La technologie objet Eiffel

Patrick Bellot[1] Akim Demaille[2]

[1]ENST - École Nationale Supérieure des Télécommunications

[2]EPITA - École Pour l'Informatique et les Techniques Avancées

June 14, 2012

La technologie objet Eiffel

Introduction à Eiffel

- un langage de haut niveau conçu pour le Génie Logiciel, portable, avec une syntaxe originale et claire
- une clarification et une conception *moderne* de l'héritage multiple de classes
- l'essentiel des outils programmatiques de très haut niveau (classes virtuelles, génériques, exceptions, etc.)
- de nombreuses bibliothèques

Les bibliothèques

EiffelCOM (COM,OLE,ActiveX),
EiffelCORBA,
EiffelMath,
EiffelNet (client-serveur),
EiffelLex & EiffelParse,
EiffelStore (BD),
EiffelWEB,
Eiffel DLE (dynamic link),
EiffelVision (GUI),
Graphical Eiffel for Windows, *Eiffel WEL* (Windows),
EiffelThreads,
etc.

Des concepts originaux

clauses d'adaptation d'héritage
: résoudre les "problèmes" de l'héritage multiple ;

programmation contractuelle
: favoriser la réutilisabilité et la modularité ;

interface graphique
: fonctionnement de l'interface, drag and drop, etc.
originales mais pas toujours appréciées.

Un système

Compilateur	à trois modes dont un mode incrémental très utile en phase de développement
EiffelBench	un environnement de développement en accord avec le langage
EiffelBuild	un environnement de développement d'interfaces graphiques
EiffelCase	un environnement de conception d'applications

Objets, Classes, Héritage

Classes et Objets

Les objets

Tous les programmeurs ont en tête la célèbre équation venue de la programmation structurée :
Structures de données + Algorithmes = Programmes
L'objet prolonge cette équation :

Vue Logicielle Un objet est la réunion d'un type de données et des programmes permettant de traiter ces données. L'outil de sa déclaration est la classe.

Vue Logique Un objet est la représentation auto-suffisante d'un concept servant à la résolution d'un problème. La classe est la description d'une catégorie d'objet.

L'objet et l'analyse

- L'analyse d'un problème à traiter en orienté objet est avant tout une analyse orientée données.
- Pour chaque donnée du problème, on décrit ses propriétés (attributs) et ses comportements (routines). Ce peut être des données réelles ou abstraites.
- Puis ces données sont utilisées pour résoudre le problème posé.

$\Rightarrow$ Très différent de la démarche algorithmique.

La classe

La classe est la déclaration (type) d'un objet. Un objet est une instance d'une classe. La classe possède deux composantes :

attributs la composante statique, les données contenues dans l'objet

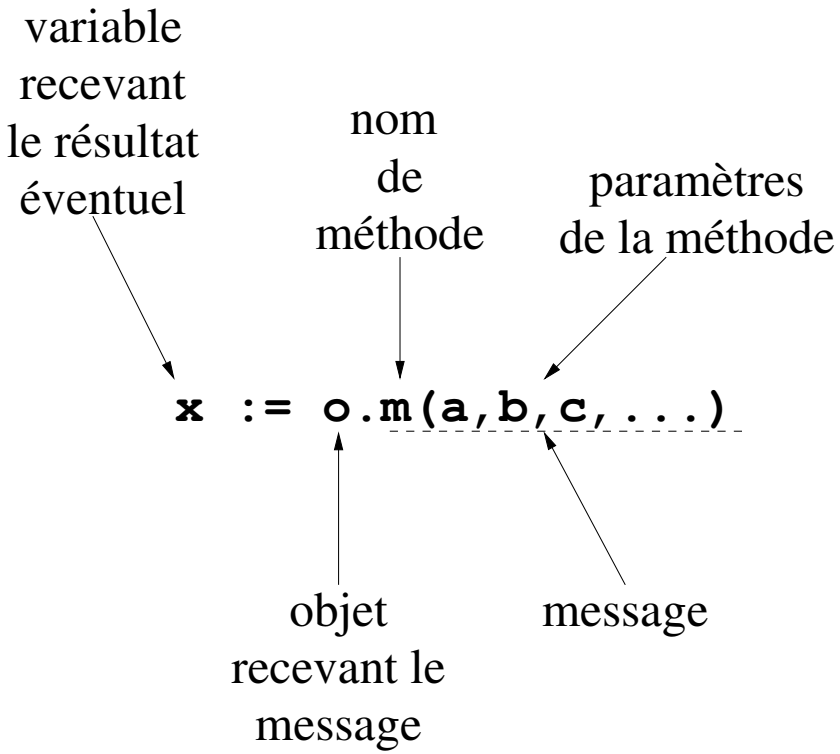
routines la composante dynamique, les procédures ou fonctions attachées à l'objet.

Attributs et routines sont globalement appelés les primitives de la classe dans la terminologie Eiffel.

Un exemple de classe

```
class POINT
  -- un point dans un dessin géométrique

feature

   -- deux attributs : les coordonnées
   xc,yc : INTEGER ;

   -- une méthode : changer les coordonnées
   set_x_y(x,y : INTEGER) is
     do
        xc := x ;
        yc := y ;
     end ;

end -- class POINT
```

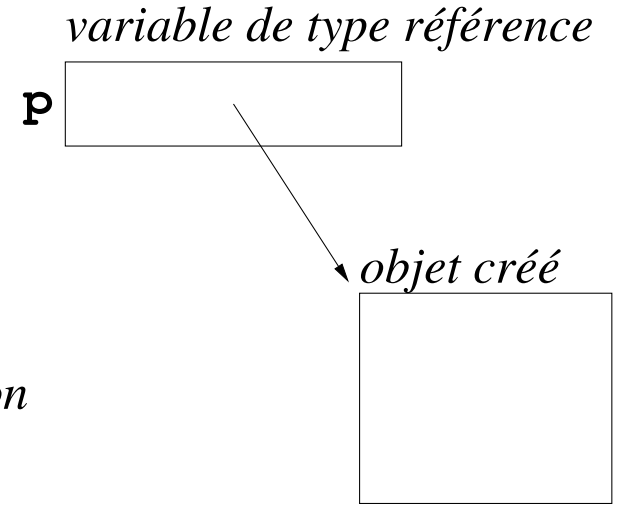
Appel de méthode



- L'objet exécute la méthode `m` qu'il connaît. Il l'exécute dans son propre contexte. C'est toujours un objet qui exécute la méthode.
- Dans le cas d'objets distribués, il s'agit réellement d'un envoi de message. Sinon, c'est un appel de procédure ou de fonction.

Création d'un objet

```
p : POINT ;

do
    ...
    !! p ; -- instruction
    ...
end ;
```



Sauf déclaration explicite, toutes les variables d'objets sont des variables de type référence. Elles contiennent des pointeurs.

