CENTRALE
L Y O N

GL et POO

CENTRALE
L Y O N

La réutilisation (1)

Quand nous commençons à écrire un nouveau programme (par exemple un compilateur) nous nous posons la question suivante :

Quel mécanisme de gestion de table allons-nous programmer?

au lieu de nous poser la question :

Quel mécanisme de gestion de table allons-nous utiliser?

L'industrie du logiciel ne s'appuie pas assez sur la sous-traitance de composants logiciels.

Pourquoi la réutilisation a mise longtemps pour devenir une réalité industrielle ?

CENTRALE LYON

La réutilisation (2)

- **PROBLEMES DE REUTILISABILITE**
  - **Problèmes d'organisation et de gestion :**
    - **Convertir les informaticiens à la réutilisation**
    - **Rangement, indexation et recherche: catalogues, bibliothèques, langages de requête, systèmes experts...**
    - **Accès aux composants: réseaux, chargement à distance**
  - **Coût : comment faire payer l'utilisation des composants ?**
  - **Problèmes techniques : quels composants ?**
    - **Réutilisabilité du code, mais: obstacle économiques (protection), politiques et techniques.**
    - **Bibliothèques de sous-programmes (ex.: bibliothèques mathématiques)**
      - **Paquetages (Ada, Modula 2)**
      - **Surcharge et généricité**

BTD/GL/POO 3

CENTRALE LYON

La réutilisation (3)

- **Les limitations des sous-programmes**

  Les sous-programmes ne conviennent que si les conditions suivantes sont satisfaites :

  - **Un ensemble de problèmes bien définis**
  - **Chaque cas de chacun des problèmes peut être caractérisé par un petit nombre de paramètres**
  - **Chaque problème est séparé; pas de sous-ensembles communs**
  - **Pas de structures de données complexes**

  Exemples : certains domaines du logiciel numérique
  Bibliothèques : NAG, ...

BTD/GL/POO 4

CENTRALE LYON

La réutilisation (4)

- **Les problèmes techniques**

Exemple: un sous-programme de recherche

```
Recherche (x : ELEMENT, t : TABLE_D_ELEMENTS)
return boolean is
   -- Recherche l'éléments x dans la table t
   pos : POSITION
begin
   pos := POSITION_INITIALE (x,t);
   while
            not EPUISE (pos, t)
            and then not TROUVE (pos, x, t)
   do
            pos := SUIVANT (pos, x, t) ;
   end ;
   return not EPUISE (pos, t)
end -- Recherche
```


CENTRALE LYON

La réutilisation (5)

Algorithme commun

```
from    gauche
until   hors_limite or else trouvé
loop    avancer ;
end ;
Resultat := trouvé
```

Recherche séquentielle: variantes d'implémentation

	Tableau	Liste chaînée	Fichier séquentiel
Début de la recherche : gauche	i := 1	l :=tête	rembobiner
Position suivante : avancer	i := i+1	l := l.suivant	lire (val)
Comparaison : trouvé	t [i] = x	l.valeur = x	val
Test de fin : hors_limite	i > taille	l = null	eof

CENTRALE LYON

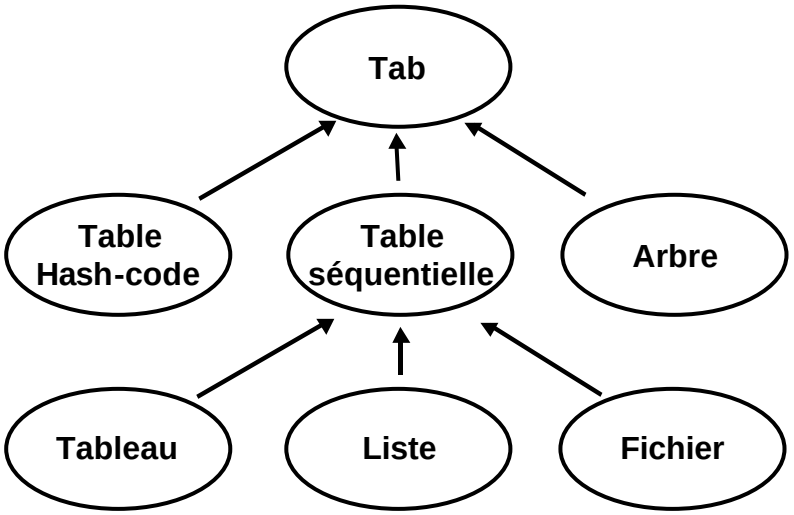
La réutilisation (6)

- **Problèmes si l'on souhaite obtenir un modèle général de recherche:**
  1. **Variations de types :**
     Que sont les éléments de la table ?
  2. **Variations de représentation :**
     Variations dans des choix de structures de données et d'algorithmes: tables séquentielles (triées ou non), tableaux, arbres binaires de recherche, fichiers,...
  3. **Groupes de sous-programmes :**
     Un sous-programme de recherche ne suffit pas: il doit être couplé avec des sous-programmes de création de table, d'insertion, de destruction, etc.
  4. **Indépendance vis-à-vis de la représentation :**
     Peut-on demander une opération telle que la recherche sans savoir quelle implémentation est utilisée de façon interne? Recherche (y, t1)
  5. **Eléments communs :**
     Comment l'auteur du module peut-il tirer parti des nombreux points communs qui existent entre les éléments d'un sous-ensemble des implémentations possibles? Par exemple: toutes les tables séquentielles.

BTD/GL/POO 7

CENTRALE LYON

La réutilisation (7)



BTD/GL/POO 8

CENTRALE LYON

La réutilisation (8)

- **LES LANGAGES A ENCAPSULATION**

ADA, Modula-2, CLU, ...

Regrouper un ensemble de sous-programmes ayant un but commun par exemple recherche, insertion, destruction etc., avec les descriptions des structures de données associées.

Paquetage (Ada), Module (Modula-2), Grappe [Cluster (CLU), sous-programmes à entrées multiples (Fortran),...
Ceci répond au problème des groupe de sous-programmes (3).

Avantages :

- **Pour le programmeur du module : on fabrique tout au même endroit. Ceci simplifie la gestion de configurations, les changements éventuels d'implémentation, l'addition de nouvelles primitives.**
- **Pour le programmeur du client : on achète tout au même endroit. Ceci simplifie la recherche de sous-programmes existants, les demandes d'extensions.**

BTD/GL/POO 9

CENTRALE LYON

La réutilisation (9)

- **VARIATIONS DE TYPES : LA GENERICITE**

Les modules CLU ou Ada peuvent être paramétrés par des paramètres génériques représentant des types.

Par exemple:

```
module Recherche_en_table [T] is
begin
        .....
        type Table = array [...] of T;

        Recherche ( x : T ; ta : Table ) : boolean is
        begin
                .... Rechercher x dans ta .....
        end ; --Recherche
        .....
end;  -- module Recherche_en_table
```

BTD/GL/POO 10

CENTRALE LYON

La réutilisation (10)

- **INSTANCIATION D'UN MODULE GENERIQUE :**

```
module Recherche_d_entiers is
   new Recherche_en_table [INTEGER];
type Table_d_entiers = Recherche_d_entiers.Table ;
   t : Table_d_entiers ; n : INTEGER ; b : BOOLEAN ;
   .......
   b := Recherche_d_entiers.Recherche (x , t) ;
```

La généricité permet à l'implémenteur **du module d'écrire un seul module pour tous les cas de la même implémentation d'une certaine abstraction, appliquée à divers types d'objets.**

CENTRALE LYON

La réutilisation (11)

- **SURCHARGE**

Introduite par Algol 68 et utilisée en CLU et Ada, la surcharge permet de donner le même nom à différents sous-programmes.

