Chap. 3: Objets et classes

Histoire des langages à objets:

1967 - Simula 67 (Origine)

1980 - Smalltalk (1er connu)

1985... - Object Pascal (Apple), Objective C (Next), C++ (B. Stroustrup), Oberon 2 (N. Wirth),...(les années folles)

1995 - Java (le préféré des jeunes)

Approche “objet”:

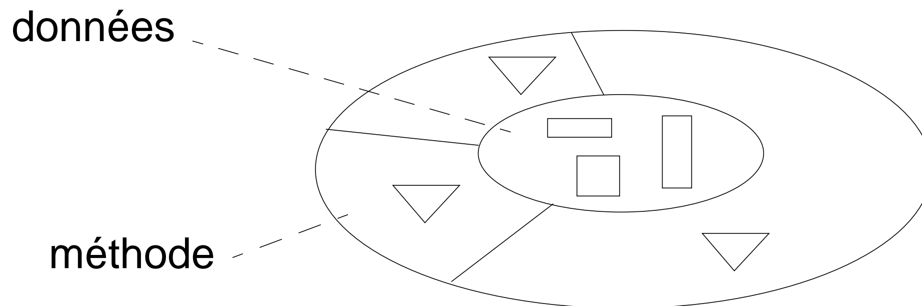
“Les langages à objets proposent de voir un système informatique comme un ensemble d’**objets** qui réalisent des activités et communiquent entre eux par envoi de **messages**. Les objets d’un même type sont regroupés dans des **classes** qui définissent leur structure et leur comportement. La réutilisation de classes existantes est facilitée par des possibilités d’adaptation, comme la notion de **sous-classe**.”

(Tiré de “Java: de l’esprit à la méthode”, M.Bonjour & al)

C’est quoi un objet ?

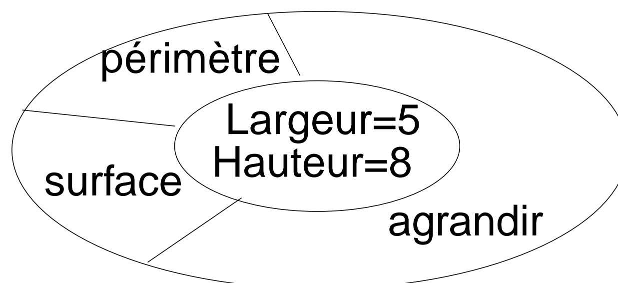
Une entité contenant:

- des données (l’état de l’objet)
- des méthodes (le comportement de l’objet)



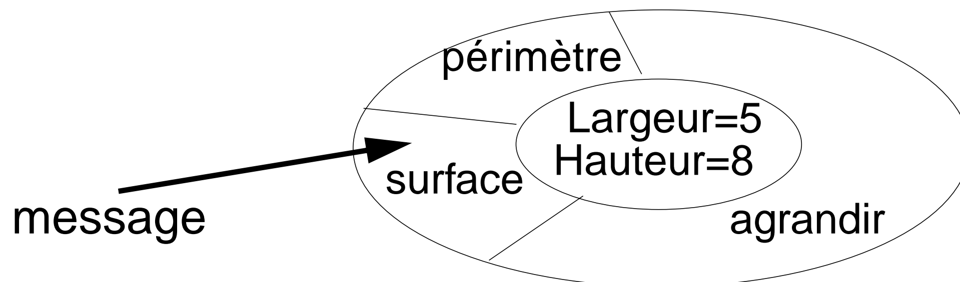
Exemple d’objet:

- le rectangle r1



Remarque: chaque objet possède une identité (object identifier ou oid)

Messages et méthodes



Le message déclenche l’exécution de la méthode sur l’objet

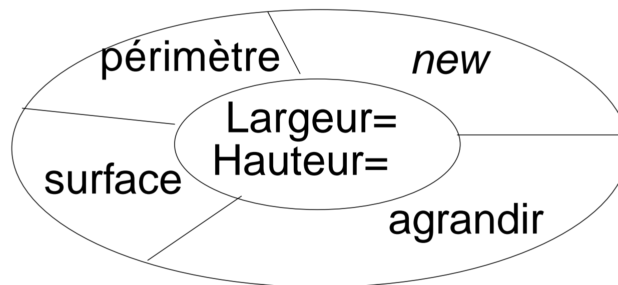
r1.surface()	retournera la valeur 40
r1.agrandir(5)	modifie l’état de r1 -> largeur = 25 et hauteur = 40
Métaphore:	pour fabriquer 1, 2, 3 objets: un artisan
	pour fabriquer N objets: une industrie -> une **classe**

Les classes

Une classe est un **moule** à fabriquer les objets

- de même structure (mêmes variables)
- de même comportement (mêmes méthodes)
- régit par les mêmes règles

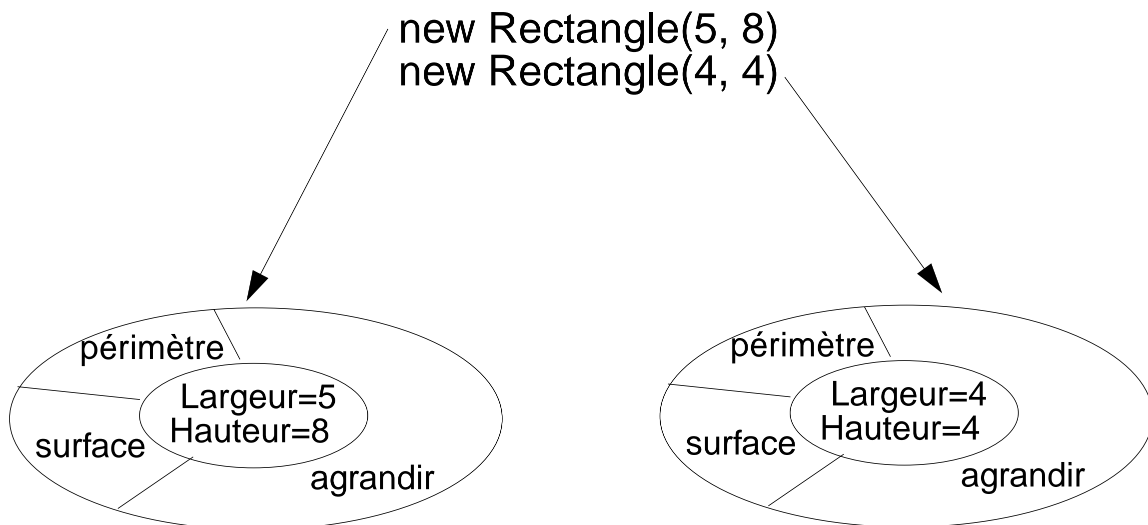
exemple: la classe *Rectangle*



Instanciation d’un objet

Instanciation d’objets à partir de la classe

Exemple:



Un objet est une instance d’une classe

Vision “comportementaliste” des objets

- une **classe** est un moule qui conditionne tous les comportements
- un **objet** est une instance d’une et une seule classe. C’est un individu qui possède tous les comportement de sa classe
- une **méthode** définit une action élémentaire que l’on peut effectuer sur un objet
- un **message** est une demande d’exécution d’une méthode à un objet.

(Tiré de “Java: de l’esprit à la méthode”, M.Bonjour & al)

Pourquoi les objets ?

Bonnes propriétés “génie logiciel”:

- Réunir les données et les algorithmes
- Encapsulation
  - cacher l’implémentation des méthodes,
  - cacher les variables
- Héritage
  - rafiner le comportement d’un objet
  - factoriser le code (unifier les parties de code communes)
- Polymorphisme
  - regrouper des objets de types différents dans la même structure de données
- Liaison dynamique
  - déclencher automatiquement la bonne méthode
  - étant donné les caractéristiques de l’objet (plus besoin d’écrire des tests explicites IF... THEN...)