Création avec initialisation 1/2

```
class POINT
  -- un point dans un dessin géométrique
create
   make  -- routine d'initialisation
feature
   -- une méthode d'initialisation
   make(x,y : INTEGER) is
     do
        set_x_y(x,y) ;
     end ;

   -- suite inchangée
   -- ...

end -- class POINT
```

- Les attributs sont initialisés avec des valeurs par défaut (e.g., 0 pour un entier, `Void` pour une variable de type référence).
- Si l'on désire initialiser les objets à la création, il faut une routine d'initialisation.

Création avec initialisation 2/2

Puis on crée l'objet en faisant appel à sa routine d'initialisation :

```
p : POINT ;
create p.make(23,64) ;; -- crée un point initialisé
```

- Il peut y avoir plusieurs routines d'initialisation pour une même classe d'objets. C'est à la création que l'on choisit laquelle utiliser.
- Lorsqu'au moins une routine d'initialisation est déclarée, il n'est plus possible de créer un objet sans faire appel à l'une de ces routines.
  $\Rightarrow$ Sécurité

Accès aux attributs

Lecture Par défaut, les attributs sont accessibles en lecture en utilisant la notation pointée : sauf déclaration contraire, tout objet peut connaître la valeur d'un attribut d'un autre objet.

Écriture Les attributs ne sont jamais accessibles en écriture sauf pour l'objet lui-même. Pour les autres objets, il faut que l'objet à modifier fournisse une routine de modification comme la routine `set_x_y(x,y : INTEGER)` de la classe `POINT`.
$\Rightarrow$ Sécurité

Accès aux routines

- Par défaut, les routines, procédures et fonctions, sont accessibles en exécution.
- Pour les routines comme pour les attributs, il est possible d'en restreindre l'accès aux objets de certaines classes.

Restriction des accès

`feature` ou `feature{ANY}`

primitives avec accès par défaut (tous les objets sont de la classe `ANY`)

`feature{A,B,C,...}`

primitives accessibles seulement par les objets des classes `A`, `B`, `C`

`feature{}` ou `feature{NONE}`

primitives inaccessibles (`NONE` : classe sans instance)

Un point

- Une *syntaxe claire* à la Pascal avec des mots clés explicites plutôt que des symboles.
- *Initialisation sécurisée* par les routines de création.
- *Encapsulation et protection* des données (attributs) :
  - interdit en écriture ;
  - modulation des accès par la clause `feature`.

L'héritage de Classes

Classes et modules

- Ainsi présentées, les classes ne sont que des modules un peu évolués. L'*héritage* fait la force logicielle des langages à objets.
- Quand on définit une classe, on peut déclarer qu'elle hérite d'une ou plusieurs autres classes. La classe qui hérite est appelée classe fille ou sous-classe de la classe dont elle hérite. Cette dernière est appelée classe mère ou surclasse.

$\Rightarrow$ La classe fille hérite de toutes les primitives accessibles définies dans la classe mère. Puis elle est spécialisée par le programmeur.

Spécialisation d'une classe

Une fois héritées les primitives accessibles de la classe mère, il existe deux manières non exclusives de spécialiser la classe fille :

l'enrichissement de nouvelles primitives sont ajoutées à la classe fille ; Eiffel parle alors d'enrichissement modulaire ;

la substitution des primitives héritées sont redéfinies et adaptées pour la classe fille ; Eiffel parle alors d'affinage.

Exemple d'héritage 1/2

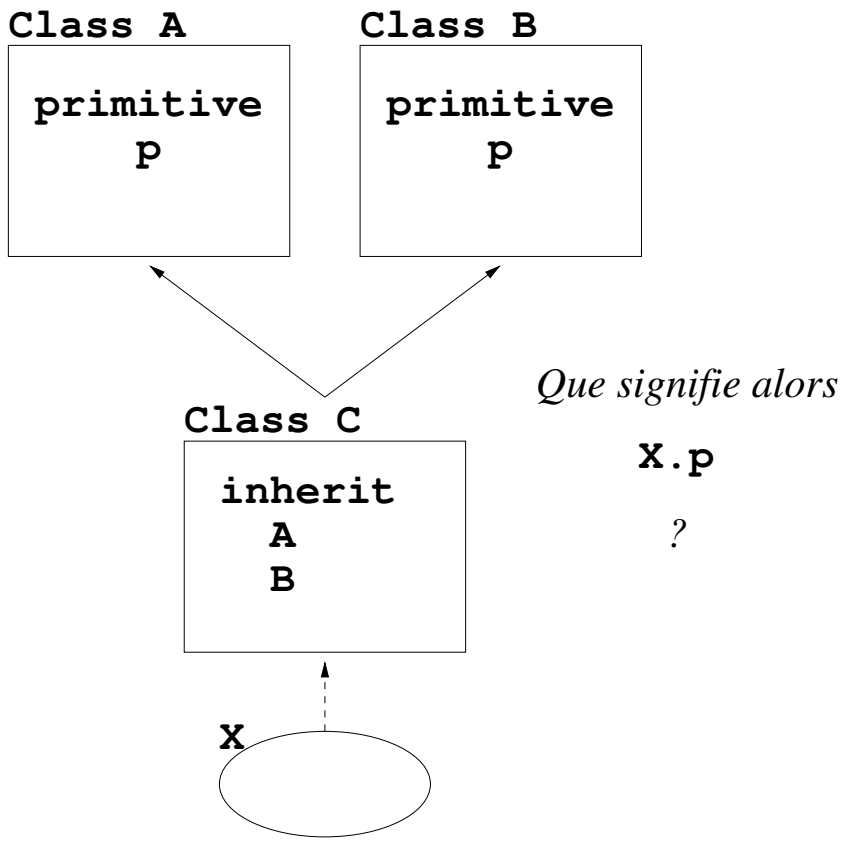
```
class SQUARE
create
   make
feature
   xc,yc : INTEGER ; -- coordonnées
   width    : INTEGER ; -- largeur

   make(x,y,l : INTEGER) is
      do
         xc := x ;
         yc := y ;
         width := l ;
      end ;

   area : INTEGER is
      do
         Result := width*width ;
      end ;
end -- class SQUARE
```

Exemple d'héritage 2/2

```
class RECTANGLE
inherit SQUARE
      rename make as make_square
      redefine area
      end ;
create make
feature
   -- EXTENSION
   height : INTEGER ; -- hauteur
   make(x,y,w,h : INTEGER) is
      do
         make_square(x,y,w) ;
         height := h
      end ;
   -- AFFINAGE
   area : INTEGER is
      do
         Result := width * height ;
      end ;
end -- class RECTANGLE
```