Dans un langage typé statiquement, les ambiguïtés peuvent être levées par le compilateur si les variantes sont différenciées par les types des paramètres.

Par exemple: différents sous-programmes de recherche (recherche en liste linéaire, en adressage associatif, arbre binaire de recherche etc.) peuvent tous porter le nom Recherche.

```
Recherche ( x : INTEGER ; t : H_TABLE )
Recherche ( x : INTEGER ; t : TABLE_CHAINEE )
Recherche ( x : INTEGER ; t : TABLE_BINAIRE )
```

Pour résoudre l'ambiguïté dans le cas d'un appel donné: déterminer le type de t1 **dans Recherche (y , t1)**

La surcharge permet aux programmeurs des clients d'écrire le même code lorsqu'ils utilisent des implémentations différentes de la même abstraction.

CENTRALE LYON

La réutilisation (12)

INTERET ET LIMITES DE LA GENERICITE ET DE LA SURCHARGE

La généricité et la surcharge correspondent aux problèmes de variation de types et de variation de représentation. Mais ces techniques ne vont pas assez loin en matière de réutilisabilité et d'extensibilité :

- **Ni l'une ni l'autre n'apporte de solution au problème des éléments communs. Seuls deux niveaux de modules génériques, paramétrés et non paramétrés, sont fournis. Pas de modules "partiellement implémentés".**
- **Ni l'une ni l'autre n'apporte de solution au problème de l'indépendance vis-à-vis de la représentation : l'implémenteur d'un module générique, comme client qui utilise un sous-programme surchargé, doivent savoir exactement quelle version est utilisée à chaque fois.**

Les techniques et langages à objets apportent une réponse systématique aux cinq problèmes de la réutilisabilité.

BTD/GL/POO 13

CENTRALE LYON

La réutilisation (13)

- **Aspects de la réutilisation**

1. Variations de types : <u>généricité</u>

Que sont les éléments de la table ?

2. Variations de représentation :

Variations dans des choix de structures de données et d'algorithmes: tables séquentielles (triées ou non), tableaux, arbres binaires de recherche, fichiers,...

3. Groupes de sous-programmes : <u>modularité</u>

Un sous-programme de recherche ne suffit pas: il doit être couplé avec des sous-programmes de création de table, d'insertion, de destruction, etc.

4. Indépendance vis-à-vis de la représentation : <u>surcharge</u>

Peut-on demander une opération telle que la recherche sans savoir quelle implémentation est utilisée de façon interne? Recherche (y, t1)

5. Eléments communs :

Comment l'auteur du module peut-il tirer parti des nombreux points communs qui existent entre les éléments d'un sous-ensemble des implémentations possibles? Par exemple: toutes les tables séquentielles.

BTD/GL/POO 14

CENTRALE LYON

L'approche par les objets

- **SEPT ETAPES VERS LES OBJETS**
  1. **Modules construits autour des objets**
  2. **Objets comme mises en œuvre de types abstraits de données**
  3. **Objets dynamiques : récupération automatique de la mémoire**
  4. **CLASSE : MODULE et TYPE**
  5. **Héritage simple (linéaire)**
  6. **Opérations retardées et liaison dynamique**
  7. **Héritage multiple**

BTD/GL/POO 15

CENTRALE LYON

1. MODULARISATION AUTOUR DES DONNEES

- **Un programme effectue des actions sur des structures de données**
- **La structure du programme peut être fondée sur les actions ou sur les données.**
  - **Décomposition classique : sur les traitements (sous-programmes)**
  - **Décomposition par les objets**

BTD/GL/POO 16

CENTRALE LYON

DECOMPOSITION PAR LES OBJETS

- **Compatibilité : on ne peut combiner les actions si les données sont incompatibles.**
- **Réutilisabilité : il faut pouvoir réutiliser des structures de données entières, non pas seulement les opérations.**
- **Extensibilité (continuité) : les objets restent plus stables dans le temps**

BTD/GL/POO 17

CENTRALE LYON

CONCEPTION-PROGRAMMATION PAR OBJETS

- **UNE PREMIERE DEFINITION**
  - **La conception par objets est la méthode de construction de logiciel qui fonde l'architecture des systèmes sur les objets qu'ils manipulent et non sur "la" fonction qu'ils assurent.**

    Ne demande pas d'abord que fait le système :

    Demander pour QUI il le fait !

BTD/GL/POO 18

CENTRALE LYON

CONCEPTION PAR OBJETS

- **Comment trouver les objets**
- **Comment décrire les objets**
- **Comment décrire les relations entre objets et les éléments communs à certaines catégories d'entre eux**
- **Comment utiliser les objets pour structurer les programmes**

Pour décrire les objets :

1. Considérer non pas un seul objet mais une CLASSE d'objets logiquement liés.

2. Définir les objets non par leur représentation mais par leur comportement externe, c'est-à-dire par les SERVICES qu'ils offrent au monde extérieur.

CENTRALE LYON

2. LA BASE THEORIQUE

LES TYPES ABSTRAITS DE DONNEES

- **Problème principal :**
  Comment décrire les objets des programmes (structures de données)
  - **complètement**
  - **sans ambiguïté**
  - **sans surspécification**

Exemple: Une pile, objet concret

```
dernier := dernier + 1
tab [ dernier ] := x

dernier := dernier - 1
tab [ dernier ] := x

new ( p ) ;
p.suivant := tête ; p.val := x ;
tête := p
```

CENTRALE LYON

LES PILES, TYPE ABSTRAIT

TYPES

PILE [X]

FONCTIONS (Opérations)
vide : PILE [X] -> BOOLEAN
nouvelle : -> PILE [X]
empiler : X * PILE [X] -> PILE [X]
dépiler : PILE [X] -\-> PILE [X]
sommet : PILE [X] -\-> X

PRECONDITIONS
pré dépiler (s : PILE [X]) = (non vide (s))
pré sommet (s : PILE [X]) = (non vide (s))

AXIOMES
Pour tous x : X, p : PILE [X]
vide (nouvelle ( ))
non vide (empiler (x , p))
sommet (empiler (x , p)) = x
dépiler (empiler (x , p)) = p

BTD/GL/POO 21

CENTRALE LYON

APPLICATION A LA CONCEPTION DES ARCHITECTURES LOGICIELLES

- **Les types abstraits fournissent une base idéale pour modulariser les logiciels**
  - **Principe : chaque module est construit sur la base d'un type abstrait, c'est-à-dire sur un ensemble de structures de données décrites par les opérations qui font partie de son interface officielle. L'interface est définie par un ensemble d'opérations (implémentant les fonctions du type abstrait) sous des contraintes formellement définies (les axiomes et préconditions).**
  - **Le module consiste en une représentation du type abstrait et en une implémentation pour chaque opération. Des opérations auxiliaires peuvent aussi être présentes.**
  - **La programmation par objets est la construction de systèmes logiciels sous forme de collections structurées d'implémentations éventuellement partielles de types abstraits de données.**

BTD/GL/POO 22

CENTRALE LYON

LA STRUCTURE DE BASE DES LANGAGES

LA CLASSE

Une classe est à la fois:

- un module

- un type

Une grande partie de la puissance conceptuelle de la méthode vient de la fusion de ces deux notions.