→ réutilisation de logiciel !

Les objets et les classes en Oberon: Terminologie

Objet	Oberon (BB)	Mot réservé
classe	enregistrement avec méthodes, module	RECORD ou POINTER TO RECORD MODULE
	enregistrement extensible	EXTENSIBLE, ABSTRACT
objet,instance	variable (désignateur)	VAR
méthode	méthode, procédure	PROCEDURE
message	appel de méthode	
oid	~ pointeur	
self	receveur	

Les objets et les classes en Oberon

- L'extension de type

- La déclaration de classes
  - La déclaration de métodes

- Méthodes de types étendus
  - déclaration
  - héritage
  - redéfinition
  - liage dynamique

L’extension de type en Oberon

Déclaration d’un type enregistrement extensible:

```
RecordType=[ABSTRACT | EXTENSIBLE | LIMITED]
            RECORD [“(“Qualident”)”]FieldList {“;” FieldList} END
```

qualificatif	extension	instanciation(NEW)	affectation d’enregistrement
aucun (final)	-	oui	oui
ABSTRACT	oui	-	-
EXTENSIBLE	oui	oui	-
LIMITED	oui*	oui*	-

*uniquement dans le module où il est lui-même défini

Exemple: annuaire de noms de personnes

```
TYPE    Personne = POINTER TO DescPers;
        DescPers = EXTENSIBLE RECORD
                                nom: ARRAY 32 OF CHAR;
                                suivant: Personne;
                            END;
```

Extension: annuaire téléphonique de personnes

```
TYPE    DescEntrée = RECORD (DescPers)
                        noTél: ARRAY 16 OF CHAR;
                    END;
    Entrée = POINTER TO DescEntrée;
```

On dit que:

- DescPers est le type de **base**;
- DescEntrée est un **type étendu** (à partir de DescPers)

Affectations

Soit les déclarations:

```
VAR     p1, p2 : DescPers;
        e1, e2: DescEntrée;
```

Les affectations ci-dessous sont-elles valides ?

```
p1.nom := "Céleste";

p1.noTél := "3442595";

e1.nom := "Cornélius"

e1.noTél := "7322577";

p2 := p1;

e2 := e1;

p2 := e2;
```

Remarque:

Il est essentiel que le type étendu reste compatible avec le type de base. Nous verrons que cette compatibilté est réalisée avec des variables de type pointeur qui pointent vers des enregistrements extensibles / étendus.

Compatibilité

Soit les déclarations

```
VAR p,q: Personne;(* pointeur sur le type de base*)
   e : Entrée; (* pointeur sur le type étendu *)
```

Instanciation:

```
NEW(p); NEW(q); NEW(e);
p.nom:= "Babar";  e.nom:="Zéphyr";
e.noTél:= "3442595";
```

Les affectations ci-dessous sont-elles correctes ?

```
q := e;
e := p;
```

Affectation type de base / type étendu = sémantique de la projection.

Analogie: Projection d'un point de l'espace à 3 dimension sur le plan (2 dim)

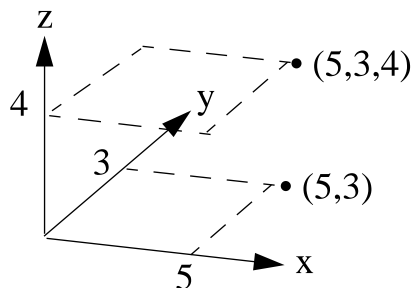
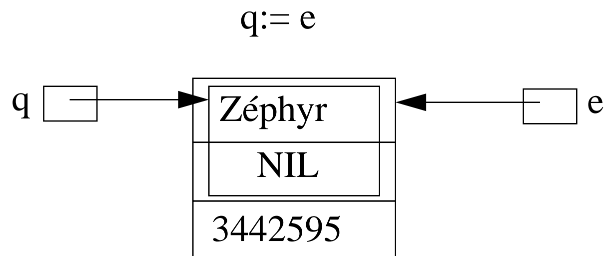
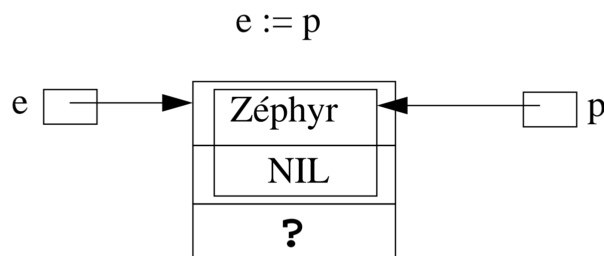


Illustration des affectations



Après l'affectation q:= e, les deux références accèdent à la même variable anonyme e^. Mais comme q est de type Personne, on ne peut pas accéder au champ noTél. Toutefois, si on est sûr que q pointe vers une instance de Entrée, le champ noTél peut être accédé (voir plus loin, garde de type). Oberon généralise le typage du point de vue statique à un point de vue dynamique.

Affectation non autorisée:



Explication: l'affectation n'aurait pas attribué de valeur au dernier champ de l'enregistrement référencé par e.

Règle d'affectation

Soit les déclarations de types (déclarations abrégée) :

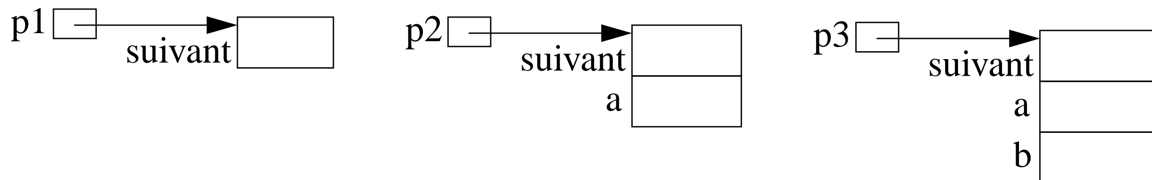
```
T1 =
 POINTER TO EXTENSIBLE
 RECORD
   suivant:T1;
 END;

T2 =
 POINTER TO EXTENSIBLE
 RECORD(T1)
   a:Donnée2;
 END;

T3 =
 POINTER TO
 RECORD(T2)
   b:Donnée3
 END;
```

- T1 est un type de base;
- T2 étend le type T1; (on dit aussi: T2 est une extension du type T1)
- T3 étend le type T2. T3 est dit type final. Par transitivité, T3 étend aussi T1.

Pointeurs de types étendus:

VAR p1:T1;p2:T2;p3:T3;

Instantiations:
NEW(p1);NEW(p2);NEW(p3)



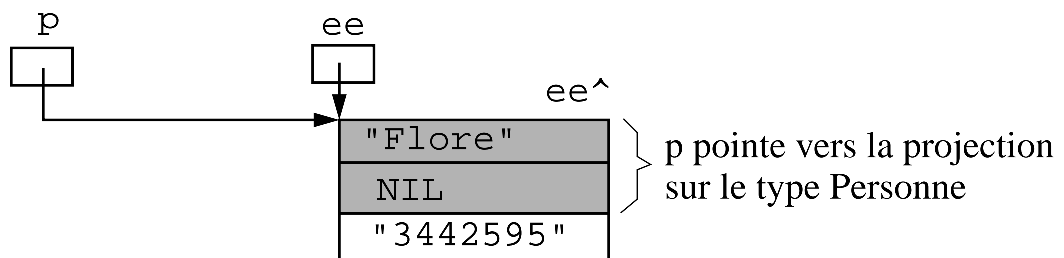
> **Règle d'affectation**: soit p1 de type pointeur T1 et p2 de type pointeur P2.
>
> L'affectation p1 := p2 est valide si T2 étend T1

p1:=p2
p2:=p1 affectation non valide !

p3:=p2 ?
p1:=p3 ?