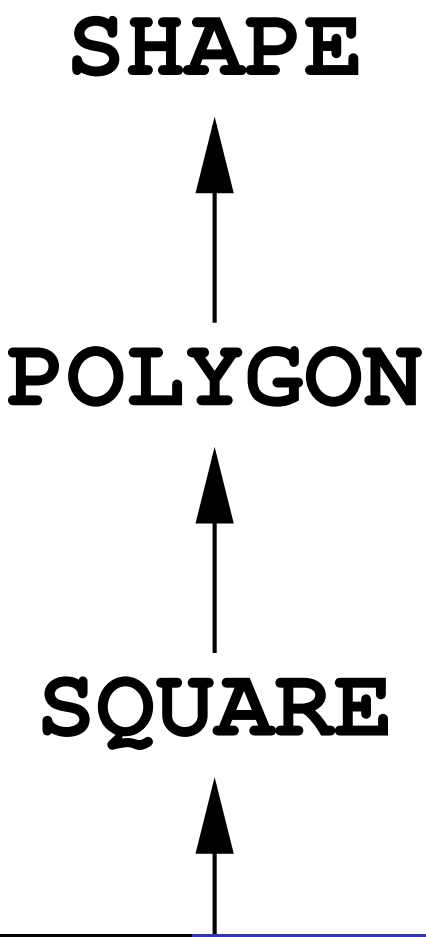
Héritage simple et multiple

- L'héritage est simple si toute classe n'hérite au plus que d'une seule classe. Sinon il est multiple.
- L'héritage multiple est bien plus puissant que l'héritage simple mais il pose des problèmes qu'Eiffel résout grâce à la *clause d'adaptation d'héritage*



Héritage et polymorphisme

Un objet a comme type toute ses surclasses. Supposons l'héritage :

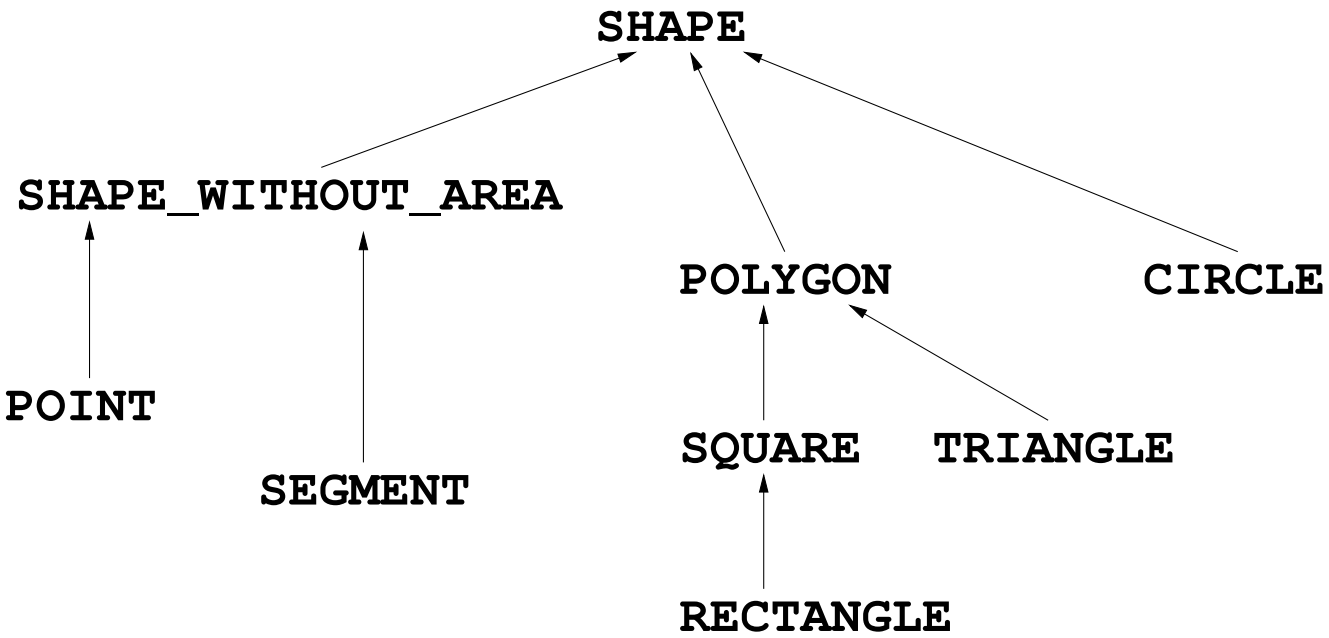


alors les affectations suivantes sont valides :

```
s : SHAPE ;
p : POLYGON ;
q : SQUARE ;
r : RECTANGLE ;

do
   s := p ;
   p := q ;
   q := r ;
end ;
```

Hériter pour modéliser

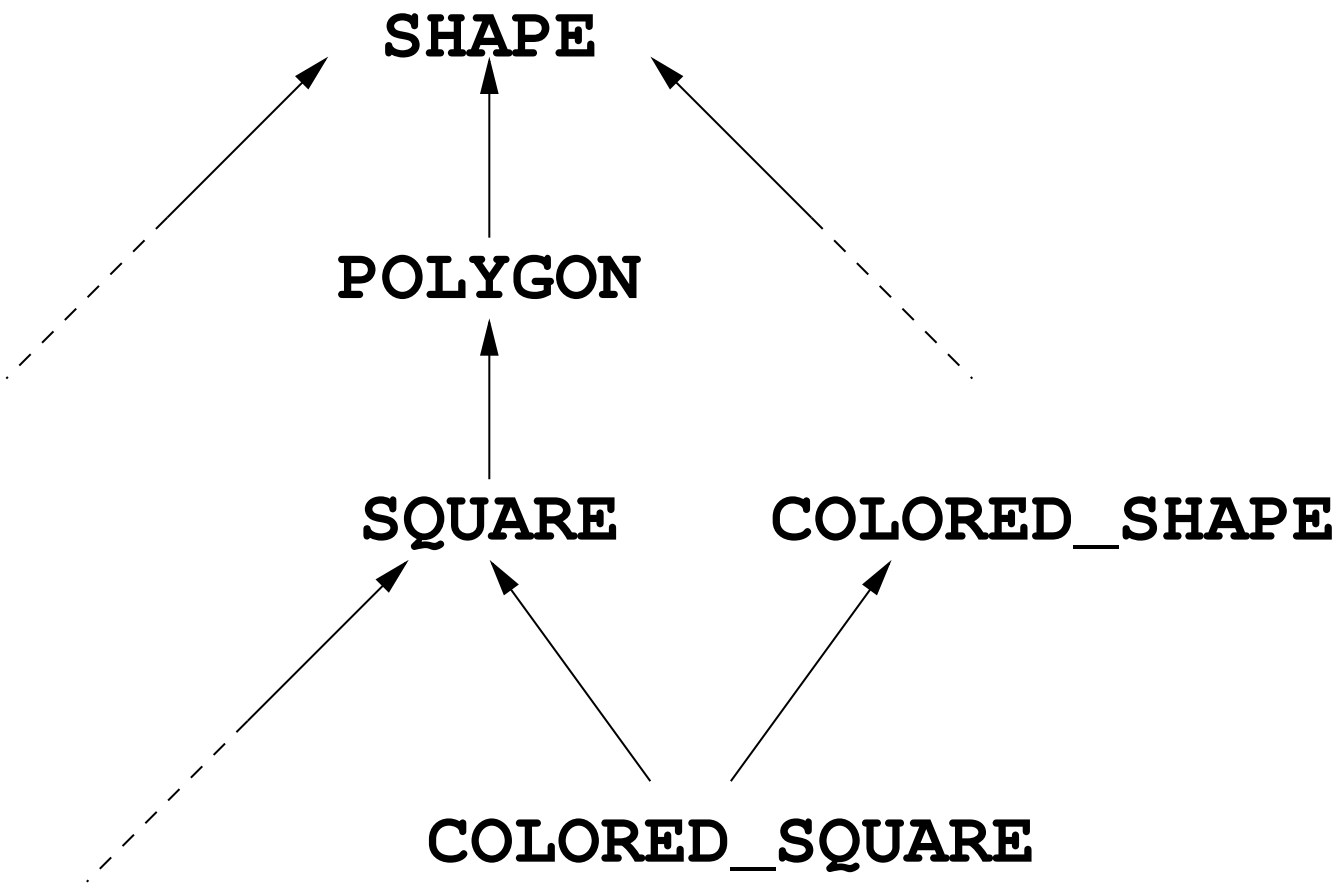
Modéliser et catégoriser les objets du problème que l'on traite informatiquement : des classes générales (classes mères) sont dérivées en classes spécialisées (classes filles) :



Dans cet optique, l'héritage multiple est rarement nécessaire.