Du point de vue du module :

- **ensemble de services (primitives, "features")**
- **masquage de l'information**
- **les classes peuvent être clientes les unes des autres**

Du point de vue du type :

- **utilisée pour déclarer des entités (" variables)**
- **possibilité de vérification de types**
- **notion de sous-type**

BTD/GL/POO 23

CENTRALE LYON

TERMINOLOGIE (1)

- **Une classe est une implémentation d'un type abstrait de données**
  - **Des INSTANCES de la classe peuvent être créées à l'exécution, ce sont des OBJETS.**
  - **Tout objet est une instance de classe**
  - **Une classe est caractérisée par des PRIMITIVES. Les primitives comprennent les ATTRIBUTS (qui représentent les champs des instances de la classe) et les ROUTINES (qui représentent les opérations sur les instances).**
    Les routines sont divisées en PROCEDURES (qui ont un effet sur l'instance, mais ne produisent pas de résultat), et FONCTIONS (qui ont un résultat, mais normalement pas d'effet).
    Toute opération (application d'une routine, accès à un attribut) est relative à une instance particulière, l'INSTANCE COURANTE de la classe.

BTD/GL/POO 24

CENTRALE L Y O N

TERMINOLOGIE (2)

- **PRIMITIVES**

  ATTRIBUTS

  ROUTINES

  PROCEDURES

  FONCTIONS

- **Autre terminologie (Smalltalk): variable d'instance (attribut); méthode (routine); passage de message (appel d'une routine sur un objet).**

BTD/GL/POO 25

CENTRALE L Y O N

ENREGISTREMENTS ET REFERENCES

- **Un objet peut contenir différents champs**
- **Ces champs sont des champs simples**
  Un objet ne peut pas contenir d'autres objets car on aurait un problème de partage
- **Notion de référence pour partager des objets**
- **Création dynamique des instances: la déclaration d'une instance (**p1: POINT**) ne conduit pas à l'allocation de la mémoire.**
- **Création est faite de façon explicite par l'opération Create.**
- **La référence peut prendre deux états: VOID CREATED**
- **Deux opérations permettent de passer d'un état à l'autre:** p1.Create, P1.Forget
- **Accès aux champs utilise la notion d'instance courante et de la notation pointée pour d'autres instances**

BTD/GL/POO 26

CENTRALE LYON

UNE CLASSE SIMPLE (en notation Eiffel)

```
class POINT export
        x, y, translation, homothétie, distance
    feature
        x, y : REAL ;
        translation (a, b : REAL) is
            -- Translater de a horizontalement
            --                    b verticalement
            do      x := x+a ;
                    y := y+b
            end ;  -- translation

        homothétie (facteur : REAL) is
            -- Homothétie de facteur
            do      x := facteur * x ;
                    y := facteur * y
            end ;  -- homothétie

        distance (p : POINT) : REAL is
            -- Distance du point courant à p
            do      Result := sqrt ((x-p.x)^2+(y-p.y)^2)
            end  -- distance
        end  -- class POINT
```

BTD/GL/POO

27

CENTRALE LYON

UTILISATION DE LA CLASSE (dans une autre)

```
class GRAPHIQUE export
        .............
    feature
        p1, p2 : POINT ;
        r, s : REAL;
        .............
        une_routine is
            do
                    ..........
                    p1.Create ; p2.Create ;
                    p1.translation (3.5, -2.4) ;
                    p1.homothétie (2*r) ;
                    r := p1.distance (p2) ;
                    p2 := p1 ;
                    s := p1.x ;
                    p2.homothétie (-3.0)
                    ..........
            end ;  -- une_routine

        ...........
        end  -- class GRAPHIQUE
```

BTD/GL/POO

28

CENTRALE LYON

LES ASSERTIONS (1)

- **But :** **Assurer la fiabilité (validité et robustesse), non pas seulement l'extensibilité et la réutilabilité.**

Idée de base : **Ecrire du logiciel correct en sachant pourquoi il l'est.**

Notion de Programmation par Contrat.

BTD/GL/POO 29

CENTRALE LYON

LES ASSERTIONS (2)

- **Solution :** **Inclure une partie de la spécification dans le code lui-même.** **Exemple la classe "Compte"**

```
class COMPTE export
        déposer, retrait_possible, retirer, solde, solde_minimum
feature solde : INTEGER ; solde_minimum : INTEGER is 1000 ;
        titulaire : STRING ;
        ajouter (somme : INTEGER) is
                -- Ajouter somme au compte
                -- (Procédure secrète)
                do          solde := solde + somme  end ;  -- ajouter
        déposer (somme : INTEGER) is
                -- Déposer somme sur le compte
                do          ajouter (somme)
                end ;  -- déposer
        retirer (somme : INTEGER) is
                -- Retirer somme du compte
                do          ajouter(- somme)  end ;  -- retirer
        retrait_possible (somme : INTEGER) : BOOLEAN  is
                -- Est-il possible de retirer somme
                -- du compte ?
                Do          Result := (solde >= solde_minimum + somme)
                end ;  -- retrait_possible
end  -- class COMPTE
```

BTD/GL/POO 30

CENTRALE LYON

UTILISATION D'ASSERTIONS

- **PRECONDITIONS, POSTCONDITIONS ET INVARIANTS**

```
class COMPTE export   ..... (comme précédemment)
feature  -- Attributs comme précédemment:solde, sole_minimum
        ajouter .....  --comme précédemment
        déposer (somme : INTEGER ) is  -- Déposer somme sur le compte
                require     somme >=0
                do          ajouter (somme)
                ensure      solde = old solde + somme
                end ;  -- déposer
        retirer (somme : INTEGER) is   -- Retirer somme du compte
                require     somme >= 0 ;
                            somme <= solde -solde_minimum
                do          ajouter (-somme)
                ensure      solde = old solde - somme
                end ; -- retirer
        retrait_possible .....  -- comme précédemment
        Create (initial : INTEGER) is
                require     initial >= solde_minimum
                do          solde := initial
                end  --  Create
        invariant
                solde >= solde_minimum
        end  -- class COMPTE
```

BTD/GL/POO 31

CENTRALE LYON

A QUOI SERVENT LES ASSERTIONS ?