Type dynamique

```
VAR p : Personne;
   ee, ex : Entrée;
(*voir page 2-8 pour décl. de Personne et Entrée*)
BEGIN
      NEW(p); NEW(ee);
      ee.nom:= "Flore"; ee.suivant:=NIL;
      ee.noTél:="3442595";
      p := ee;
      (* le type dynamique de p devient Entrée après
            l'exécution de cette affectation*)
```



Type dynamique

Après l'affectation p:=ee, les deux pointeurs pointent vers la même variable anonyme ee^. Le type dynamique de p est Entrée, mais son type statique demeure Personne (en gris sur le schéma).

Garde de type et test de type

Il est possible d'accéder de manière non ambiguë au champ noTél via p. Toutefois, il faut garantir par programme que p est du bon type. En Oberon on fait celà en faisant suivre la variable par une garde de type; dans notre exemple elle s'écrit:

```
p(Entrée)
```

où Entrée est le type dynamique attendu de p. Si à l'exécution, la vérification de type échoue, le programme se termine en erreur (fenêtre "Trap" et tutti quanti...)

Suite de l'exemple:

```
...
NEW(ex);
ex.noTél := p(Entrée).noTél;
```

Test du type dynamique

Si on n'est pas sûr du type dynamique de p et pour éviter une erreur à l'exécution, il est possible de le tester à l'exécution:

```
IF p IS Entrée THEN
    ex.noTél := p(Entrée).noTél;
END;
```

Garde de type dans l'instruction WITH

Permet de spécifier une garde valable dans une région délimitée par WITH ... END

Exemple: les employés d'une compagnie d'aviation.

```
TYPE    PersonDesc =       EXTENSIBLE RECORD
                                nom, prénom: ARRAY 32 OF CHAR;
                                noEmployé: INTEGER;
                                dateNais: Date;
                           END;
        PiloteDesc =       RECORD (PersonDesc)
                                heuresDeVol: INTEGER;
                           END;
        MécanicienDesc =        RECORD (PersonDesc)
                                     CodeQualification: Code;
                                END;
        Person = POINTER TO PersonDesc;
        Pilote = POINTER TO PiloteDesc;
        Mécanicien = POINTER TO MécanicienDesc;
```

Garde de type dans l'instruction WITH (suite)

```
VAR p : Person; ...

BEGIN
    NEW(p);
    ... (* traitements communs à tous les employés *)
    IF p IS Pilote THEN
        WITH p:Pilote DO p.heuresDeVol:=...;
        ... (* traitements spécifiques aux pilotes *)
        END;
    ELSIF p IS Mécanicien THEN
        WITH p:Mécanicien DO
        ...(* traitements spécifiques aux mécaniciens*)
        END;
    ELSIF p IS ...
    END;
```

Ce genre de construction étant souvent utilisée pour manipuler des variables de type étendu, Oberon offre une notation abrégée:

```
    WITH
        p: Pilote DO p.heuresDeVol:=...;
        ...(* traitements spécifiques aux pilotes *)
    |   p: Mécanicien DO
        ...(* traitements spécifiques aux mécaniciens*)
    |   p: ...
    END;
```

Page blanche

Déclaration des classes en Oberon

Rappel:

- En Oberon, il est possible de déclarer des procédures spéciales (appelées “méthodes”) qui sont syntaxiquement connectées à un type, plus précisément à un RECORD ou à un POINTER TO RECORD
- Les types (POINTER TO) RECORD auxquels sont connectées des méthodes s’appellent des “classes”.
- Les instances (ou valeurs) de ces classes sont applées “objets”.
- Lorsque les classes sont construites à partir de types extensibles, on parle de sous-classes
- Les messages (appel des méthodes) donne lieu à un liage dynamique des méthodes aux objets.

Déclaration des classes (suite)

- La déclaration du (pointer to) record se fait "normalement"
- Pour rattacher les méthodes à un (pointer to) record:
  - on fait précéder le nom de la méthode par un paramètre formel du type du record
  - ce paramètre est appelé le "receveur"
  - si la classe est construite avec un type RECORD, le "receveur" est soit VAR soit IN
  - si la classe est construite avec un type POINTER TO RECORD, le receveur est "par valeur" (pas de qualificatif)
- En Oberon on ne peut pas définir de méthode d'instanciation pour les classes POINTER TO RECORD (constructeur en Java). Pour instancier un objet, il faut explicitement créer une nouvelle instance avec NEW (comme pour les pointeurs) puis invoquer une méthode d'initialisation des champs de l'objet.

Déclaration (exemple): la classe Rectangle

```
MODULE Figure;

TYPE Rectangle* = POINTER TO RECORD
            largeur, hauteur: REAL;
      END;

PROCEDURE(r: Rectangle)InitRectangle*(l,h :REAL),NEW;
BEGIN
      r.largeur:=l; r.hauteur:=h;
END InitRectangle;

PROCEDURE(r: Rectangle)Agrandir*(coef:REAL),NEW;
BEGIN
      r.largeur:=r.largeur * coef; r.hauteur:=r.hauteur*coef;
END Agrandir;

PROCEDURE (r: Rectangle) Périmètre*():REAL, NEW;
BEGIN
      RETURN 2 * (r.largeur + r.hauteur);
END Périmètre;

PROCEDURE (r: Rectangle) Surface*(): REAL, NEW;
BEGIN
      RETURN r.largeur * r.hauteur;
END Surface;

PROCEDURE (r: Rectangle) Dessiner*(), NEW;
BEGIN
      ... (* dessine le rectangle à l'écran *)...
END Dessine;
END Figure.
```

Interface et utilisation de la classe Rectangle

Interface générée par BB pour la classe Rectangle:

```
DEFINITION Figure;

	TYPE
		Rectangle = POINTER TO RECORD
			(r: Rectangle) Agrandir (coef: REAL),NEW;
			(r: Rectangle) Dessiner, NEW;
			(r: Rectangle) InitRectangle (l, h: REAL), NEW;
			(r: Rectangle) Périmètre (): REAL, NEW;
			(r: Rectangle) Surface (): REAL, NEW
		END;

END Figure.
```

Exemple d'utilisation de la classe rectangle par un module client:

```
MODULE Client;
IMPORT Figure;

VAR r1, r2: Figure.Rectangle;
BEGIN
	...
	NEW(r1); r1.InitRectangle(5, 8);
	NEW(r2); r2.InitRectangle(4, 4);
	r1.Agrandir(4.5);
	...
END Client.
```

Déclaration de classes et sous-classes

Classes et sous-classes en Oberon:

- On utilise l'extension de types
- Une sous-classe hérite des champs **et** des méthode de la classe dont elle est issue
- De nouvelle méthodes peuvent être définies dans une sous-classe
- Les méthodes peuvent être **redéfinie** dans une sous-classe, celles-ci masqueront alors les méthodes de même nom d'une classe plus basique
- L'opérateur ^ , permet d'atteindre les méthodes "masquées" par redéfinition ("super call").
  Notation: o.Q^(<liste des paramètres>)

Particularités BB:

- Pour être extensibles, les types doivent être explicitement qualifiée EXTENSIBLE (ou ABSTRACT)
- Pour autoriser sa redéfinition dans une sous-classe, une méthode doit être explicitement qualifiée EXTENSIBLE (ou ABSTRACT)
- Remarque: on ne peut pas instancier une classe qualifiée ABSTRACT; une méthode ABSTRACT se résume à sa signature (et ne contient pas de code)

Exemple de sous-classe: RectangleTramé

A partir de la classe Rectangle, on aimerait définir une classe de rectangles tramés (rectangles avec un fond coloré)

Premièrement, dans le module *Figure*, il faut rendre la classe *Rectangle* extensible, càd:

- qualifier l'enregistrement d'extensible
- idem pour les méthodes que l'on voudra redéfinir dans la (ou les) sous-classe (*Dessiner* dans notre exemple)