Modélisation et héritage multiple

Il reste cependant bien plus pratique de disposer de l'héritage multiple :



Attention ! La modélisation par les classes demande un sérieux

Contre-sens ? 1/2

Est-ce valide :

```
q : SQUARE ;
r : RECTANGLE ;
do
   ...
   q := r ;
   ...
end ;
```

Valide, car RECTANGLE hérite de SQUARE!

Est-ce sain ? Non, car un rectangle n'est pas un carré !

$\Rightarrow$ "hérite de" = "est une sorte de"

alors que nous avons appliqué :

$\Rightarrow$ "hérite de" = "est une extension de"

Contre-sens ? 2/2

```
q : SQUARE ;
r : RECTANGLE ;
do
   ...
   q := r ;
   ...
end ;
```

Et si nous faisons hériter `SQUARE` de `RECTANGLE` ? En appliquant :

$\Rightarrow$ "hérite de ¿¿ = ¡¡ est une spécialisation de"

Ça ne marche pas non plus nous apprend la programmation contractuelle car on peut changer une dimension d'un rectangle sans changer l'autre.

Confondre héritage et composition

Une mauvaise classe SEGMENT

```
class SEGMENT
inherit
   POINT
feature
   extremite : POINT ;
   ...
end -- class SEGMENT
```

Car elle autorise :

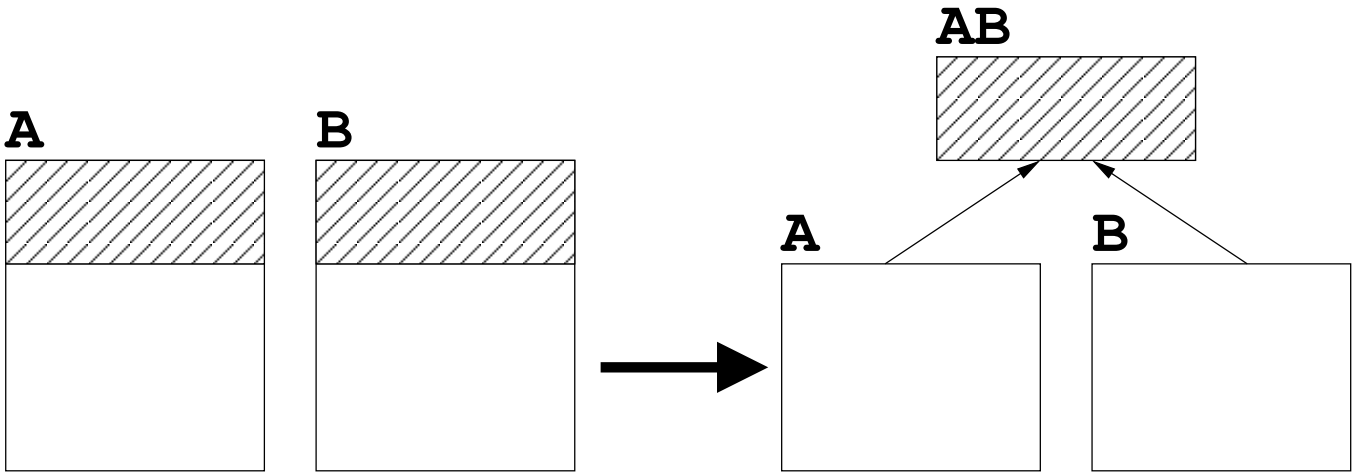
```
p : POINT ;
s : SEGMENT ;
...
   p := s ;
```

La bonne classe SEGMENT :

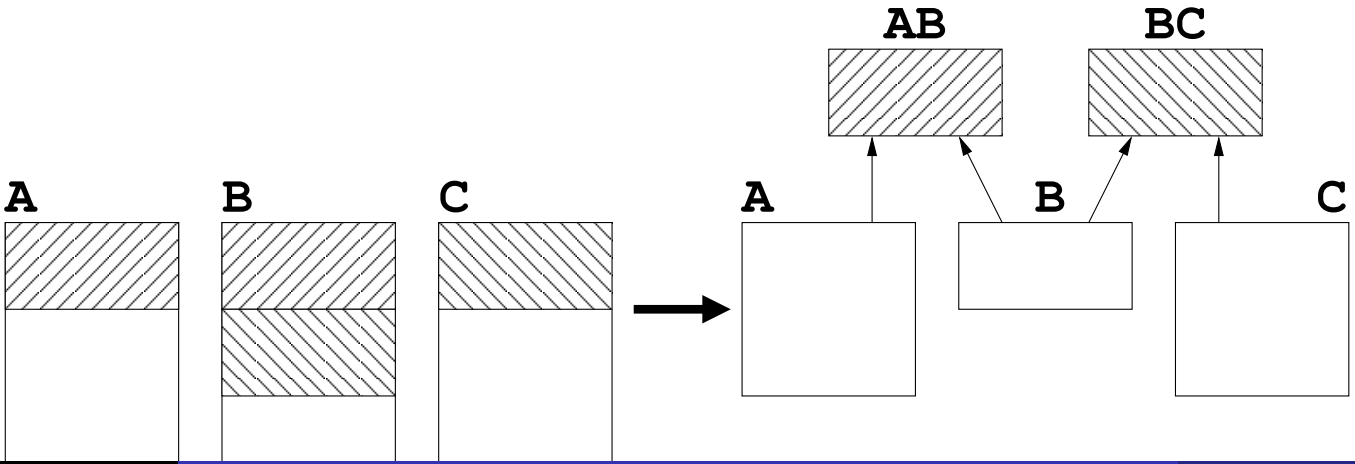
```
class SEGMENT
inherit
   SHAPE_WITHOUT_AREA
feature
   first, second : POINT ;
   ...
end -- class SEGMENT
```

Hériter pour factoriser

L'héritage sert à la factorisation du code.



L'héritage multiple est nécessaire.