- 1. **Production de programmes corrects**
  2. **Documentation**

  ---

  3. **Aide à la mise au point**
  4. **Traitement d'exceptions**

  Options de compilation (par classe):

  - NO_ASSERTION_CHECK (pas de protection)
  - PRECONDITIONS (préconditions seulement)
  - ALL_ASSERTIONS (tout vérifier)

BTD/GL/POO 32

CENTRALE LYON

Spécification d'une fonction de calcul de la racine carrée

```
srqt ( x, epsilon : REAL ) is
                -- Racine carrée de x, précision epsilon
                require
                        x>= 0 ;
                        epsilon >= 10^(-6)
                do
                        .............
                ensure
                        abs (Result^2 - x) <= epsilon^2
        end sqrt
```

L'appel est un contrat

Programmeur client :

Obligations : N'appeler la routine que si x >= 0, epsilon >= 10^{-6}

Bénéfices : Obtenir en retour l'approximation souhaitée de la racine carrée

Implémenteur du module :

Obligations : Renvoyer l'approximation demandée

Bénéfices : Inutile de traiter les cas x > 0, ouepsilon <= 10^{-6}

CENTRALE LYON

Un autre exemple de contrat

```
class TABLE [ T ] export
                insérer, valeur, détruire, ...
        feature
                insérer (élément : T, clé : STRING) is -- Insérer élément, associé à clé
                        require     nb_éléments < max_éléments
                        do          ... Algorithme d'insertion ...
                        Ensure      nb_éléments <= max_éléments;
                                    valeur (clé) = élément ;
                                    nb_éléments = old nb_éléments + 1
                end ; -- insérer
                .... Autres déclarations de primitives ...
        Invariant   0 <= nb_élément ;
                    nb_élément <= max_éléments
end -- class TABLE
```

Client :

Obligations : Appeler insérer **seulement sur une table non pleine**

Bénéfices : Obtenir une table où x **a été associé à** clé

Maître d 'œuvre :

Obligations : Insérer x **pour qu'il puisse être retrouvé par** clé

Bénéfices : Inutile de se préoccuper du cas d'une table pleine avant une insertion

CENTRALE LYON

Règles d'utilisation d'assertions

- **Le code d'une routine ne doit PAS contenir de tests de la précondition, comme dans:**

```
sqrt ( x, epsilon : REAL) : REAL is
                require     x>=0 ; epsilon 10^-6
                do          if x<0 then …
                            elsif epsilon < 10^-6 then ...
                            else … CAlcul normal de la racine ... end
                ensure
                            abs (Result^2 - x) <= epsilon^2
        end sqrt
```

```
        insérer ( élément : T, clé : STRING) is
                require     nb_éléments < max_éléments
                do          if nb_éléments >= max_éléments then
                                    ... Traiter le cas d'une table pleine ...
                            else
                                    ... Insertion normale ...
                ensure
                            ... comme précédemment  ...
        end;  -- insérer
```

BTD/GL/POO 35

CENTRALE LYON

LES CLASSES GENERIQUES

- **Comment réconcilier la flexibilité avec la sécurité**
- **Comment définir des structures de données cohérentes, par exemple une pile d'entiers, une pile de points ?**
- **Langages non typés (ou typés dynamiquement à (Smalltalk, Objective-C): définir une classe générale PILE; vérifier à l'exécution.**
- **Langages typés (Eiffel, Trellis-Owl): définir une classe paramétrée (ou "générique"): PILE [T] .**

BTD/GL/POO 36

CENTRALE LYON

Utilisation de la classe générique

```
pile_entiers: PILE [ INTEGER ] ; pile_points: PILE [ POINT ] ;

n : INTEGR ; p1 : POINT ;

pile_entiers.Create ; pile_points.Create ;

pile_entiers.empiler  (3) ; pile_entiers.empiler (-5) ;

n := pile_entiers.sommet ;

pile_points.empiler (p1) ; .....
```

Typage statique : le compilateur refusera

```
pile_points.empiler (3) ;

pile_entiers.empiler (p1)
```

BTD/GL/POO 37

CENTRALE LYON

LA NOTION DE TYPAGE DANS UN CONTEXTE A OBJETS

- **Le but du typage fort est de respecter la contrainte suivante :**

  Un langage à objets est dit statiquement typé si pour toute occurrence de la notation x.f **ou équivalent, dans un système correct, le compilateur garantit que tout objet associé avec** x **à l'exécution possédera au moins une primitive correspondant à** f.

BTD/GL/POO 38

CENTRALE LYON

CLASSE GENERIQUE PILE (1)

```
class PILE [ T ]
        -- Piles
        export      nb_éléments, vide, pleine, sommet, empiler,
                    dépiler, nettoyer, chager_sommet
        feature     nb_éléments: INTEGER is          -- Nombre d'éléments insérés
                            do .....
                            end;  -- nb_éléments
                    vide : BOOLEAN is       -- La pile est-elle vide ?
                            Do              Result  := (nb_éléments = 0)
                            ensure          Result  = (nb_éléments = 0)
                            end ;  -- vide
                    pleine : BOOLEAN is     -- La pile est-elle pleine ?
                            do .....        end ;  -- pleine
                    sommet : T is           -- Dernier élément empilé
                            require         not vide
                            do …            end ;  -- sommet
                    empiler (x : T) is      -- Ajouter x sur la pile
                            require         not pleine
                            do ...
                            Ensure          not vide ; sommet = x ;
                                            nb_éléments = old nb_éléments + 1
                    end ;  -- empiler
```


CENTRALE LYON

CLASSE GENERIQUE PILE (2)

```
                    dépiler is      -- Retirer l'élément au sommet
                            require         not vide
                            do ...
                            ensure          not pleine ;
                                            nb_éléments = old nb_éléments -1
                    end ; -- dépiler
                    changer_sommet (x : T) is     -- Remplacer l'élément au sommet par x
                            require         not vide
                            do              dépiler ; empiler (x)
                            ensure
                                            not vide ; sommet = x ;
                                            nb_éléments = old nb_éléments
                    end ;  -- changer_sommet
                    nettoyer is             -- Retirer tous les éléments
                            do ...
                            ensure
                                            vide
                    end ;  -- nettoyer
        invariant
                    nb_éléments >= 0
end  -- class PILE
```

CENTRALE L Y O N

Une classe générique tableau

– **UTILISATION DES TABLEAUX**

```
a : ARRAY [ REAL ] ;
a.Create (1, 300) ;
a.enter (25, 3.5) ;                -- en PASCAL: a[25]:= 3.5
x := a.entry (i);                  -- en PASCAL: x:= a[i]
```

Il existe également une classe ARRAY2 [T] pour des tableaux à deux dimensions.

```
class ARRAY [T] export  lower, size, upper, entry, enter
        feature  lower: INTEGER ; upper: INTEGER ; size : INTEGER ;
                Create (min: INTEGER, max: INTEGER) is
                  -- Allouer le tableau avec les bornes min et max
                        do ..... end ;
                entry (i: INTEGER): T is  -- Elément d'index i
                        require      lower <= i ; i <= upper
                                     do ..... end ;
                enter (i: INTEGER, v : T) is
                    -- Affecter la valeur de v à l'élément d'index i
                        require      lower <= i ; i <= upper
                        do ..... end ;
        invariant    size = upper - lower + 1
end  -- class ARRAY [ T ]
```


CENTRALE L Y O N

L'HERITAGE

- **Une technique fondamentale pour la réutilisabilité**
- **<u>Principe :</u> décrire une classe non pas à partir de zéro, mais par extension ou spécialisation d'une classe existante - ou de plusieurs dans le cas de l'<u>héritage multiple.</u>**
  - **<u>Du point de vue du module :</u> si B hérite de A, tous les services de A sont disponibles dans B (éventuellement avec une implémentation différente).**
  - **<u>Du point de vue du type :</u> l'héritage est la relation "est-un". Si B hérite de A, toute instance de B est une instance de A.**

