BASES DE DONNEES A OBJETS

MARCIN SKUBISZEWSKI

De quoi parlons-nous ?

– D'une technique moderne, destinée à remplacer les bases de données relationnelles
– De bases de données
  $\Longrightarrow$ ça ressemble à SQL
  (jointures, sélections, index, optimisation de requêtes)
– De la programmation par objets
  $\Longrightarrow$ ça ressemble à la programmation moderne
  (pointeurs, héritage, methodes)

Mail :

`skubi@skubi.net`

Ce cours est sur

`http://www.skubi.net`

CE QUI SE FAIT : LES TROIS APPROCHES

Créer un SGBD à objets complet, en partant de zéro

– **O$_2$ (Ardent), Versant**

Prendre un SGBD relationnel, ajouter progressivement des propriétés objet (*systèmes relationnels-objet*)

– **Oracle**

Systèmes à objets persistents : prendre un système de programmation par objets, et ajouter

– la possibilité de mémoriser les objets
– d'autres propriétés de SGBD (contrôle de concurrence, journalisation)
– **ObjectStore (ObjectDesign)**

Toutes ces approches aboutissent finalement à la même chose.

IDEES

Les objets : ils ressemblent aux tuples ou aux enregistrements, mais avec

- pointeurs
- héritage
- methodes

Les objets dans une base de données

- accès direct à une base de données
  à partir d'un langage de programmation
- collections
- racines de persistence
- persistence par magie (par attachement)

Requêtes

- jointures **et** pointeurs **et** méthodes **et** index **et** sélections
- OQL ressemble à SQL

PLAN DU COURS (NON DEFINITIF)

Idées essentielles, pointeurs

- Notion d'objet
- Pointeurs
- Collections
- Racines de persistence
- Persistence par magie ; *ici s'arrete le premier cours*
- Schéma objet

Idées propres au monde objet : encapsulation

- Methodes
- Héritage

Les langages essentiels

- Langage OQL
- Bibliothèque C++ ODMG
- Java persistant

Quelques éléments sur le fonctionnement interne

- systemes client-serveur
  - serveurs de pages
  - serveurs de verrous
- systemes relationnel-objet

NOTION D'OBJET—IDEE DE BASE

C'est la même chose qu'un *tuple* ou qu'un *enregistrement* (`struct`, `class`)

```
class Personne {
    char prenom[30];
    char nom[30];
    char NumSecu[16];
};
```

C'est un tuple

```
select p.NumSecu from p in TableDePersonnes
    where p.prenom = "Gertrude",
    p.nom = "Dubois"
```

C'est un enregistrement, comme dans un langage impératif traditionnel
(genre Pascal, Java, C, C++, O_2C)

(on serait tenté de dire : comme dans un langage non destiné aux bases de données)

```
int f ()
{
    Personne p;
    strcpy (p.prenom, "Gertrude");
    .......
```

PREMIERE IDEE : POINTEURS

```
class Personne {
    char prenom[30];
    char nom[30];
    char NumSecu[16];
    Personne *pere, *mere;    (en C++)
ou bien
    d_ref <Personne> pere, mere;    (en ODMG)
};
```

Cherchons les enfants de Gertrude Dubois

```
select p->nom from p in TableDePersonnes
    where p->mere->prenom = "Gertrude",
          p->mere->nom = "Dubois"
```

Inscrivons Gertrude Dubois dans la base de données

```
int f ()
{
    d_ref<Personne> p = new Personne;
    strcpy (p->prenom, "Gertrude");
    .......
    p->pere = q;
    p->mere = r;
    TableDePersonnes += p; // optionnel
```

LES POINTEURS, C’EST NATUREL

Sans les pointeurs (en relationnel), on aurait

```
class Personne {
    char prenom[30];
    char nom[30];
    char NumSecu[16];
    char pere[16], mere[16];  // numéros
                  // de sécurité sociale
};
```

Numéro de sécurité sociale $\leftrightarrow$ identifiant de la personne

– on réinvente les pointeurs (solution *ad hoc*)
– méthode pas toujours correcte

C’est comme des pointeurs *ad hoc*, en un peu plus compliqué

LES POINTEURS ET LES JOINTURES

Cherchons les couples (mère, enfant) :

```
select (enfant: p, mere: p->mere)
    from p in TableDePersonnes
    where p->mere != nil
```

La même chose en relationnel (sans les pointeurs)

```
select (enfant: p, mere: q)
    from p, q in TableDePersonnes
    where p.mere == q.NumSecu
```