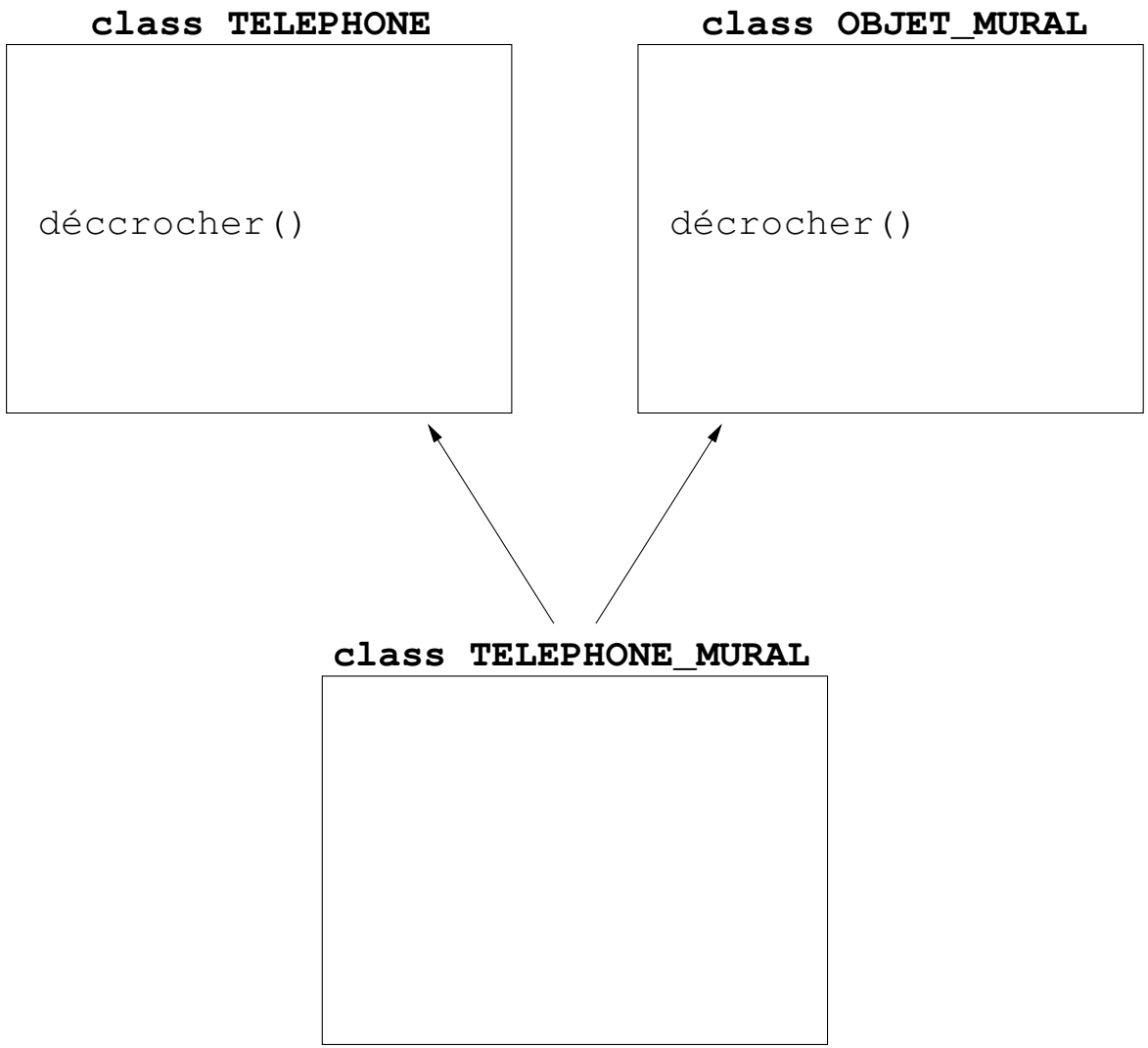
Hériter pour réutiliser

Un formidable outil de réutilisation du code.

```
class BLUE_BUTTON

inherit
   PUSH_B -- EiffelVision
      rename make as old_make end ;
   COLORS -- EiffelVision

create make

feature

   make(t : STRING; w:WINDOW; x,y,dx,dy : INTEGER) is
      do
         old_make(t,w) ;
         set_x_y(x,y)  ;
         set_size(dx,dy) ;
         set_color(blue) ;
      end ;

end -- class BLUE_BUTTON
```

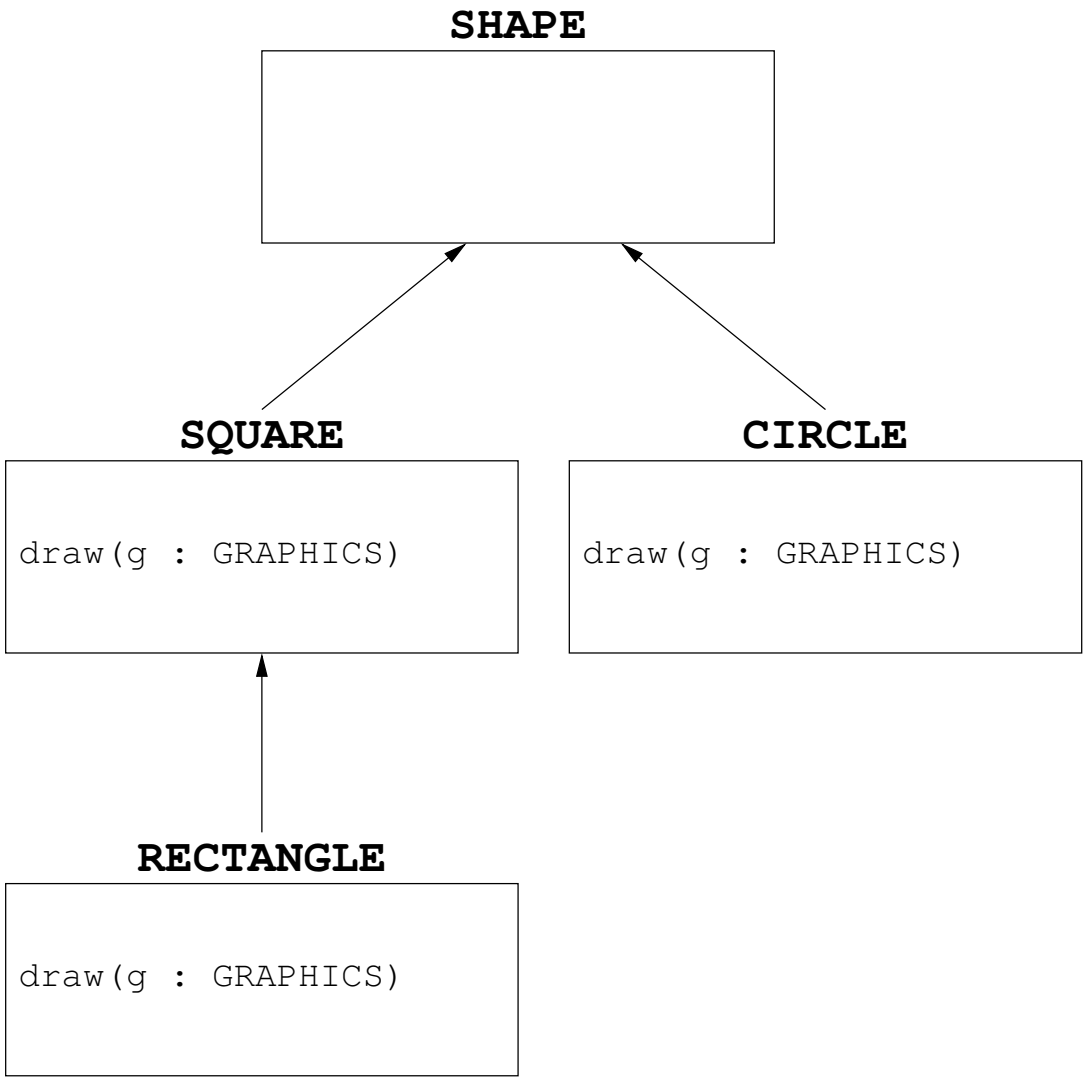
Héritage multiple et réutilisation

Pour la réutilisation aussi, l'héritage multiple est nécessaire et préférable :