  TERMINOLOGIE :
  Héritier (sous-classe), parent (sur-classe), descendant, ancêtre

CENTRALE LYON

Un exemple de l'héritage: POLYGONE, RECTANGE, CARRE …(1)

```
class POLYGONE export nombre_de_côtés, périmètre, ...
        Feature     nombe_de_côtés : INTEGER ;
                    sommet : ARRAY [ POINT ] ;

        Create is   do .....    end ; --  Create
        périmètre : REAL  is            -- Longueur du périmètre
                    local       i : INTEGER, p1 : POINT ; p2 : POINT
                    do          from    i := 1
                                until   i > nombre_de_côtés
                                loop    p1 := sommet.entry (i) ;
                                        if i= nombre_de_côtés then
                                                p2 := somme.entry (1)
                                        else    p2 := sommet.entry (i+1)
                                        end;
                                        Result  := Result + p1.distance (p2)
                                end
                    end -- périmètre
                    ..... autres primitives
        and  -- class POLYGONE
```

BTD/GL/POO

43

CENTRALE LYON

Un exemple de l'héritage: POLYGONE, RECTANGE, CARRE …(2)

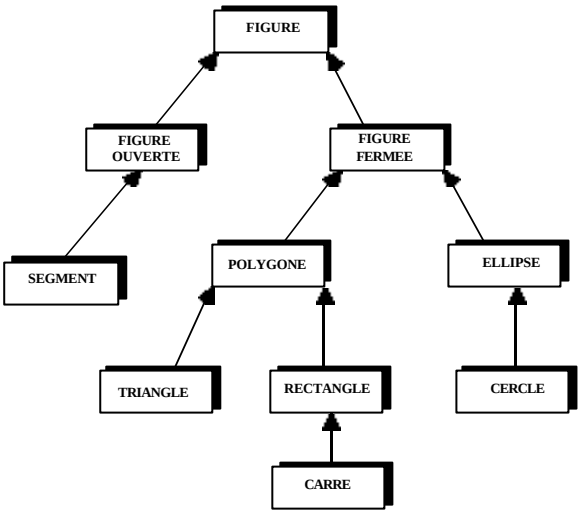
```
        class RECTANGLE export
                    périmètre, côté1, côté2, diagonale, ...
        inherit
                    POLYGONE
                            redefine périmètre
        feature
                    côté1, côté2, diagonale : REAL;
                    ....
                    périmètre: REAL is
                                    -- Longueur périmètre
                            do
                                    Result  := 2 * ( côté1 + côté2 )
                            end  -- périmètre
        invariant
                    nombre_de_côtés = 4 ;
                    sommet.entry(1).distance(sommet.entry(2)) = côté1;
                    sommet.entry(3).distance(sommet.entry(4)) = côté1;
                    ......
        end  -- class RECTANGLE
```

BTD/GL/POO

44

CENTRALE
L Y O N

UNE HIERARCHIE DE CLASSES



BTD/GL/POO

45

CENTRALE
L Y O N

TYPAGE, POLYMORPHISME ET LIAISON DYNAMIQUE

En supposant :

```
p : POLYGONE ; r : RECTANGLE ; x : REAL
p.Create (.....) ; r.Create (.....) ;
```

Autorisé:

```
x := p.périmètre ;
x := r.périmètre ;
x := r.diagonale ;
p := r ;
x := p.périmètre ;
```

Interdit:

```
x := p.diagonale        -- pas de diagonale pour POLYGONE
r := p                  -- tout POLYGONE n'est pas RECTANGLE
```

POLYMORPHISME: p **peut changer de forme à l'exécution**

LIAISON DYNAMIQUE: l'effet de p.périmètre **dépend de la forme de** p **à l'exécution.**

BTD/GL/POO

46

CENTRALE LYON

Langage à objet

- **Un langage à objets est dit statiquement typé si pour toute occurrence de la notation** x.f **ou équivalent, dans un système correct, le compilateur garantit que tout objet associé avec** x **à l'exécution possédera au moins une primitive correspondant à** f.

  "au moins" : exprime le polymorphisme

BTD/GL/POO 47

CENTRALE LYON

HERITAGE ET TYPAGE

- **<u>Première règle de typage :</u>**

  Soit x **déclaré d'un type classe** C**. La notation** x.f **n'est correcte que si un ancêtre de la classe** C **contient une déclaration de la primitive** f **et si** C **exporte** f**.**

- **<u>Deuxième règle de typage :</u>**

  Soit x **déclaré d'un type classe** C **et** y **d'un type classe** D**. L'affectation** x:=y **n'est correcte que si** D **est un descendant de** C**. La même condition s'applique à toute utilisation de** y **comme argument effectif correspondant à l'argument formel** x**.**

BTD/GL/POO 48

CENTRALE LYON

Règles de typage

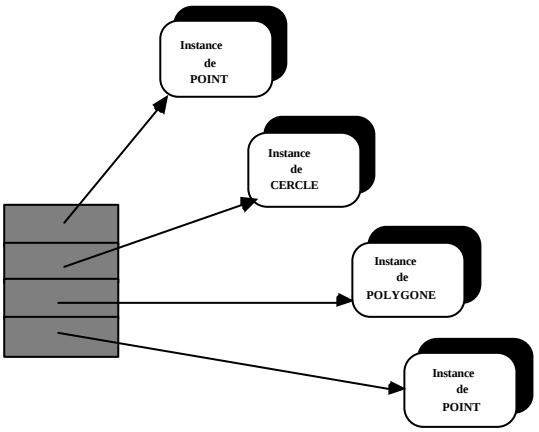
- **Type statique : type avec lequel x est déclaré.**
- **Type dynamique : type de l'objet associé à x à un instant de l'exécution**
- **Deuxième règle exprime que :**
  le typage dynamique doit être un descendant du typage statique.

BTD/GL/POO 49

CENTRALE LYON

UNE STRUCTURE DE DONNEES POLYMORPHE

PILE [FIGURE]



BTD/GL/POO 50

CENTRALE
L Y O N

QU'EST-CE QUE L'HERITAGE ?