Comment c'est réalisé : avec un index sur `NumSecu` ;
pour chaque enfant `p`, on utilise l'index pour trouver sa mère `q`, c'est-à-dire
un `q` tel que `p.mere == q.NumSecu`

Souvent, le fait de suivre un pointeur remplace la jointure

– Suivre pointeur `p->mere`
au lieu de chercher un `q` tel que `p.mere == q.NumSecu`
– Plus simple, plus naturel
– Ajouter les pointeurs, c'est combler une lacune du modèle relationnel,
grâce à une idée qui existe depuis longtemps dans le langages traditionnels

LES POINTEURS ET LES JOINTURES, suite

La jointure est quand même utile : cherchons des homonymes

```
select (premier: p, second: q)
    from p, q in TableDePersonnes
    where p->prenom == q->prenom
```

OBJET : UNE DEFINITION PRECISE

Une structure de données qui peut être pointée

– (en C++ c'est différent, toute variable peut être pointée)

Un objet n'est pas nécessairement un enregistrement

– une collection est un objet ou non, selon ce qui est déclaré

Un enregistrement n'est pas nécessairement un objet

Un objet n'est jamais contenu dans un autre :
on ne pointe jamais un morceau d'objet

```
class A {
    class B {
        ....
    } b;
};

class A {
    d_ref <B> b;
};
```

DEUXIEME IDEE : COLLECTIONS

```
class Personne {
    char prenom[30];
    char nom[30];
    char NumSecu[16];
    d_ref <Personne> pere, mere;
    d_set <d_ref <Personne>> enfants;
};
```

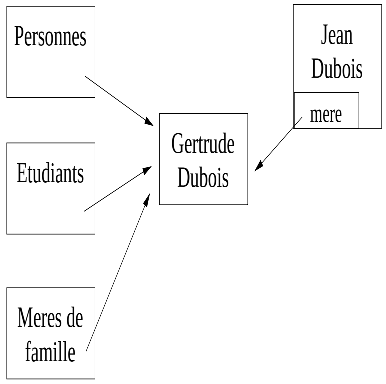
A quoi ça correspond ?

– aux tableaux et aux structures *ad hoc* des langages traditionnels
– aux tables des BD relationnelles (mais c'est beaucoup plus général)

COLLECTIONS, suite

Qu'est-ce qu'on met dans une collection ?

– Des pointeurs, pas des objets



– On peut aussi mettre directement des valeurs (qui ne sont pas des objets)

  – entiers, chaînes de caractères...

Les trois sortes de collections (ODMG, O_2, consensus large)

set ensemble (non ordonné, chaque élément est présent une fois au plus)

bag ensemble, mais un élément peut être présent plusieurs fois

list liste ordonnée

COLLECTIONS, détails

Set

```
set (a,b,c)
A+B
A*B
A-B
A += set(a) // cas spécialement optimisé
```

Bag

Comme set, mais on prend en compte les éléments qui se répètent

```
bag(a,a,b,b,c,c) - bag(a,a,b) = bag(b,c,c)
```

List

```
list (a,b,c)
A+B
```

COLLECTIONS PETITES ET GRANDES, INDEX

Il y a de petites collections à l'intérieur d'objets

... on vient de le voir

De grosses collections font office de tables

La collection `TableDePersonnes` contient des pointeurs vers de nombreux objets, ça équivaut à une table de tuples représentant des personnes

- On peut imaginer des situations intermédiaires
- L'idée de collection est simple et générale, de 1 à 10 milliards d'élements (c'est donc une bonne idée)
- Les détails de réalisation varient
  - O_2 comporte 3 réalisations de collections, suivant le nombre d'éléments

Les index

Comme dans une base relationnelle

```
select p->NumSecu from p in TableDePersonnes
    where p->prenom = "Gertrude",
    p->nom = "Dubois"
```

TROISIEME IDEE : RACINES DE PERSISTENCE

Il s'agit d'entités ayant un nom

– donc, directement accessibles

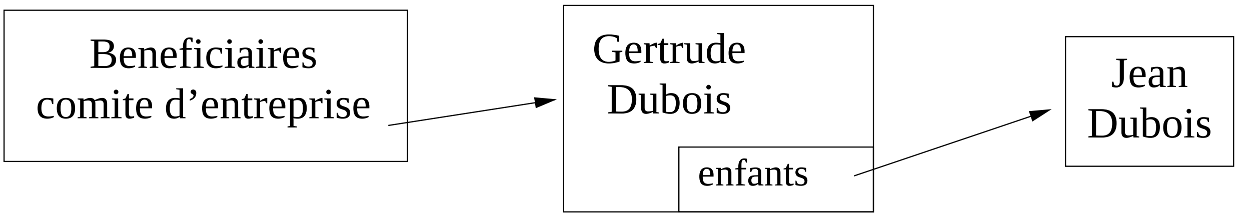
Un objet peut être accédée

– directement par son nom (`TableDePersonnes`, `BeneficiairesCE`)
– indirectement, en passant par d'autres entités