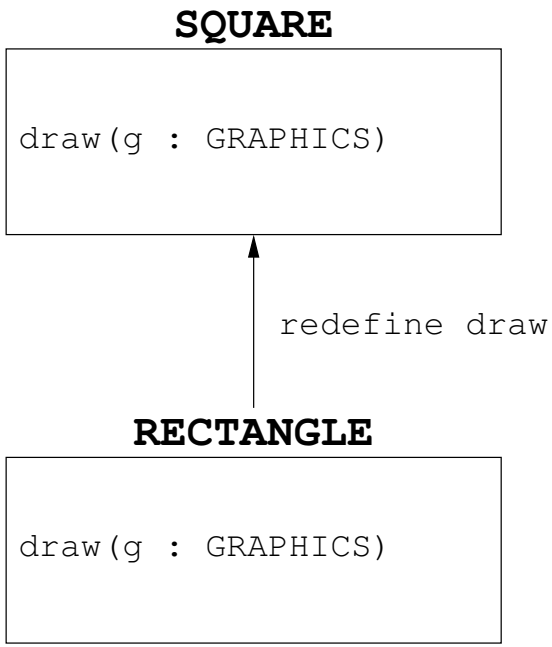


Recherche dynamique de méthode

Prenons le graphe d'héritage suivant :



Recherche dynamique de méthode



- `c : RECTANGLE`, que dire de `c.draw(g)` ?
  Il est clair que c'est la méthode `draw` de la classe `RECTANGLE` qui est appelée.
- `r : SQUARE` que dire de `r.draw(g)` ?
  Les choses sont moins claires car `r` peut contenir un `RECTANGLE` ou un `SQUARE` et il serait logique de *se baser sur la classe de l'instance* pour déterminer la méthode `draw` à utiliser.

Recherche dynamique de méthode

Si l'on se base sur le type de la variable

$\Rightarrow$ *recherche statique de méthode*

La méthode utilisée est alors déterminée à la compilation.

Si l'on se base sur le type de l'objet

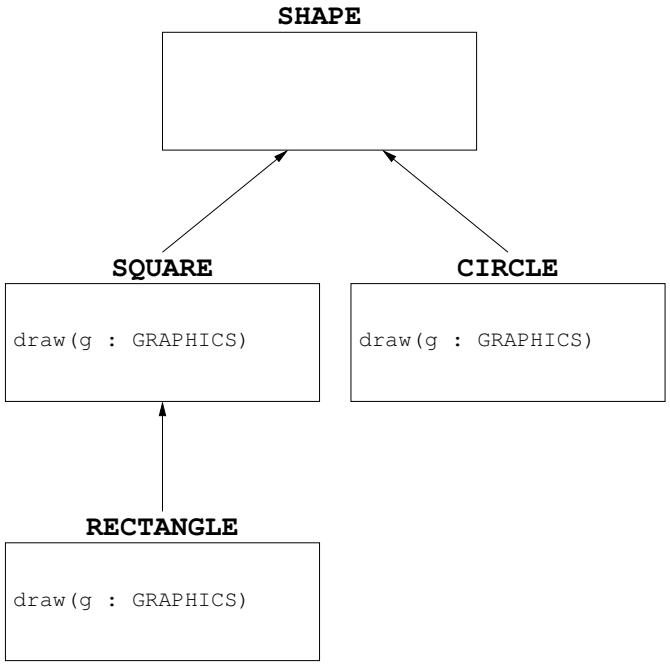
$\Rightarrow$ *recherche dynamique de méthode*

La méthode utilisée est déterminée dynamiquement au moment de l'exécution de programme.

Eiffel pratique toujours la recherche dynamique de méthode mais aussi d'attributs

Les méthodes retardées

Prenons une variable `f : SHAPE`, que peut-on dire de `f.draw(g)` ?



- Si `f` contient une instance de `SQUARE`, appeler `draw` de cette classe.
- Si `f` contient une instance de `CIRCLE`, appeler `draw` de cette classe.
- Etc.

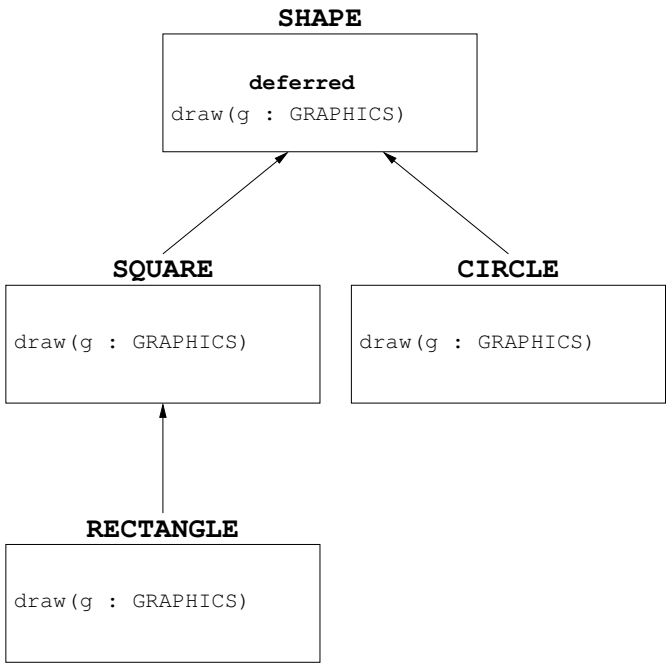
Comment être sûr que tous les objets ayant le type `SHAPE` ont une méthode `draw` ?

Les méthodes retardées

```
deferred class SHAPE

feature -- common to all the figures.
   xc,yc : INTEGER ;

   set_x_y(x,y : INTEGER) is
      do
         xc := x ;
         yc := y ;
      end ;

feature -- to implement in subclasses
   draw(g : GRAPHICS) is deferred end ;

end -- class SHAPE
```

Une méthode retardée (ou virtuelle ou abstraite) doit être mise en œuvre dans toutes les sous-classes.

Les classes retardées



- Si `f : SHAPE` contient une instance de la classe `SHAPE` alors `f.draw(g)` n'est pas définie.
- Si l'on conçoit bien ce qu'est une instance de la classe `SQUARE` ou `CIRCLE`, on voit bien qu'une instance de la classe `SHAPE` n'a aucun sens.