- **<u>Point de vue du module</u> (réutilisabilité) :**

  Un module doit être ouvert et fermé

- **<u>Point de vue du type</u> :**

  Extensibilité: Adaptation automatique de l'opération à sa cible

BTD/GL/POO

51

CENTRALE
L Y O N

ROUTINES ET CLASSES RETARDEES

```
f : FIGURE ; c : CERCLE ; p : POLYGONE ;
v : VECTEUR ;
.........
c.Create (....) ; p.Create (....) ;

if .... then f := c else f := p end ;

f.translation (v) ; f.rotation (....) ; ..... ; ...
```

Pour se réconcilier avec le système de typage:

```
deferred class FIGURE export ....feature
        translation (v : VECTEUR) is
                deferred
                end;

        rotation (a : ANGLE ; p : POINT) is
                deffered
                end;          .....
end  --  class FIGURE

INTERDIT:          f.Create (....)
```

BTD/GL/POO

52

CENTRALE LYON

LES CLASSES RETARDEES :

- **Capturent des comportements communs**
- **Ont une ou plusieurs routines retardées (venant éventuellement d'un ancêtre, non redéfinies)**
- **Ne peuvent être instanciées**
- **Peuvent contenir des routines non retardées**
- **Jouent un rôle fondamental dans l'application de la méthode à objets au niveau de la conception**

Comparer avec la séparation interface/implémentation (ADA, Modula-2) :

- **Des éléments retardés et non retardés peuvent cohabiter**
- **Plus d'une implémentation peut être offerte à l'intérieur du même système**
- **Spécification formelle de la sémantique (assertions)**

BTD/GL/POO 53

CENTRALE LYON

Exemple : pour différentes implémentations des tables séquentielles

Algorithme commun

```
from    gauche
until   hors_limite or else trouvé
loop    avancer ;
end ;
Resultat := trouvé
```

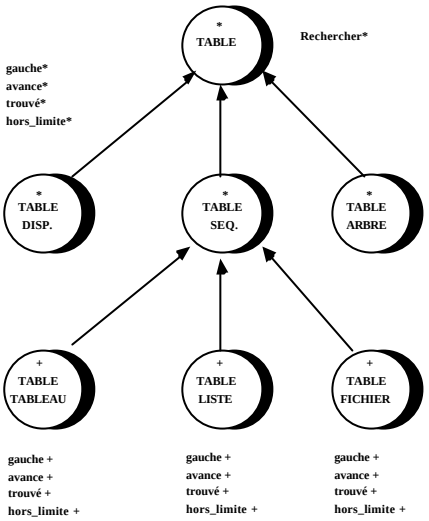
Recherche séquentielle : variantes d'implémentation

	Tableau	Liste chaînée	Fichier séquentiel
Début de la recherche : gauche	i := 1	l :=tête	rembobiner
Position suivante : avancer	i := i+1	l := l.suivant	lire (val)
Comparaison : trouvé	t [i] = x	l.valeur = x	val
Test de fin : hors_limite	i > taille	l = null	eof

BTD/GL/POO 54

CENTRALE
L Y O N

Hiérarchie de classes



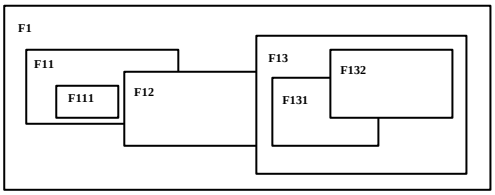
BTD/GL/POO

55

CENTRALE
L Y O N

HERITAGE MULTIPLE (1)

Exemple : **Fenêtres**



Primitives "graphiques": hauteur, largeur, changer_hauteur, changer_largeur, xpos, ypos, déplacer, etc..

Primitives "hiérarchiques": sur_fenêtre, sous_fenêtres, changer_sous_fenêtres, ajouter_sous_fenêtres, etc...

```
Solution:
        class FENETRE export
                ... Toutes les primitives sur les fenêtres
        inherit
                RECTANGLE ;
                ARBRE
        feature ...
        end -- class FENETRE
```

BTD/GL/POO

56

CENTRALE
L Y O N

HERITAGE MULTIPLE (2)

Offrir aux clients une terminologie localement bien adaptée.

```
Exemple:
        class FENETRE export
                ... Toutes les primitives significatives pour les clients
                ... de l'abstraction FENETRE
        inherit
                RECTANGLE ;
                ARBRE
                        rename
                                insérer_noeud as ajouter_sous_fenêtre,
                                changer_enfant as changer_sous_fenêtre,
                                etc .....
```

L'héritage est une technique d'implémentation

L'héritage multiple est le mariage de convenance.

CENTRALE
L Y O N

HERITAGE MULTIPLE (3)

Exemple

```
class PILE_FIXE [T] export ... inherit
        PILE [T] ;
        ARRAY [T]
feature
        .......
end  --class PILE_FIXE

class PILE_CHAINEE [T] export ... inherit
        PILE [T] ;
        LISTE_CHAINEE [T]
feature
        .......
end  --class PILE_CHAINEE

class LISTE_FIXE [T] export ... inherit
        LIST [T] ;
        ARRAY [T]
feature
        .......
and  --class LISTE_FIXE
```

CENTRALE LYON

Héritage multiple : problème de conflit des noms

- **Dans différentes classes on utilise les mêmes noms pour des "choses" différentes.**
- **Le problème des conflits de noms est un problème syntaxique**

Solution immédiate est le renommage:

```
class PARIS export .... inherit
        NEW_YORK
                rename foo as zoo
        LONDON
                rename foo as fog
feature
....
end -- class PARIS
```

BTD/GL/POO 59

CENTRALE LYON

Une autre justification de renommage

- **IMPLEMENTATIONS DE PILES**

```
class PILE_FIXE [T]
        -- piles avec taille physique fixe,
        -- représentées par des tableaux
export
        taille_max,
        nb_éléments, vide, pleine,
        sommet, empiler, dépiler, changer_sommet,
        nettoyer

inherit
        PILE [T]
                redefine changer_sommet ;
        ARRAY [T]
                rename
                        Create as tableau_Create,
                        size as taille_max
```

BTD/GL/POO 60

CENTRALE LYON

IMPLEMENTATIONS DE PILES (2)

```
feature
Create (n : INTEGER) is
        do      tableau_Create (1, n)
        ensure
                taille_max = n
        end ; -- Create

nb_éléments : INTEGER ;
        -- Nombre d'éléments insérés ;
        -- redéfini ici somme attribut

pleine : BOOLEAN is
                -- La pile est-elle vide ?
        Do      Result := (nb_éléments = taille_max)
        ensure
                Result = (nb_éléments = taille_max)
        end ;  -- pleine

sommet : T is
                -- Dernier élément empilé
        require not vide
        do
                Result := entry (nb_éléments)
        end ;  -- sommet
```

BTD/GL/POO 61

CENTRALE LYON

IMPLEMENTATIONS DE PILES (3)

```
empiler (x : T) is      -- Ajouter x sur la pile
        require
                not pleine
        do
                nb_éléments := nb_éléments +1 ;
                enter (nb_éléments, x)
        ensure
                not vide ; sommet = x ;
                entry (nb_éléments) = x ;
                nb_éléments = old nb_éléments + 1
        end ;  -- empiler

dépiler is
                -- Retirer l'élément au sommet
        require
                not vide
        do
                nb_éléments := nb_éléments - 1
        ensure
                not pleine ;
                nb_éléments := old nb_éléments - 1
        end ;  -- dépiler
```

BTD/GL/POO 62

CENTRALE LYON

IMPLEMENTATIONS DE PILES (4)

```
        changer_sommet (x : T) is
                        -- Remplacer l'élément au sommet par x
                require
                        not vide
                do
                        enter (nb_éléments, x)
                ensure
                        not vide ; sommet =  x ;
                        nb_éléments = old nb_éléments
                end ;  -- changer_sommet


        nettoyer is  -- Retirer tous les éléments
                do
                        nb_éléments := 0
                ensure
                        vide
                end ;  -- nettoyer

        invariant
                nb_éléments >= 0 ; nb_éléments < taille_max

end  -- class PILE_FIXE
```