Un objet peut être complètement inaccessible

RACINES DE PERSISTENCE, exemple

On suppose que `BeneficiairesCE` est la seule racine de persistance




```
select j from j in (
    element (select p->enfants
        from p in BeneficiairesCE
        where p->nom == "dubois",
              p->prenom == "gertrude")
) where j->prenom == jean
```

L'ensemble d'enfants de Gertrude Dubois est un morceau d'objet, accessible indirectement

Jean, fils de Gertrude, est représenté par un objet accessible indirectement

RACINES DE PERSISTENCE, suite

Une racine de persistance contenant une collection correspond précisement à une table relationnelle

Mais les collections sont plus générales que les tables.

D'autres cas

- Une racine de persistance contenant un objet
  - Base de données *la famille de Gertrude* : `Gertrude`
- Une racine de persistance contenant un entier
  - `NombreAdherents`
- Une collection qui ne correspond à aucune racine de persistance

QUATRIEME IDEE : PERSISTENCE PAR MAGIE (PAR ATTACHEMENT)

L'utilisateur

– n'insère pas les objets dans la base de données
– n'enlève pas les objets de la base de données

```
int f ()
{
    d_ref<Personne> p = new Personne;
    strcpy (p->prenom, "Gertrude");
    ......
    BeneficiairesCE += set(p);

    d_ref<Personne> q = new Personne;
    strcpy (p->prenom, "Jean");
    ......
    p->enfants += set(q);
    q->mere = p;

    d_set<d_ref<Personne>> aContacter;
    ......
}

void virerDuCE (d_ref<Personne> p)
{
    BeneficiairesCE -= set(p);
}
```

QUELLE EST LA REGLE ?

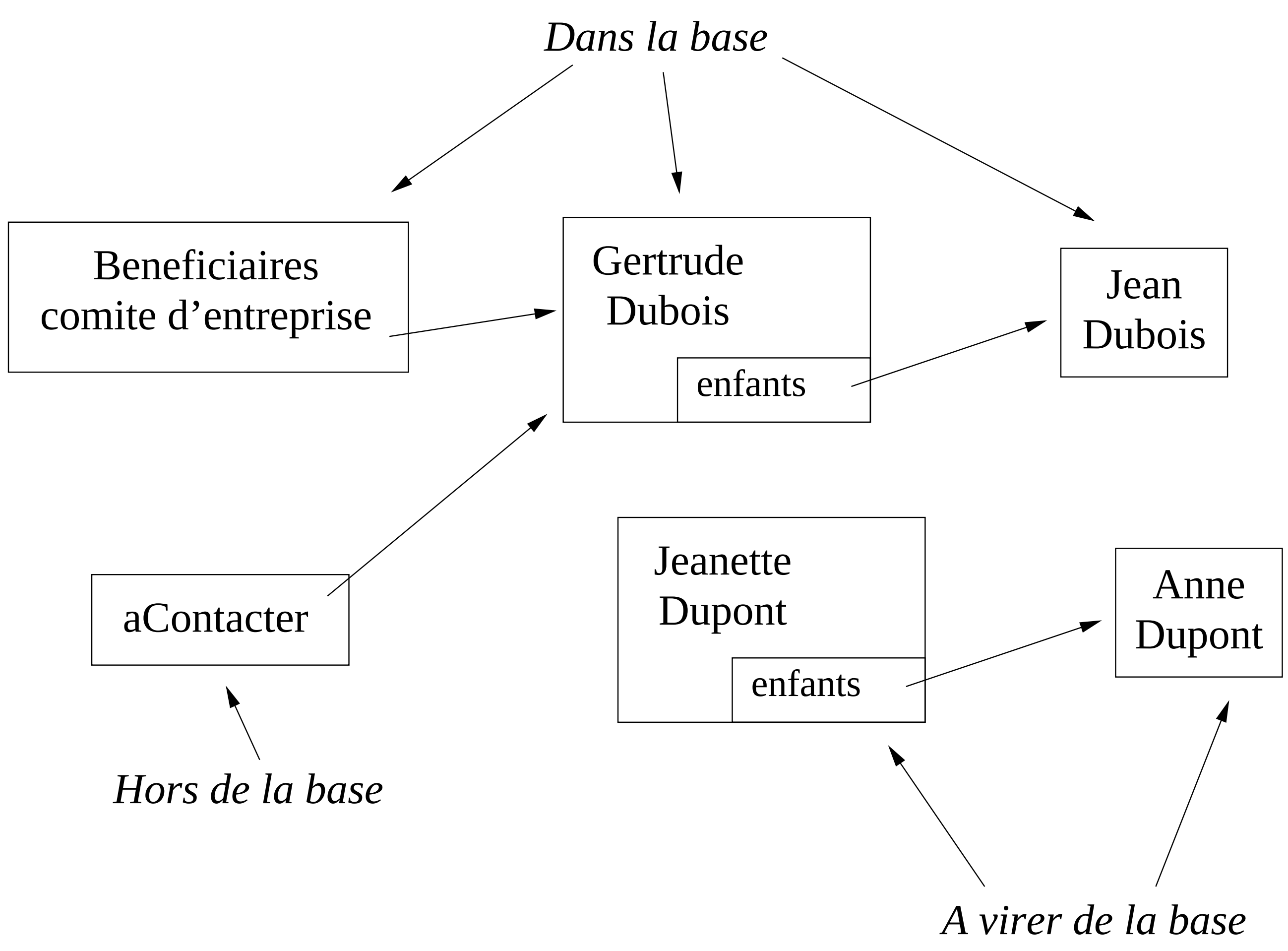
Un objet qui peut être accédé (**accessible**, **atteignable**)