Les modifications sont minimes:

```
DEFINITION Figure;

    TYPE
        Rectangle = POINTER TO EXTENSIBLE RECORD
            (r: Rectangle) Agrandir (coef: REAL), NEW;
            (r: Rectangle) Dessiner, NEW, EXTENSIBLE;
            (r: Rectangle) InitRectangle (l, h: REAL), NEW;
            (r: Rectangle) Périmètre (): REAL, NEW;
            (r: Rectangle) Surface (): REAL, NEW
        END;

END Figure.
```

Exemple de sous-classe: RectangleTramé (suite)

Déclaration de la sous-classe *RectangleTramé* par extension de la classe *Rectangle*

```
MODULE Figure;
TYPE Rectangle* = POINTER TO EXTENSIBLE RECORD
            ...(* voir p. 3-21 pour les champs et les méthodes *)

      RectangleTramé* = POINTER TO RECORD (Rectangle)
            couleurTrame: INTEGER;
      END;

PROCEDURE(r: RectangleTramé)InitRectangleTramé*
                                    (l,h:REAL; ct: INTEGER), NEW;
BEGIN
      r.InitRectangle(l,h); r.couleurTrame:=ct;
END InitRectangleTramé;

PROCEDURE (r: RectangleTramé)Dessiner*();
BEGIN
       ...(* dessine le rectangle tramé à l'écran *)...
END Dessiner;
END Figure.
```

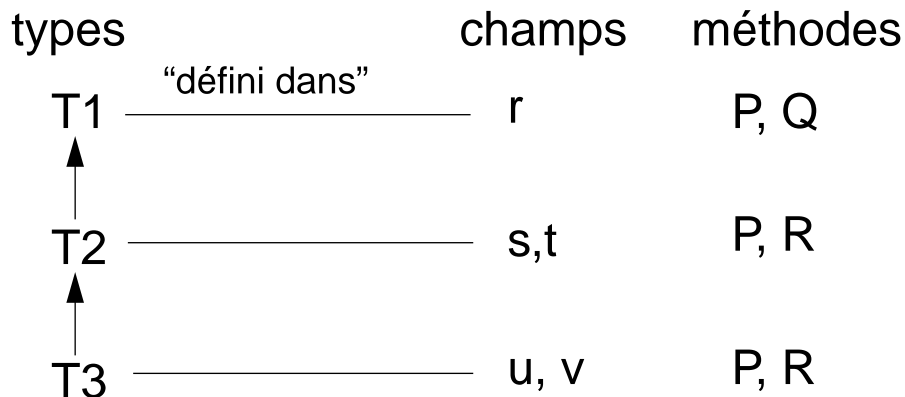
Remarques:

- la méthode *Dessiner* redéfinit celle de *Rectangle*
- la méthode *InitRectangleTramé* initialise uniquement son champ spécifique et utilise la méthode *InitRectangle* pour initialiser les champs de base

Héritage dans une hiérarchie de classes

Rappel: une sous-classe (type étendu) **hérite** de tous les champs **et** méthodes définis dans la classe (le type) dont elle (il) est l’extension

Exemple:



Lecture de la hiérarchie:

- T1 est le type (classe) de base,
- T2 étend T1
- T3 étend T2 (et T1 par transitivité)

Héritage:

- T2 hérite de r et de Q (mais P est redéfini)
- T3 hérite de r,s,t et de Q (mais P et R sont redéfini)

Liage dynamique

A l’exécution, les méthodes sont liées à l’objet en fonction de son type dynamique et de la hiérarchie des types.

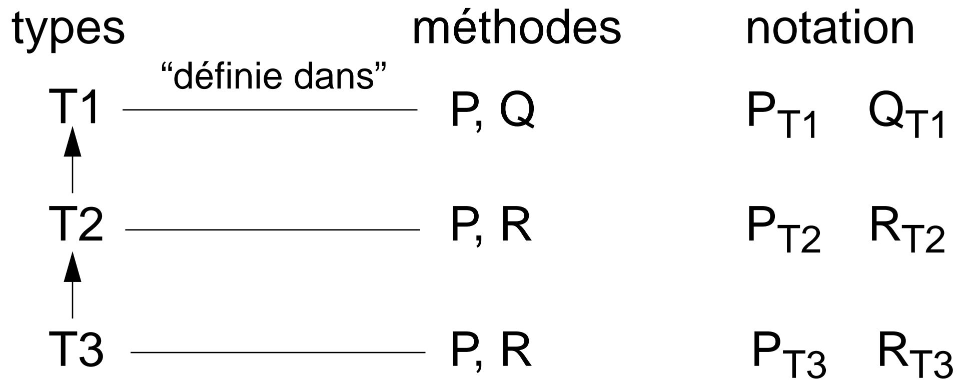
Principe

Un objet o de type dynamique T reçoit le message o.Q:

- si la méthode Q est définie (ou redéfinie) dans le type T, c’est cette méthode qui sera déclenchée
- si la méthode Q n’est pas définie dans le type T, c’est la première méthode Q trouvée en remontant la hiérarchie des types de base qui sera déclenchée.

Liage dynamique - exemples

Soit la hiérarchie de types étendus:



Soit les déclarations:

```
VAR a: T1;    b: T2;    c: T3;
```

Les messages suivants déclenche l'exécution de:

```
NEW(a);
a.P;     (* -> méthode   P_T1   *)
NEW(b);
b.P;     (* -> méthode   P_T2   *)
b.R;     (* -> méthode   R_T2   *)
NEW(c);
c.P;     (* -> méthode   P_T3   *)
c.Q;     (* -> méthode   Q_T1   *)
b:=c;
b.R;     (* -> méthode   R_T3   *)
a:=b;
a.P;     (* -> méthode   P_T3   *)
a.Q;     (* -> méthode   Q_T1   *)
```

Objets et classes: à quoi ça sert ?

- types abstraits: regrouper dans une unité syntaxique les données et les opérations (voir chapitre 2, pp 2-25 à 2-32)
- structures de données génériques: construire des structures dont le type des composantes élémentaires ne sera défini qu'au moment de l'utilisation de la structure par le client. Ex: définition du type "queue prioritaire" puis déclaration par le client du type des éléments de la queue, queue d'entiers, queue de processus, queue de "personnes" etc.
- **structures de données hétérogènes** (exemple dans la suite du chapitre). Le plus intéressant car utilisation du liage dynamique.
- composantes semi-finies: définition du noyau d'une structure de données; le client l'étendra au moment de l'utilisation. Ex: type définissant les opérations les plus basiques d'une fenêtre graphique. Le client l'enrichira avec tous les "gadgets" dont il aura besoin (boutons, ascenseur, couleur, agrandisseur etc.)

Structure de données hétérogène

Structure hétérogène: “Structure composée de différents types d’éléments, néanmoins apparentés.”

