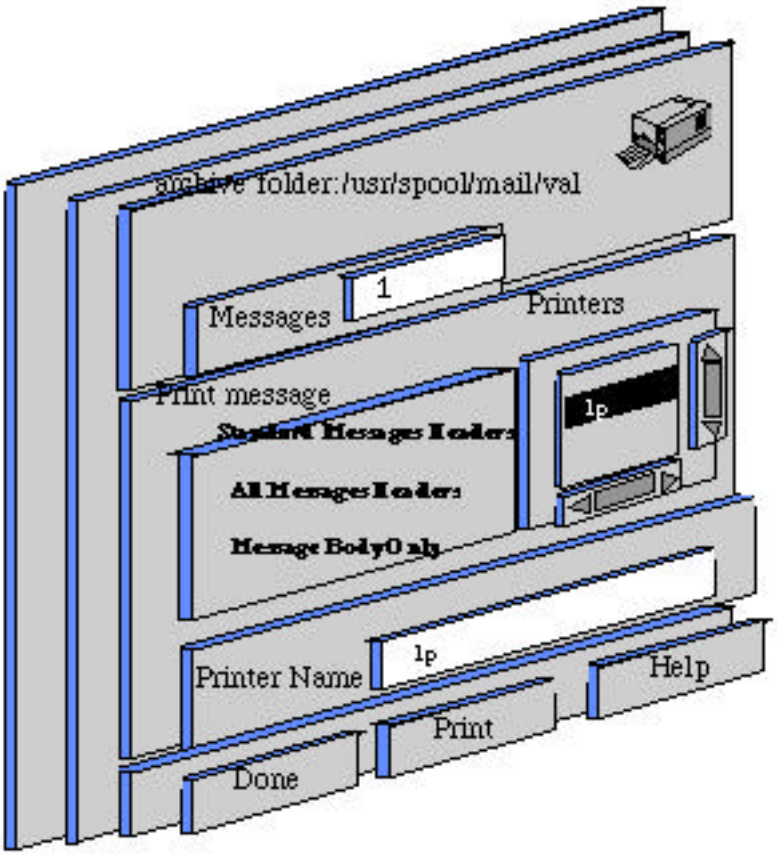
Interfaces graphiques

Quand vous voyez ceci :



C'est qu'on a programmé cela :

Interfaces graphiques (suite)

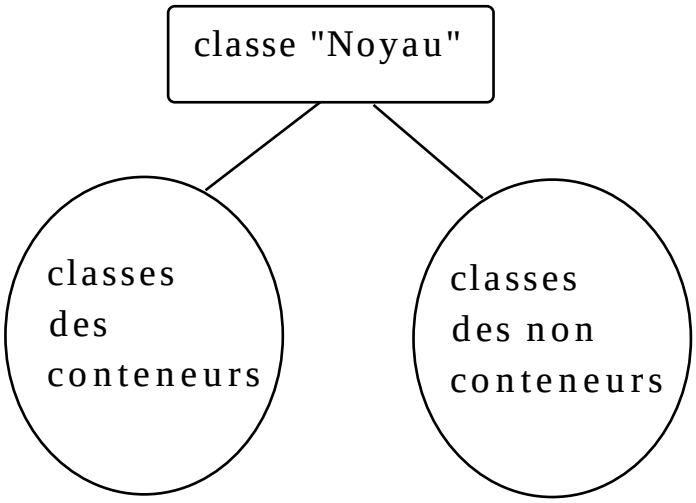


Il existe donc des composants graphiques qui en contiennent d'autres (et gèrent leur apparition, leur positionnement, ...) et d'autres qui n'en contiennent pas comme les boutons poussoirs, les fenêtres textes de saisie, ... Ces derniers sont parfois appelés les contrôles.

Héritage dans les interfaces graphiques

héritage dans les classes

Les classes d'objets graphiques sont donc rangées en arborescence d'héritage comme :



conclusion

Les 2 arbres dessinés ont peu de choses à voir l'un l'autre.

Le premier est l'architecture de l'interface i.e. le placement des divers composants graphiques les uns par rapport aux autres, le second est un arbre d'héritage de classes donné une bonne fois par les distributeurs d'objets graphiques (OSF pour Motif, SUN pour Java)

Bibliographie sur l'orienté objet

Concept généraux et méthodes

Conception et programmation par objets: Jacques Ferber ed Hermès.

Conception orientée objet et applications: Grady Booch ed InterEditions.

http://cedric.cnam.fr/~farinone/Java2810/7.html
jusqu'à
http://cedric.cnam.fr/~farinone/Java2810/16.html