– directement par son nom (`TableDePersonnes`, `BeneficiairesCE`)
– indirectement, en passant par d'autres entités

... doit être dans la base de données

Un objet complètement inaccessible (pour toujours) (**miette**)

... doit de préférence être supprimé de la base de données



POURQUOI C'EST IMPORTANT

Parce que **l'utilisateur ne sait pas quand enlever un objet de la base**

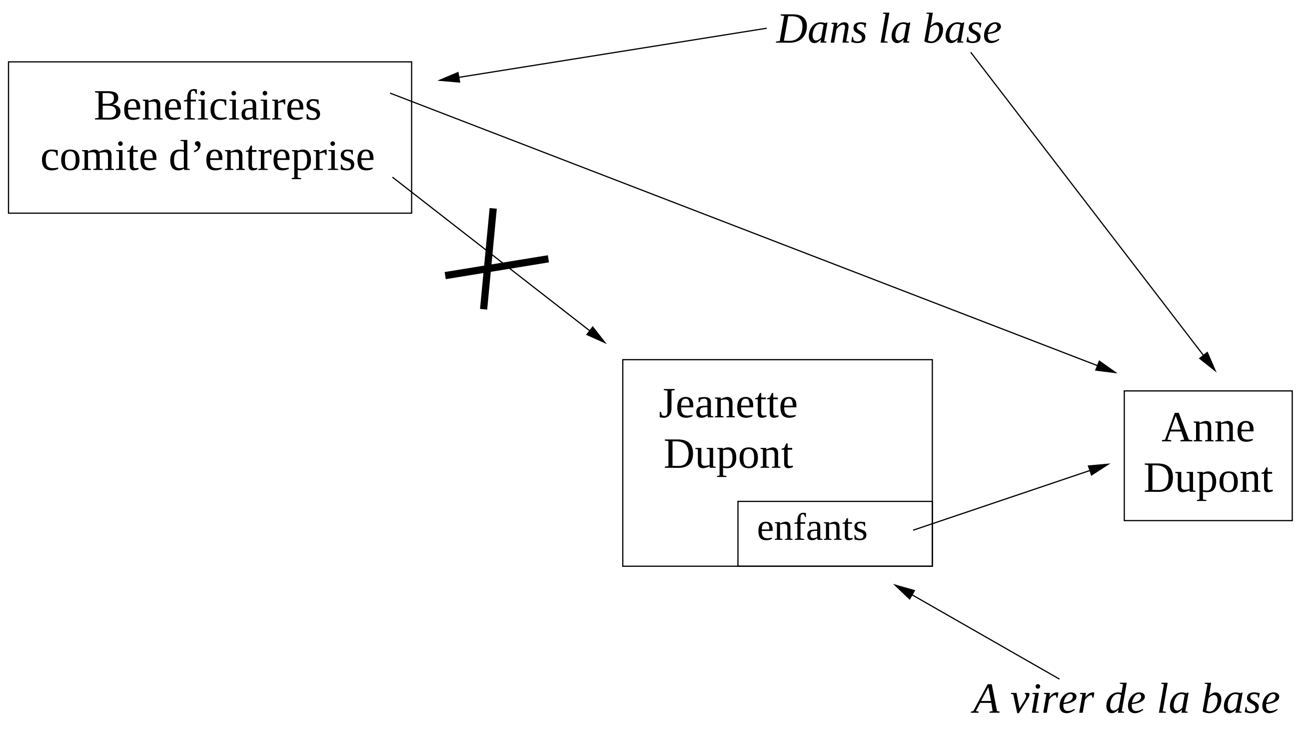
Une personne peut être

- client
- fournisseur
- créancier
- débiteur
- mauvais payeur
- adversaire dans un procès
- salarié
- parent de salarié (avec le bénéfice du CE)

Opération faisable : enlever une personne d'une liste

- on maitrise ce qui se passe dans un module

Opération infaisable, non modulaire : décider qu'un objet est réellement inutile



L’unification entre bases de données et programmation par objets est maximale : l’utilisateur traite de la même façon des objets dans la base de données et des objets propres à son programme.