Principe d’implémentation avec les classes:

(1) déclarer un type de base commun

(2) étendre le type de base pour chaque type de données particulier

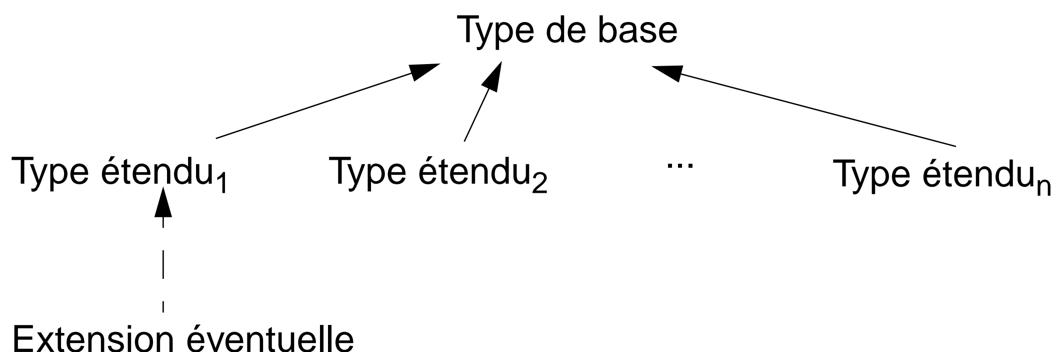
Le type de base comprendra:

- les liens entres les objets (liste etc.)
- les données communes à tous les objets

Les types étendus comprendront:

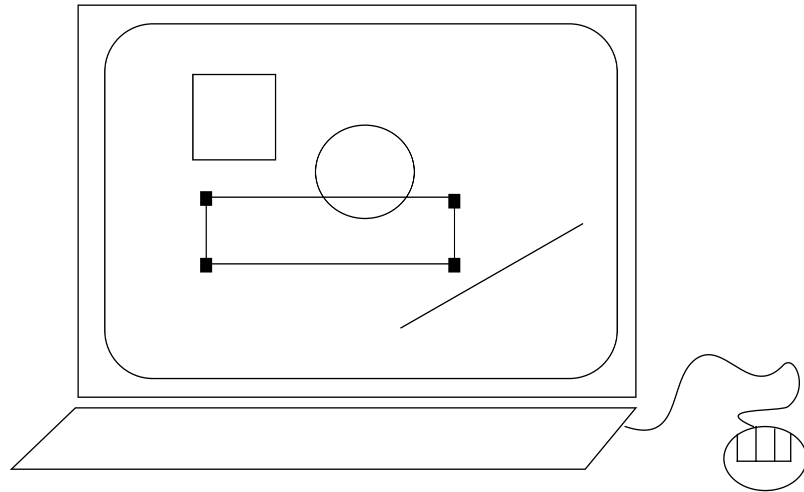
- les données spécifiques à chaque extension

Hiérarchie des classes:



Structure de données hétérogène: un exemple

Editeur graphique



Figures:

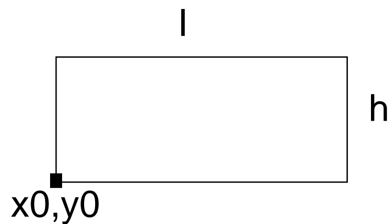
- segment de droite (ligne)
- rectangle
- cercle

Opérations:

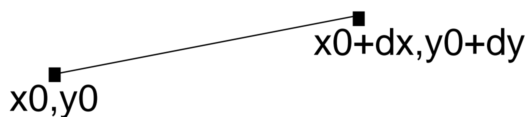
- créer (initialiser) une figure
- sélectionner
- supprimer
- déplacer
- dessiner (redessiner) toutes les figures

Structure de données

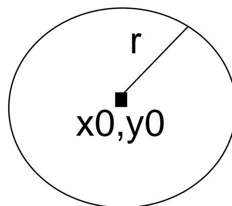
Rectangle:



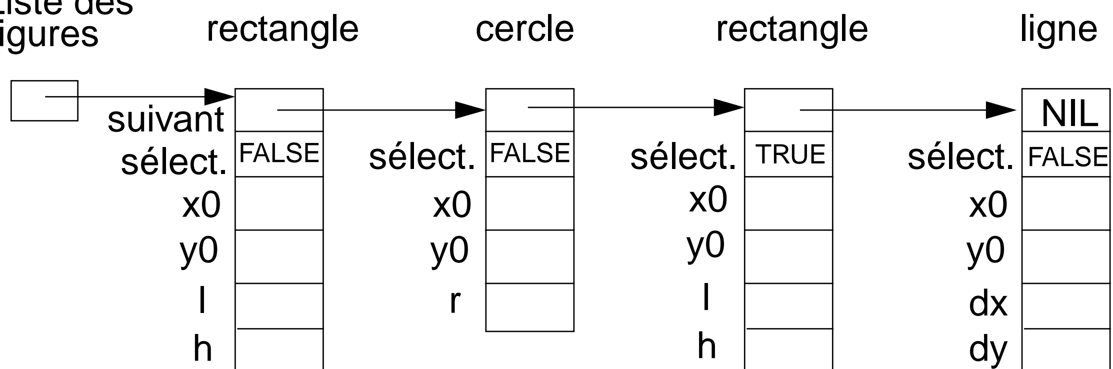
Ligne:



Cercle:



Pour garder la trace des figures affichées à l'écran, l'éditeur utilise une *liste* d'objets graphiques:



Le type de base “Figure”

Le type “Figure” définit les données et les opérations communes à toutes les figures:

```
TYPE Figure* = POINTER TO ABSTRACT RECORD
                suivant: Figure;
                sélectionné: BOOLEAN;
                x0-, y0-: INTEGER; (* origine de la figure*)
        END;

PROCEDURE(f: Figure)InitFigure*(listef: ListeFigures;
                                x,y: INTEGER), NEW;
    ... (* initialise l’origine de la figure et l’ajoute à la liste *)...

PROCEDURE(f: Figure)Dessiner*, NEW, ABSTRACT;
    (* la méthode de dessin est spécifique à chaque figure et
    devra être définie pour chaque type de figure *)

PROCEDURE(f: Figure)Déplacer*(dx ,dy :INTEGER), NEW;
    ... (* déplacer la figure de dx horizontalement et de dy
        verticalement *) ...

PROCEDURE(f: Figure)Effacer*(listef: ListeFigures), NEW;
    ... (* effacer la figure *) ...

PROCEDURE (f: Figure)Sélectionner*, NEW;
    ... (* marquer une figure comme étant sélectionnée *) ...
```

Pour le code complet des méthodes, voir annexe p. 3-44

Le type pour gérer la liste de figures

Le type “ListeFigures” est défini pour instancier un objet qui servira d’”ancre” à la liste de figures; ses opérations servent à gérer la liste de figures:

```
TYPE    ListeFigures* = RECORD
              tête: Figure;
        END;

PROCEDURE(VAR listef: ListeFigures)InitListeFigures*, NEW;
    ... (* vider la liste de figures *) ...

PROCEDURE(VAR listef:ListeFigures)Ajouter(f:Figure), NEW;
    ... (* ajouter la figure “f” à la fin de la liste *) ...

PROCEDURE(VAR listef: ListeFigures)Supprimer*(f: Figure),
                                        NEW;
    ... (* supprimer la figure “f” de la liste *) ...

PROCEDURE(VAR listef: ListeFigures)Ouvrir*, NEW;
    ... (* ouvrir le fichier et charger les figures dans la liste *)...

PROCEDURE(IN listef: ListeFigures)Sauver*, NEW;
    ... (* sauver la liste de figures dans un fichier *)...

PROCEDURE(IN listef: ListeFigures)DessinerToutes*, NEW;
    ... (* dessiner toutes les figures de la liste à l’écran *)...

PROCEDURE(IN listef: ListeFigures)DéplacerToutes*
  (dx, dy: INTEGER), NEW; ...(* déplacer toutes les figures *)...
```

L’interface des structures de base

Le type de base ainsi que le type de la liste de figures sont définis dans le module “Figures”. L’interface du module “Figures” est:

```
DEFINITION Figures;
TYPE
    Figure = POINTER TO ABSTRACT RECORD
        x0-, y0-: INTEGER;
        (f: Figure) Dessiner, NEW, ABSTRACT;
        (f: Figure) Déplacer (dx, dy: INTEGER), NEW;
        (f: Figure) Effacer (listef: ListeFigures), NEW;
        (f: Figure) InitFigure (listef: ListeFigures;
                                x, y: INTEGER), NEW;
        (f: Figure) Sélectionner, NEW
    END;

    ListeFigures = RECORD
        (IN listef: ListeFigures) DessinerToutes, NEW;
        (IN listef: ListeFigures) DéplacerToutes
                                (dx, dy: INTEGER), NEW;
        (VAR listef: ListeFigures)InitListeFigures, NEW;
        (VAR listef: ListeFigures) Ouvrir, NEW;
        (IN listef: ListeFigures) Sauver, NEW
        (VAR listef: ListeFigures) Supprimer (f: Figure),
                                NEW
    END;
END Figures.
```

Un exemple d’extension: le type “rectangle”

Chaque figure spécifique est représentée par l’instance d’un type qui **étend le type de base** “Figure”. Le type de la figure spécifique est définit dans un module client.