Les classes retardées

```
deferred class SHAPE

feature -- common to all the figures.
   xc,yc : INTEGER ;

   set_x_y(x,y : INTEGER) is
      do
         xc := x ;
         yc := y ;
      end ;

feature -- to implement in subclasses
   draw(g : GRAPHICS) is deferred end ;

end -- class SHAPE
```

- Une classe retardée ne peut être instanciée ni même avoir de routine de création.
- Une classe qui contient des méthodes retardées doit être retardée.

$\Rightarrow$ Sécurité du code

Granularité du code

Code classique

```
f : ARRAY[1..N] of SHAPE ;
for i:=1 to N do
   case type(f(i)) of
     TYPE_SQUARE :
       begin ... end;
     TYPE_CIRCLE :
       begin ... end;
     ....
   end ;
```

⇒ taille du code dépendant du nombre de types
⇒ difficilement extensible

Code objet

```
f : ARRAY[1..N] of SHAPE ;
for i:=1 to N do f.draw(g) ;
```

⇒ taille ne dépend pas du nombre de types
⇒ granularité fine du code

Extensibilité du code

Pour ajouter un nouveau type de figure, on ajoute une classe.

```
class OVAL

inherit
   SHAPE

feature

   draw(g : GRAPHICS) is
      do
         ...
      end ;
   ...
end ;
```

$\Rightarrow$ on ne touche pas à l'existant

$\Rightarrow$ le compilateur vérifie les oublis

Un point

- Héritage multiple :
  ⇒ Modélisation
  ⇒ Factorisation
  ⇒ Réutilisation
  Mais problèmes de l'héritage multiple...
- Recherche dynamique de méthodes et d'attributs :
  ⇒ Souplesse
  ⇒ Granularité du code
  Mais certains problèmes de compilation...
- Classes et méthodes retardées :
  ⇒ Sûreté du code
  ⇒ Extensibilité

Le Système Eiffel

Les Applications Eiffel

Les Éléments d'une application

Une application Eiffel est appelée un système.
Elle est composée de :

- classes :
  - une par fichier (`.e`)
  - regroupées en *clusters*
  - l'une d'elle est la classe principale
- bibliothèques Eiffel (une seule en fait)
- bibliothèques externes
- un fichier de description de l'application
  - fichier LACE, *Langage pour l'assemblage des classes en Eiffel*

Les clusters

Point de vue LOGIQUE C'est un ensemble de classes formant une partie auto-suffisante de l'application.

Point de vue PHYSIQUE Ces classes sont dans un même répertoire.

Point de vue LACE Un cluster est un nom associé à un répertoire.

Exemple de fichier lace

```
system
   geo

root
   TEST(TEST): "main"

default
   precompiled("$EIFFEL3/precomp/spec/$PLATFORM/base")

cluster
   TEST:    "$EIFFELDIR/TEST" ;
   FIGS:    "$EIFFELDIR/FIGURES" ;

external
   object: "$(EIFFEL3)/library/lex/spec/$(PLATFORM)/lib/lex.a"

end
```

Trois modes de compilation

FINALIZING réalise toutes les optimisations possibles en terme de vitesse d'exécution et d'occupation mémoire. Produit un exécutable. Très long !

FREEZING compile et produit un exécutable.

MELTING compilation incrémentale par ajout de *patch* dans le code. Très rapide : une modification n'entraîne que la recompilation de ce qu'il est vraiment nécessaire mais les performances résultantes ne sont pas très bonnes. Utile en cours de développement.

Les Interfaces

Principe 1 : abstraction des données

Quels sont les objets manipulés par l'interface ?

À chaque objet doit correspondre une classe décrivant l'objet lui-même et des routines réalisant les opérations qu'on peut lui appliquer.

Principe 2 : manipulation directe

L'utilisateur doit voir sur son écran des représentations graphiques explicites des objets de l'application.

Il doit pouvoir réaliser des opérations sur ces objets en manipulant ces leurs représentations graphiques.

Pour l'utilisateur de l'interface, l'objet doit être sa représentation graphique.

Principe 3 : cohérence sémantique

L'utilisateur doit pouvoir sélectionner tous les objets de l'interface et leurs appliquer toutes les opérations sémantiquement valides pour ces objets.

Principe 4 : typage fort

Le type d'un objet de l'interface doit être graphiquement explicite.

L'interface doit mettre en évidence les opérations valides et invalides en utilisant des techniques d'interaction favorisant la prévention de l'erreur plutôt que sa détection a posteriori.

Principe 5 : rédemption

¡¡ Mieux vaut prévenir que guérir ¿¿ s'applique aux erreurs de type mais aussi à toutes les sortes d'erreurs.

L'interface doit avertir et guider plutôt que corriger.

Elle doit proposer des solutions alternatives permettant de transformer une opération erronée en opération valide.

Le drag and drop

Très critiqué

- *Saisie d'un objet* : cliquer avec le troisième bouton et relâcher. Une représentation iconique de l'objet est attachée au pointeur de la souris.
- *Transport de l'objet* : en déplaçant le pointeur de la souris. Un trait relie en permanence l'objet au pointeur de souris.
- *Largage de l'objet* : cliquer avec le troisième bouton et relâcher le bouton, puis action ou rédemption.
- *Abandon du drag and drop* : cliquer avec un autre bouton et relâcher ce bouton.

Les accessoires

Ce sont des genres de boutons représentant des fonctions comme *Éditer*, *Détruire*, etc.

Drag and droper un objet dans son accessoire d'édition permet de l'éditer.