BTD/GL/POO 63

CENTRALE LYON

IMPLEMENTATIONS DE PILES (5)

```
class PILE_CHAINEE [T]
        -- piles sans limitation de taille,
        -- représentées par des listes chaînées
export
        taille_max,
        nb_éléments, vide, pleine,
        sommet, empiler, dépiler, changer_sommet,
        nettoyer

inherit
        PILE [T]
                redefine changer_sommet ;
        LINKED_LIST [T]
                rename
                        vide as liste_vide,
                        nettoyer as nettoyer_liste,
                        nb_éléments as liste_nb_éléments
feature
        Create  is
                do
                ensure
                        vide
                end ; -- Create
```

BTD/GL/POO 64

CENTRALE LYON

IMPLEMENTATIONS DE PILES (6)

```
nb_éléments : INTEGER  is
            -- Nombre d'éléments empilés ;
            do          Result  := liste_nb_éléments
            end ;  -- nb_éléments

pleine : BOOLEAN is
                        -- La pile est-elle vide ?
                        -- (La réponse est toujours non dans cette
                        -- implémentation)
            do          Result  := false
            end ;  -- pleine

sommet : T is           -- Dernier élément empilé
            require     not vide
            do          Result  := first
            end ;  -- sommet


empiler (x : T) is      -- Ajouter x sur la pile
            require     not pleine
            do          insérer_à_droite (x)
            ensure      not vide ; sommet = x ;
                        nb_éléments = old nb_éléments + 1
            end ;  -- empiler
```


CENTRALE LYON

IMPLEMENTATIONS DE PILES (7)

```
dépiler is  -- Retirer l'élément au sommet
            require     not vide
            do          détruire_à_droite(l)
            ensure      not pleine ;
                        nb_éléments :=  old nb_éléments - 1
            end ;  -- dépiler

changer_sommet (x : T) is
                        -- Remplacer l'élément au sommet par x
            require     not vide
            do          changer_i_ème(l,x)
            ensure      not vide ; sommet =  x ;
                        nb_éléments = old nb_éléments
            end ;  -- changer_sommet

nettoyer is             -- Retirer tous les éléments
            do          nettoyer_liste
            ensure      vide
            end ;  -- nettoyer

invariant
            position = 0

end  -- class PILE_CHAINEE
```

CENTRALE LYON

L'HERITAGE REPETE (1)

- **Que faire lorsqu'une classe hérite plus d'une fois de la même classe, directement ou indirectement ?**
  Politique de renommage :
  - ➔ **Primitive non renommée : partage**
  - ➔ **Primitive renommée : duplication**

On suppose une classe CONDUCTEUR avec les attributs

âge: INTEGER ;
adresse: STRING ;
no_de_permis: INTEGER ;
véhicules: LISTE [VOITURE] ;
nb_d_amendes : INTEGER

et des routines telles que : anniversaire is do âge := âge + 1 end;
payer_vignette is do .... end ;

Les héritiers peuvent inclure des classes CONDUCTEUR_FRANCAIS et CONDUCTEUR_AMERICAIN.

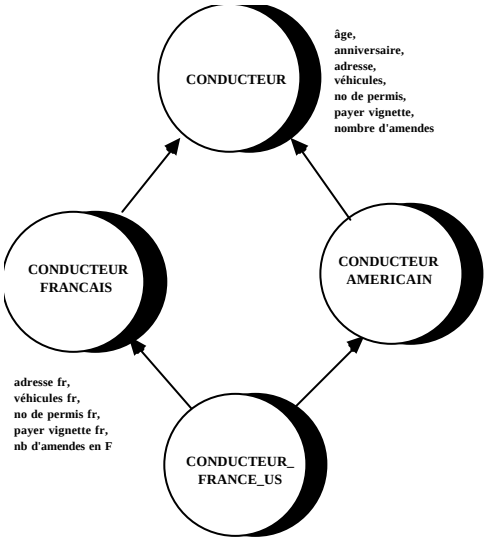
Celles-ci peuvent à leur tour avoir un héritier commun CONDUCTEUR_FRANCE_US.

Qu'advient-il des primitives héritées deux fois de l'ancêtre commun, telles qu'âge, adresse, payer_vignette etc. ?

BTD/GL/POO 67

CENTRALE LYON

L'HERITAGE REPETE (2)



BTD/GL/POO 68

CENTRALE LYON

L'HERITAGE REPETE (3)

Clause d'héritage de la classe CONDUCTEUR_FRANCE_US :

```
inherit
        CONDUCTEUR_FRANCAIS
                rename
                        adresse as adress_fr,
                        payer_vignette as payer_vignette_fr,
                        no_de_permis as no_de_permis_fr,
                        véhicules as véhicules_fr,
                        nb_d_amendes as nb_d_amendes_fr
                        .....
        CONDUCTEUR_AMAIRICAIN
                rename
                        adresse as US_address,
                         payer_vignette as pay_vehicle_registration,
                         no_de_permis as drivers_license_number
                        véhicules as US_vehicles,
                         nb_d_amendes as bad_driver_points,
                        .....
```

Les primitives âge **et** anniversaire**, qui n'ont été renommées sur aucun des chemins d'héritage, sont partagées.**

BTD/GL/POO 69

CENTRALE LYON

LES ASSERTIONS ET L'HERITAGE

- **Instruction "Check"**
- **Relation avec l'héritage : dans une classe descendante**

  ➔ **Inclure les invariants des ancêtres**
  ➔ **Garder ou affaiblir les pré-conditions des routines**
  ➔ **Garder ou renforcer les post-conditions des routines**

BTD/GL/POO 70

CENTRALE LYON

L'HERITAGE EST SOUS-TRAITANCE

Programmeur du client :

Obligations : N'appeler la routine que si x >= 0, epsilon >= 10^{-6}

Bénéfices : Obtenir en retour l'approximation souhaitée de la racine carrée

Implémenteur du module :

Obligations: Renvoyer l'approximation demandée

Bénéfices: Inutile de traiter les cas x > 0, ou epsilon <= 10^{-6}

Dans une redéfinition:

Pré-condition: epsilon >= 10^{-10}

Post-condition : approximation à epsilon / 2

BTD/GL/POO 71

CENTRALE LYON

Traitement des cas anormaux (1)

➔ **VERIFICATION A PRIORI**

➔ **VERIFICATION A POSTERIORI**

➔ **EXCEPTIONS**

BTD/GL/POO 72

CENTRALE LYON

Traitement des cas anormaux (2)

Un Contrat Logiciel

Client

Obligations: **Appeler** insérer **seulement sur une table non pleine**

Bénéfices: **Obtenir une table où** x **a été associé à** clé

Maître d'oeuvre

Obligations: **Insérer** x **pour qu'il puisse être retrouvé par** clé

Bénéfices: **Inutile de se préoccuper du cas d'une table pleine avant une insertion**

```
class TABLE [T] export insérer, valeur, détruire, ...
Feature     insérer (élément: T, clé: STRING) is  -- Insérer élément, associé à clé
                    require nb_éléments < max_éléments
                    do... "Algorithme d'insertion" ...
                    Ensure nb_éléments <= max_éléments ;
                              valeur (clé) = élément ;
                              nb_éléments = old nb_éléments +1
                    end ; -- insérer
            ... "Autres déclarations de primitives" ...
  Invariant 0 <= nb_éléments
            nb_éléments <= max_éléments
  end  -- class TABLE
```