Par exemple, pour les rectangles:

```
TYPE Rectangle* = POINTER TO EXTENSIBLE
        RECORD(Figures.Figure)
            l, h: INTEGER; (* largeur et hauteur *)
        END;

PROCEDURE(r: Rectangle)Dessiner*, EXTENSIBLE;
BEGIN
    (* ... dessiner le rectangle à l’écran...*)
END Dessiner;

PROCEDURE(r: Rectangle)InitRectangle*
                        (listef: Figures.ListeFigures), NEW;
VAR origX, origY, arrX, arrY: INTEGER;
BEGIN
(* ... capter les clicks et les mouvements de souris ...*)
    r.InitFigure(listef, origX, origY);
    r.l:=arrX - origX; r.h:=arrY - origY;
    r.Dessiner;
END InitRectangle;
```

Le type rectangle: remarques

- la méthode “Dessiner” redéfinit la méthode abstraite “Dessiner” du type “Figure”.
- si le type “Rectangle” n’avait pas redéfini la méthode “Dessiner”, le compilateur aurait signalé l’erreur car le type de base l’avait définie abstraite.
- la méthode “InitRectangle” initialise les données propres aux rectangles et utilse la méthode héritée “InitFigure” pour initialiser les autres données.
- l’instanciation d’un rectangle, NEW(r), si r est déclaré de type Rectangle, se fait par contre en dehors de la méthode “InitRectangle”. En effet, pour qu’un objet puisse traiter un message, par exemple r.InitRectangle(4,5), il doit être instancié au préalable. C’est donc au module client de se charger de l’instanciation.

Le type “cercle”

Tout comme “Rectangle”, le type “Cercle” est une extension du type de base “Figure”:

```
TYPE Cercle* = POINTER TO EXTENSIBLE
            RECORD(Figures.Figure)
                r: INTEGER; (* rayon du cercle *)
            END;

PROCEDURE(c: Cercle)Dessiner*, EXTENSIBLE;
VAR
BEGIN
    (* ... dessiner la ligne dans la fenêtre ...*)
END Dessiner;

PROCEDURE(c: Cercle)InitCercle*
                        (listef: Figures.ListeFigures), NEW;
VAR origX, origY, arrX, arrY: INTEGER;
BEGIN
    (* ... capter les clicks et les mouvements de souris ...*)
    c.InitFigure(listef, origX, origY);
    c.r:=SHORT(ENTIER(Math.Sqrt((arrX - origX)
                *(arrX - origX)+ (arrY - origY)*(arrY - origY))));
    c.Dessiner;
END InitCercle;
```

Le bénéfice du liage dynamique

Une action typique sur une structure hétérogène est le parcours de toute la structure et le traitement de chaque élément en fonction de son type spécifique.

Exemple: Le type “ListeFigures” définit une méthode pour dessiner toutes les figures de la liste. **Sans** utiliser le **liage dynamique** nous aurions:

```
PROCEDURE(IN listef: ListeFigures)DessinerToutes*, NEW;
(*appel des procédures de dessin sans liage dynamique *)
VAR fig: Figure;
BEGIN
	fig:=listef.tête;
	WHILE fig # NIL DO
		IF fig IS Rectangle THEN
			(*  ... dessiner le rectangle... *)
		ELSIF fig IS Ligne THEN
			(*  ... dessiner la ligne... *)
		ELSIF fig IS Cercle THEN
			(*  ... dessiner le cercle... *)
		END;
		fig.Dessiner; fig:=fig.suivant;
	 END;
END DessinerToutes;
```

Le bénéfice du liage dynamique (suite)

Supposons que l'on ajoute les triangles à l'éditeur graphique

```
TYPE Triangle* = POINTER TO EXTENSIBLE
            RECORD(Figures.Figure)
                  x1, y1, x2, y2: INTEGER;
            END;

PROCEDURE(t: Triangle)Dessiner*, EXTENSIBLE; ...

PROCEDURE(t: Triangle)InitTriangle*
                              (listef: Figures.ListeFigures), NEW;...
```

La méthode “DessinerToutes” doit être modifiée par l'ajout des instructions:

```
            ...
            ELSIF fig IS Triangle THEN
                  ... (*dessiner le triangle *) ...
            ELSIF
            ...
```

De manière générale, partout où il y un traitement qui est dépendant du type de la figure, il faudra ajouter:

```
            ...
            ELSIF fig IS Triangle THEN
                  WITH fig: Triangle DO
                        ... (*manipulation du triangle *) ...
                  END;
            ELSIF
            ...
```

Le bénéfice du liage dynamique (suite)

Méthode “DessinerToutes” en **utilisant le liage dynamique**:

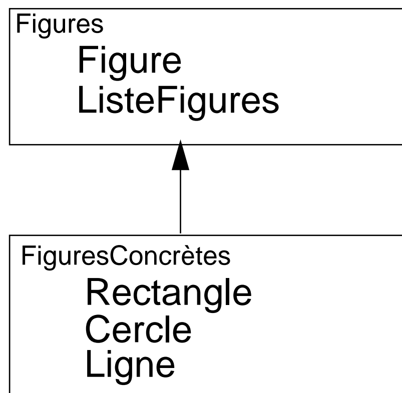
```
PROCEDURE(IN listef: ListeFigures)DessinerToutes*, NEW;
VAR fig: Figure;
BEGIN
	fig:=listef.tête;
	WHILE fig # NIL DO
		fig.Dessiner;
		fig:=fig.suivant;
	END;
END DessinerToutes;
```

Bénéfice:

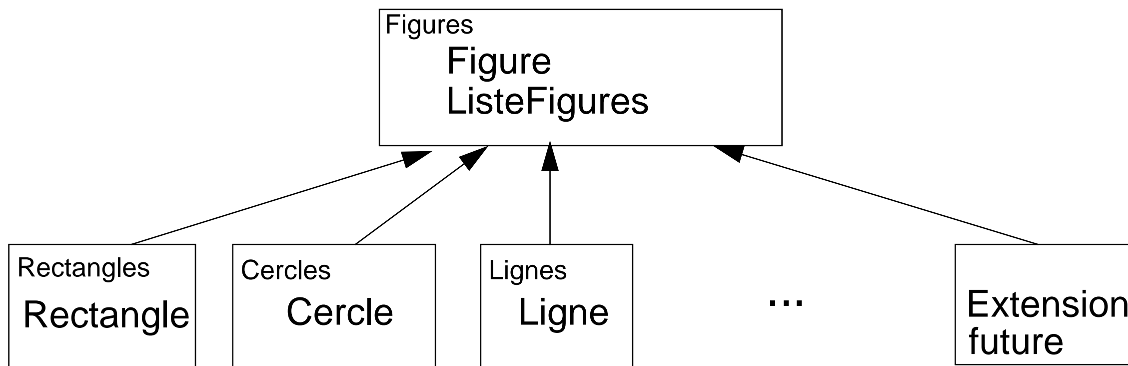
- la méthode qui dessine tous les membres de la liste des figures est **indépendante** du type de la figure
- elle est valide pour toutes les figures manipulées par l’éditeur graphique (rectangle, cercle, ligne, triangle) ainsi que tout type de figure ajouté dans le futur !
- la méthode “Dessiner” appropriée sera déclenchée par liage dynamique, c’est-à-dire en fonction du type dynamique de la figure.
- l’ajout d’un nouveau type de figure ne nécessitent aucune modification du type de base, ni de la recompilation du module où le type de base est défini.