COMMENT CA MARCHE

Pendant chaque transaction :

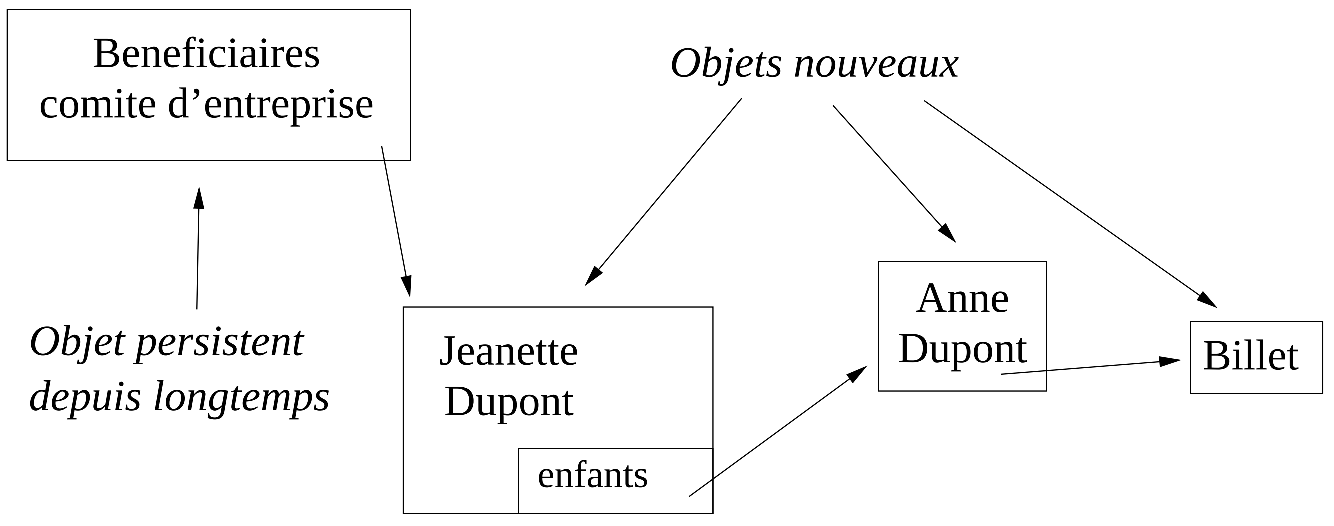
La promotion persistante

– Il s'agit de supprimer toutes les cas où un objet **persistant** (présent dans la base de données) pointe sur un objet **transitoire** (privé, appartenant à la transaction mais hors de la base)

– On s'intéresse seulement aux nouveaux cas, crées par cette transaction

– A chaque fois que la transaction modifie un pointeur `r` dans un objet persistant `A`, qui désormais se met à pointer sur un objet transitoire `B`, on appelle `p` le pointeur incriminé et on exécute `promotion (p)`

```
promotion (Object *p) {
     copier *p dans la base de données (le rendre persistant)
     pour tout pointeur q dans l'objet *p, faire
          si *q est transitoire
               promotion (q)
}
```