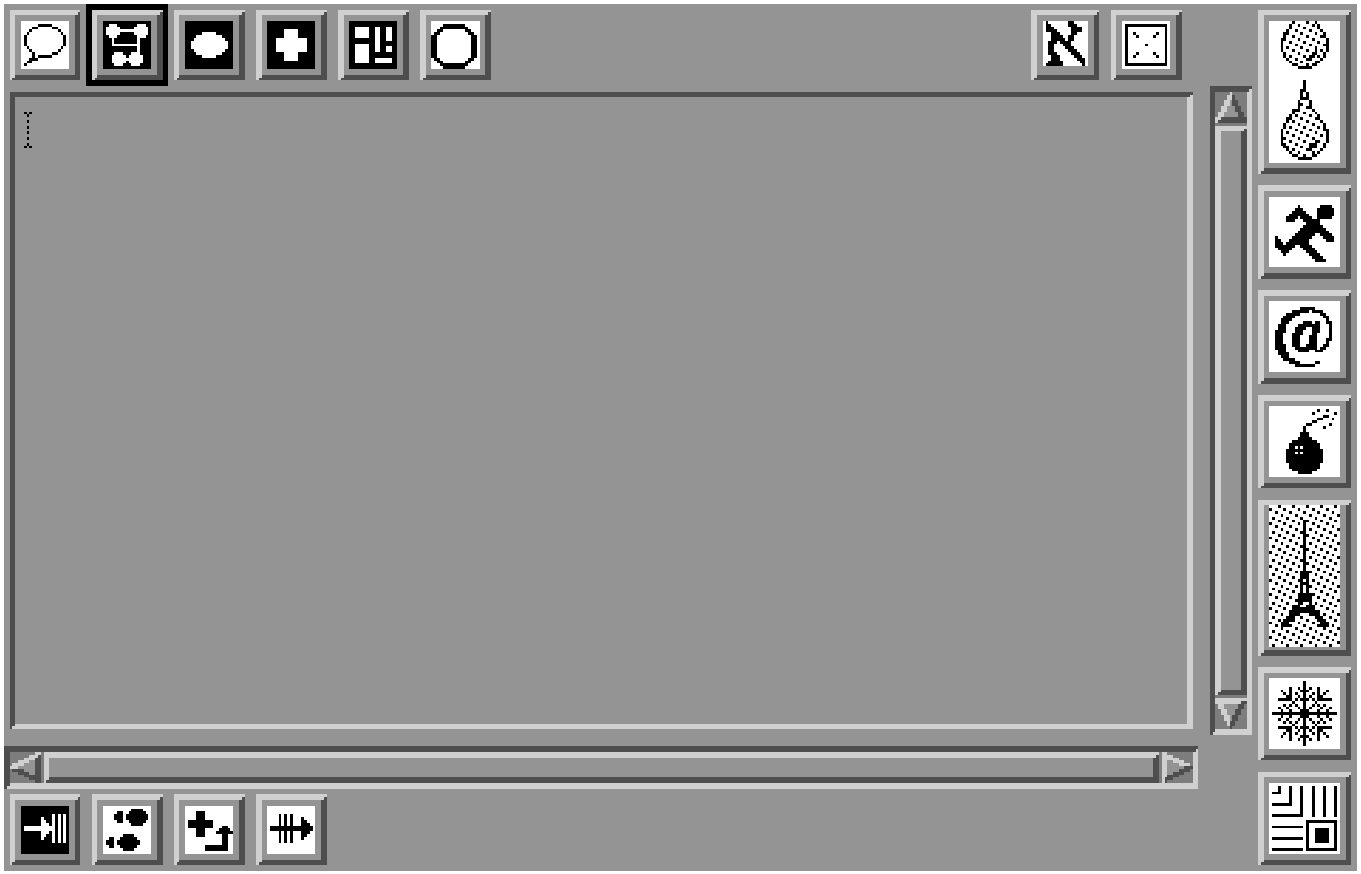
En pratique, cela donne quoi ?

L'environnement *EiffelBench* est une concrétisation de ces principes.

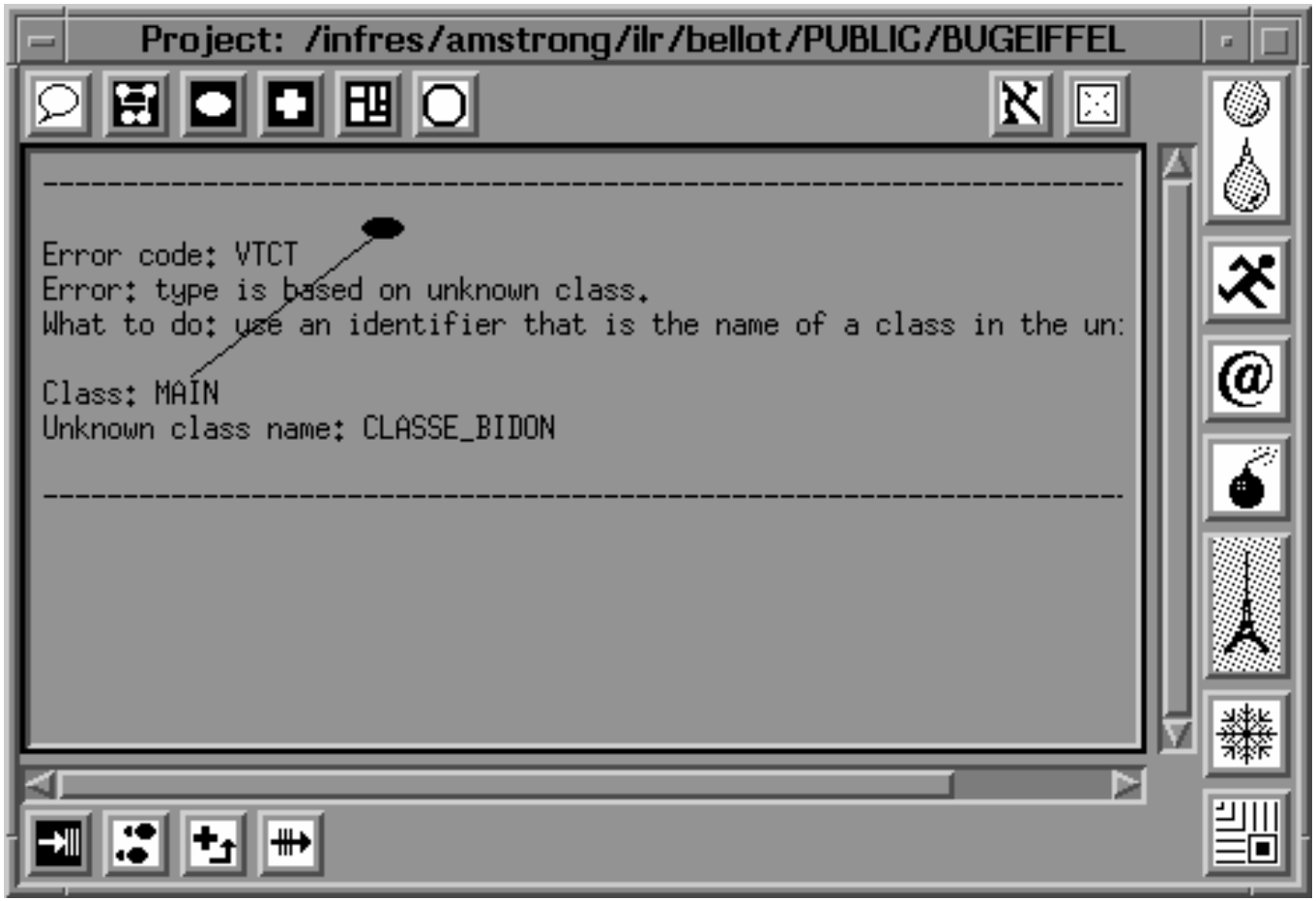
- Absence de menus.
- Absence de bouton texte.

Tout est graphique.

La fenêtre ebench



Éditer une classe



C'est général !

Chaque type d'objet a sa représentation iconifiée :

- les classes ;
- les routines ;
- les objets ;
- les codes d'erreur ;
- etc.

Les accessoires récepteurs sont dessinés en fonction de le représentation iconifiée des objets qu'ils peuvent recevoir. On retrouve la même chose dans *EiffelBuild*, le générateur d'interfaces.

Le point

On n'aime ou on n'aime pas car c'est ¡¡différent¿¿...

Quelques bons principes mais un apprentissage est nécessaire.

Très loin de faire l'unanimité !