Objets et modules

Dans le code donné en annexe (page 3-44), le type de base et le type sont définis dans le module “Figures” et les types étendus (rectangle, cercle ...) sont regroupés dans le module “FiguresConcrètes”:



Néanmoins, pour pouvoir ajouter de nouveaux types de figures de manière tout à fait indépendante (sans avoir à recompiler les autres types de figures), il faut créer un module par type étendu. Dans ce cas, la hiérarchie des modules reflète fidèlement la hiérarchie des classes:

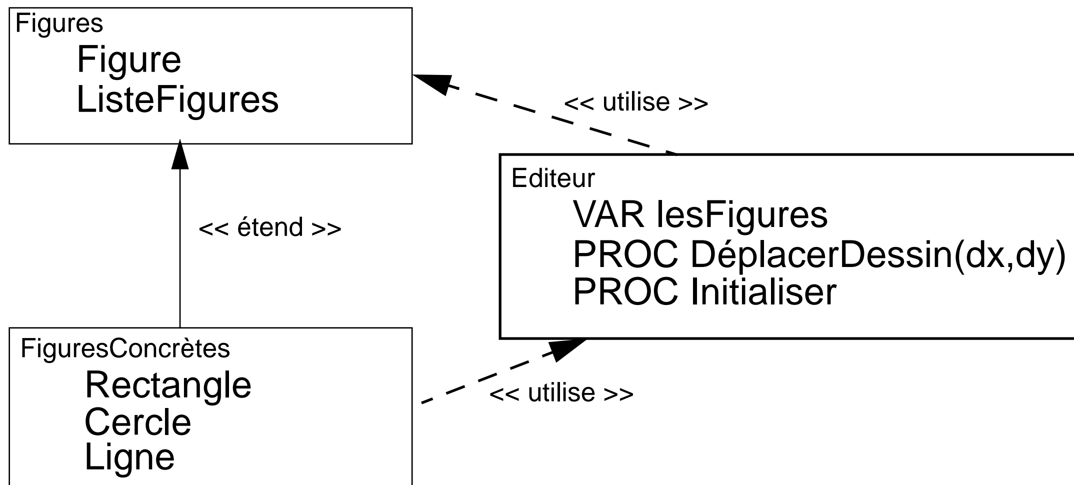


Objets et structures hétérogènes: En résumé

- Un type de base définit toutes les données et opérations communes aux objets (composantes) de la structure. Il définit également les données qui lient les objets entre eux(champ “suivant” dans notre exemple).
- Certaines des méthodes sont “abstraites” : elle devront obligatoirement être redéfinies par les types étendus. A l’exécution, le liage dynamique choisira la bonne méthode à déclencher.
- Un autre type est défini, destiné à ancrer et à gérer la structure d’objets dans sa globalité.
- On encapsule le type de base dans un module: les types étendus ne verront que le nom du type de base ainsi que la signature des méthodes exportées.
- Les types étendus ajoutent les champs et opérations spécifiques à chaque type d’objet.
- C’est aux types étendus que revient la responsabilité de définir la méthode d’**initialisation** des instances de composantes; typiquement on nomme cette opération “Init” suivi du nom du type (p.e. “InitRectangle”).
- Mais c’est au module client de ces types que revient la tâche **d’instancier** les objets (NEW...).
- Les structures de données hétérogènes et le liage dynamique des méthodes constituent l’essence même de l’approche orientée objet.

Annexe: le code de l’éditeur graphique

- Architecture



- Le module *Editeur* a pour but
  - d’”ancrer” les figures du dessin avec la variable *lesFigures*
  - de définir les deux procédures *DéplacerDessin* et *Initialiser* qui sont liées aux boutons de l’interface graphique de l’éditeur

Module *Figures*

```
MODULE Figures;

TYPE Figure* =          POINTER TO ABSTRACT RECORD
                              suivant: Figure;
                              sélectionné: BOOLEAN;
                              x0-, y0-: INTEGER; (* origine *)
                        END;

        ListeFigures* =RECORD
                              tête: Figure;
                        END;

PROCEDURE(VAR listef: ListeFigures)InitListeFigures*, NEW;
BEGIN
        listef.tête:=NIL;
END InitListeFigures;

PROCEDURE(VAR listef: ListeFigures)Ajouter(f: Figure), NEW;
VAR fig: Figure;
BEGIN
        IF listef.tête = NIL THEN listef.tête:=f;
        ELSE
                fig:=listef.tête;
                WHILE fig.suivant # NIL DO fig:=fig.suivant; END;
                fig.suivant:=f;
        END;
END Ajouter;

PROCEDURE(VAR listef: ListeFigures)Supprimer*(f: Figure), NEW;
VAR fig, prec: Figure;
BEGIN
        fig:=listef.tête;
        WHILE (fig # NIL) & (fig # f) DO prec:=fig; fig:=fig.suivant; END;
        IF fig # NIL THEN prec.suivant:=fig.suivant; END;
END Supprimer;

PROCEDURE(VAR listef: ListeFigures)Ouvrir*, NEW;
BEGIN (* ...ouvrir un fichier et charger le graphe dans la liste... *)
END Ouvrir;

PROCEDURE(IN listef: ListeFigures)Sauver*, NEW;
BEGIN (* ...sauver le graphe dans un fichier... *)
END Sauver;
```

Module *Figures* (suite)

```
PROCEDURE(f: Figure)Dessiner*,  NEW, ABSTRACT;

PROCEDURE(IN listef: ListeFigures)DessinerToutes*, NEW;
VAR fig: Figure;
BEGIN
        fig:=listef.tête;
        WHILE fig # NIL DO
                fig.Dessiner;
                fig:=fig.suivant;
        END;
END DessinerToutes;

PROCEDURE(f: Figure)Déplacer*(dx, dy: INTEGER), NEW, EXTENSIBLE;
BEGIN
        f.x0:=f.x0 + dx; f.y0:=f.y0 + dy;
END Déplacer;

PROCEDURE(IN listef: ListeFigures)DéplacerToutes*(dx, dy: INTEGER), NEW;
VAR fig: Figure;
BEGIN
        fig:=listef.tête;
        WHILE fig # NIL DO
                fig.Déplacer(dx,dy);
                fig:=fig.suivant;
        END;
END DéplacerToutes;

PROCEDURE(f: Figure)InitFigure*(VAR listef: ListeFigures; x,y: INTEGER), NEW;
BEGIN
        f.x0:=x; f.y0:=y; f. sélectionné:=TRUE;
        listef.Ajouter(f);
END InitFigure;

PROCEDURE(f: Figure)Effacer*(listef: ListeFigures), NEW;
BEGIN
        listef.Supprimer(f);
END Effacer;

PROCEDURE (f: Figure)Sélectionner*, NEW;
BEGIN
        f.sélectionné:=TRUE;
END Sélectionner;
END Figures.
```

Module *Editeur*

```
MODULE Editeur;

IMPORT Figures,  XYplane;

VAR lesFigures*: Figures.ListeFigures;

PROCEDURE DéplacerDessin*(dx, dy: INTEGER);
BEGIN
        lesFigures.DéplacerToutes(dx,dy);
        XYplane.Clear;
        lesFigures.DessinerToutes;
END DéplacerDessin;

PROCEDURE Initialiser*;
BEGIN
        XYplane.Open;
        lesFigures.InitListeFigures;
END Initialiser;

END Editeur.
```