BTD/GL/POO 73

CENTRALE LYON

Traitement des cas anormaux (3)

VERIFICATION A PRIORI

```
if not a then
        "Action pour traiter le cas anormal"
else
        r (arguments)
        "Suite du traitement"
end
```

Si possible, pour assurer une plus grande linéarité et donc lisibilité:

```
if not a then
        "Action pour traiter le cas anormal"
else
        r (arguments)
end ;
        "Suite du traitement"
```

BTD/GL/POO 74

CENTRALE LYON

Traitement des cas anormaux (4)

- **Quand la vérification a priori n'est pas possible :**

  1/ Impossible ou impraticable de vérifier la précondition à l'avance

  2/ La précondition n'est pas entièrement exprimable dans le langage d'assertions

  3/ Nombreux appels similaires; les situations anormales sont toutes traitées de la même façon

Equation matricielle:

```
if not a.régulière then
        "Traitement spécial"
else
        x:=a.solution (b)
end
```

BTD/GL/POO 75

CENTRALE LYON

Traitement des cas anormaux (5)

Techniques a posteriori

➔ Essayer l'opération, PUIS tester son succès

```
a.résoudre (b) ;
if not a.régulière then
        "Traitement spécial"
else    ... a.solution  est accessible ici ...
end
```

```
écrire ;
if erreur_es then ...
```

```
lire_caractère ;
if erreur_es then ...
Else    ... Ici le résultat de la dernière lecture est disponible
        ... par dercar , calculé par lire_caractère
end
```

BTD/GL/POO 76

CENTRALE LYON

Traitement des cas anormaux (6)

LES EXCEPTIONS (Quand le contrat est rompu)

Définitions :

Exception : événement anormal se produisant à l'exécution

Echec : impossibilité d'accomplir le but d'une routine

Un échec produit une exception chez le client.

Exceptions en Eiffel:

1/ Assertion non satisfaite (si elles sont contrôlées)

2/ Echec d'une routine appelée

3/ Accès à un objet inexistant (x.f**, avec** x.Void **vrai)**

4/ Signal du matériel (débordement, épuisement de la mémoire,...)

CENTRALE LYON

Traitement des cas anormaux (7)

Les trois cas d'utilisation des exceptions

Les exceptions ne sont pas une structure de contrôle pour les cas spéciaux mais attendus. Dans la plupart des situations, les structures de contrôle habituelles sont suffisantes.

Il reste trois cas:

1/Impossible de tester la précondition avant l'appel (ex.: débordement arithmétique, mémoire, BREAK,...)

2/Nécessité d'arrêter immédiatement

3/Tolérance aux fautes logicielles

CENTRALE LYON

Traitement des cas anormaux (8)

EXCEPTIONS: LES PRINCIPES

➔ **Première loi des contrats de logiciel: Une routine ne peut se terminer que de deux façons: soit elle remplit son contrat, soit elle échoue dans l'accomplissement de son contrat.**

➔ **Deuxième loi des contrats de logiciel: Si une routine échoue dans l'accomplissement de son contrat, il en est de même de l'exécution en cours de la routine qui l'a appelée.**

➔ **Seules deux réponses sont acceptables :**

1/ Panique contrôlée

2/ Résorption

(Exceptions non disciplinées: exemple ADA)

BTD/GL/POO

79

CENTRALE LYON

Traitement des cas anormaux (9)

EXCEPTIONS DISCIPLINEES

➔ **Rescue et retry**

➔ **Clause de secours ("rescue") d'une routine: exécutée lorsqu'une exception se produit durant l'exécution de la routine.**

➔ **Instruction "retry": réexécute la clause "do".**

➔ **Le but de la clause de secours est de remettre les choses en place et de placer l'objet dans un état stable, puis soit d'admettre l'échec (panique contrôlée), soit de relancer l'exécution, normalement en utilisant une autre stratégie (résorption).**

➔ **La clause de secours n'est pas chargée d'assurer le contrat de la routine, exprimé par la postcondition.**

BTD/GL/POO

80

CENTRALE LYON

Traitement des cas anormaux (10)

Exemples:

```
1/ essayer-transmission (message: STRING) is
        -- Transmettre message (5 essais au plus)
        -- positionner succès en conséquence
   local  échecs: INTEGER
   do
        if échecs < 5 then
                transmit (message) ;
                succès := true ;
        else
                succès := false
        end ;
   rescue
        échecs := échecs + 1 ;
        retry
   end -- essayer transmission
```


CENTRALE LYON

Traitement des cas anormaux (11)

```
2/ lire_entier: INTEGER is
        -- Lire un entier
        -- (jusqu'à 5 essais)
   local
        échecs: INTEGER
   do
        Result := getint
   rescue
        échecs := échecs + 1 ;
        if échecs <= 5 then
                message ("L'entrée doit être un entier.") ;
                message ("Veuillez essayer à nouveau") ;
                retry
        end;
   end -- lire_entier
```

CENTRALE LYON

Traitement des cas anormaux (12)

```
3/ quasi_inverse (x: REAL): REAL is
                    -- 1/x si représentable, 0 sinon
        local
                    déjà_essayé: BOOLEAN
        do
                    if not déjà_essayé then
                            Result := 1/x
                    else
                            Result := 0
                    end
        rescue
                    déjà_essayé := true ;
                    retry
        end  -- quasi_inverse
```


CENTRALE LYON

Traitement des cas anormaux (13)

```
4/ édition is           -- Boucle de base d'un éditeur de texte
        do
                    .... loop
                            exécuter_commande
                    end -- loop
        end  -- édition

   exécuter_commande is
                    -- Cycle de base de l'éditeur
        do
                    "Décoder commande utilisateur"
                    "exécuter l'opération correspondante"
        rescue
                    message ("Opération impossible") ;
                    message ("Veuillez en essayer une autre");
                    retry
        end  -- exécuter_commande
```

CENTRALE LYON

Traitement des cas anormaux (14)

```
5/ essai is
        local
                pair : BOOLEAN
        do
                if pair then algorithme_2
                else algorithme_1 end
        rescue
                pair := not pair ; retry
        end  --  essai
```


CENTRALE LYON

Aspects de la réutilisation

1. **Variations de types : <u>généricité</u>**
   - **Que sont les éléments de la table ?**
2. **Variations de représentation : polymorphisme**
   - **Variations dans des choix de structures de données et d'algorithmes: tables séquentielles (triées ou non), tableaux, arbres binaires de recherche, fichiers,...**
3. **Groupes de sous-programmes : <u>modularité</u>**
   - **Un sous-programme de recherche ne suffit pas: il doit être couplé avec des sous-programmes de création de table, d'insertion, de destruction, etc.**
4. **Indépendance vis-à-vis de la représentation : <u>surcharge</u>**
   - **Peut-on demander une opération telle que la recherche sans savoir quelle implémentation est utilisée de façon interne?** Recherche (y, t1)
5. **Eléments communs : héritage**
   - **Comment l'auteur du module peut-il tirer parti des nombreux points communs qui existent entre les éléments d'un sous-ensemble des implémentations possibles?** Par exemple: toutes les tables séquentielles.