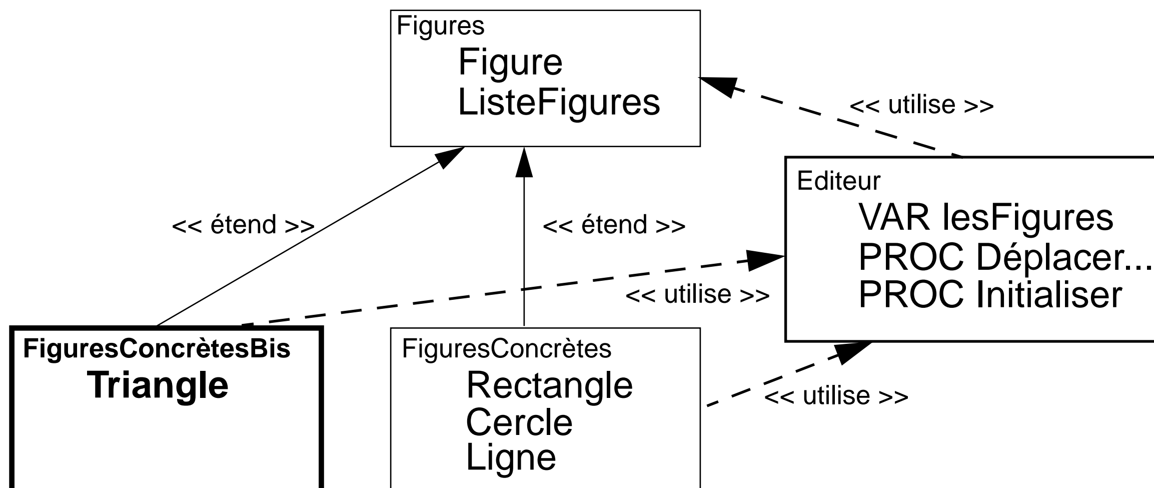
Module *FiguresConcrètes*

```
MODULE Figuresconcrètes;

IMPORT Figures, XYplane, Editeur;

TYPE Rectangle* = POINTER TO EXTENSIBLE RECORD(Figures.Figure)
                        l, h: INTEGER;
                END;

                Droite* = POINTER TO EXTENSIBLE RECORD(Figures.Figure)
                        dx, dy: INTEGER;
                END;

                Cercle* = POINTER TO EXTENSIBLE  RECORD(Figures.Figure)
                        r: INTEGER;
                END;

VAR r: Rectangle;
        c:Cercle;
        d: Droite;

PROCEDURE(r: Rectangle)Dessiner*, EXTENSIBLE;
VAR i : INTEGER;
BEGIN
   FOR i := r.x0 TO r.x0 +r. l DO
                XYplane.Dot(i,r.y0,XYplane.draw);
                XYplane.Dot(i,r.y0 + r.h,XYplane.draw);
  END;
  FOR i := r.y0 TO r.y0 +r. h DO
                XYplane.Dot(r.x0,i,XYplane.draw);
                XYplane.Dot(r.x0 +r. l,i,XYplane.draw);
  END;
END Dessiner;

PROCEDURE(r: Rectangle)InitRectangle*(VAR listef: Figures.ListeFigures; x0, y0, l,
h: INTEGER), NEW;
BEGIN
        r.InitFigure(listef,x0, y0); r.l:=l; r.h:=h;
        r.Dessiner;
END InitRectangle;
```

Module *FiguresConcrètes* (suite)

```
PROCEDURE(d: Droite)Dessiner*, EXTENSIBLE;
VAR i: INTEGER;  p: REAL;
BEGIN
        p:=d.dy / d.dx;
        FOR i:=d.x0 TO d.x0 + d.dx  DO
                XYplane.Dot(i,d.y0 + SHORT(ENTIER(p *(i-d.x0))) ,XYplane.draw);
        END;
END Dessiner;

PROCEDURE(d: Droite)InitDroite*(VAR listef: Figures.ListeFigures; x0, y0, x1, y1:
INTEGER), NEW;
VAR origX, origY, arrX, arrY: INTEGER;
BEGIN
        d.InitFigure(listef, x0, y0); d.dx:=x1-x0; d.dy:=y1-y0;
        d.Dessiner;
END InitDroite;

PROCEDURE(c: Cercle)Dessiner*, EXTENSIBLE;
CONST c1 = 256; c2 = 128;
VAR x, y, r: INTEGER;
BEGIN
        r:=c.r*c2; x:=r; y:=0; r:=r-1;
        REPEAT XYplane.Dot(x DIV c2 + c.x0, y DIV c2 + c.y0,XYplane.draw);
                x:=x-y DIV c1; y:=y + x DIV c1;
        UNTIL (x>=r) & (y<=0)
END  Dessiner;

PROCEDURE(c: Cercle)InitCercle*(VAR listef: Figures.ListeFigures; x0, y0, r:
INTEGER), NEW;
VAR origX, origY, arrX, arrY: INTEGER;
BEGIN
        c.InitFigure(listef, x0, y0); c.r:=r;
        c.Dessiner;
END InitCercle;

PROCEDURE DessinInitial*;
BEGIN
        NEW(r); r.InitRectangle(Editeur.lesFigures,10,10,30,50);
        NEW(c); c.InitCercle(Editeur.lesFigures,140,140,20);
        NEW(d); d.InitDroite(Editeur.lesFigures,60,60,100,120);
END DessinInitial;

END Figuresconcrètes.
```

Ajout de la classe *Triangle*



- pour ajouter les triangles, il suffit de compiler le module “FiguresConcrètesBis”
- pas nécessaire de recompiler les autres modules !
- pas nécessaire d’arrêter l’execution de l’éditeur !

```
MODULE Figuresconcrètesbis;

IMPORT Figures, XYplane, Editeur;

TYPE Triangle* = POINTER TO EXTENSIBLE RECORD(Figures.Figure)
                        x1, y1, x2, y2: INTEGER;
                END;
VAR t: Triangle;

PROCEDURE(t: Triangle)Dessiner*, EXTENSIBLE;
VAR i: INTEGER;  p: REAL;
BEGIN
        p:=(t.y1-t.y0) /(t. x1-t.x0);
        FOR i:=t.x0 TO t.x1  DO
                XYplane.Dot(i,t.y0 + SHORT(ENTIER(p *(i-t.x0))) ,XYplane.draw);
        END;
        p:=(t.y2-t.y0) /(t. x2-t.x0);
```

La classe *Triangle* (suite)

```
        FOR i:=t.x0 TO t.x2  DO
                XYplane.Dot(i,t.y0 + SHORT(ENTIER(p *(i-t.x0))) ,XYplane.draw);
        END;
        p:=(t.y1-t.y2) /(t. x1-t.x2);
        FOR i:=t.x2 TO t.x1  DO
                XYplane.Dot(i,t.y2 + SHORT(ENTIER(p *(i-t.x2))) ,XYplane.draw);
        END;
END Dessiner;

PROCEDURE(t: Triangle)InitTriangle*(VAR listef: Figures.ListeFigures; x0, y0,  x1,
y1, x2, y2: INTEGER), NEW;
BEGIN
        t.InitFigure(listef, x0, y0); t.x1:=x1; t.y1:=y1; t.x2:=x2; t.y2:=y2;
        t.Dessiner;
END InitTriangle;

PROCEDURE (t: Triangle)Déplacer*(dx, dy: INTEGER);
(* Remarque: la méthode Déplacer est redéfinie car la manière la plus simple de
déplacer un triangle est d'ajouter le déplacement dx et dy à ses trois sommet. On
invoque la méthode Déplacer définie dans la classe Figures (super call) pour
déplacer l'origine puis on déplace les deux autres sommets en leur ajoutant dx et
dy*)
BEGIN
        t.Déplacer^(dx, dy);
        t.x1:=t.x1 + dx; t.y1:=t.y1 + dy;
        t.x2:=t.x2 + dx; t.y2:=t.y2 + dy;
END Déplacer;

PROCEDURE DessinInitial*;
BEGIN
        NEW(t); t.InitTriangle(Editeur.lesFigures,10,80,60,80, 30,120);
END DessinInitial;

END Figuresconcrètesbis.
```