De temps en temps :

Le ramassage de miettes

- Il s'agit de supprimer de la base de données les objets miettes (inaccessibles)
- Pour chaque racine de persistance `p`, on fait `propagation (p)`

```
propagation (Object *p) {
     marquer *p
     pour tout pointeur q dans l'objet *p, si *q n'est pas
encore marqué, faire
          propagation (q)
}
```

- On supprime tout ce qui n'a pas été marqué

LE SCHEMA OBJET

Contient

- Les types des objets pouvant être utilisés
  - champs
  - héritage
  - méthodes
- Les racines de persistance et leurs types
  - comme en relationnel
  - connaissant le schéma, vous pouvez écrire

```
select (enfant: p, mere: p->mere)
    from p in TableDePersonnes
    where p->mere != nil
```

Ces éléments déterminent la sémantique et la validité des programmes.

Ne contient pas

- Les index
- Les tailles des collections

Ces éléments influent sur la manière d'exécuter les programmes (optimisation), mais pas sur la légalité et la sémantique des programmes.

PLAN DU DEUXIEME COURS

Quelques principes de génie logiciel

- Notion de module
- Comment on compose des modules

Les méthodes

L'héritage

Les éléments (membres) privés et publics d'un objet

Détails d'O_2C

La syntaxe O_2C, C++, OQL, avertissement

- ce que je dis est incomplet, notamment il y a souvent plusieurs syntaxes pour exprimer la même chose, et je ne présente qu'une syntaxe

NOTION DE MODULE

Personne ne peut comprendre un programme de 100000 lignes.

On peut comprendre un programme comportant 4 modules **simples, dont le rôle est clair**

Personne ne peut comprendre un module de 25000 lignes.

On peut comprendre un module comportant 4 sous-modules **simples, dont le rôle est clair**

– et ainsi de suite...

Donc, on subdivise chaque logiciel en modules

– Subdivision récursive
– A chaque niveau, peu de modules
– Le rôle de chaque module est simple et clair, le module est simple à utiliser

PROPRIETES DES MODULES

Un module représente un objet du monde réel facile à comprendre

- ouvrier
- étudiant
- compte en banque
- figure géometrique
- pièce automobile

CE QUI COMPOSE UN MODULE

Des données, correspondant à l'objet représenté

- nom, prénom, numéro de sécurité sociale

Une interface de programmation (API: *application programmer's interface*)

- payer un ouvrier
- obtenir le solde d'un compte
- modifier une figure géometrique
- calculer la surface d'une figure géometrique
- commander une pièce automobile

Des **invariants** (aussi appelés **contraintes d'integrité**): propriétés toujours vérifiées, sur lesquelles on peut compter

- nombre de pièces en stock ≥ 0
- toutes les personnes se trouvent dans la collection `ListeDesPersonnes`
- liens inverses: `a` pointe sur `b` ssi `b` pointe sur `a`
  - Exemple: pointeurs vers les parents, pointeurs vers les enfants

MODULES ET OBJETS

Un module correspond à un type d'objet

- Un objet dans la base pour chaque objet représenté
- Champs de l'objet dans la BD $\longleftrightarrow$ données concernant l'objet représenté
- ***Méthodes*** de l'objet $\longleftrightarrow$ interface de programmation

Les invariants ne sont pas maintenus par le système

- A vous de les maintenir
- Exception dans tous les systèmes sérieux: il n'y a jamais de *poineurs errants*
  - c'est obtenu par la persistance par attachement
- Diverses exceptions dans divers systèmes (pas dans O_2C)
  - *Class extents*: collections contenant tous les objets d'une classe, maintenues automatiquement
  - Liens inverses

METHODES

Une méthode: une fonction attachée à un type

- qui prend comme premier argument implicite un objet de ce type
- étroitement attachée au type: les méthodes, c'est *ce qu'on peut faire avec un objet du type donné*

Les méthodes font corps avec les objets correspondants comme s'il s'agissait de champs

- Similitudes essentielles: syntaxe, utilisation en OQL

Syntaxe O_2C, utilisation de la méthode

```
o2 Ouvrier p;
o2 CompteBancaire c;
p->payer(1000);
c = p->refBancaires; /* ici on ne sait pas si c'est un champ
ou une méthode */
```

METHODES, SYNTAXE O_2C (suite)

Syntaxe O_2C, déclaration de la méthode

```
Class Ouvrier
    public type tuple(...)
    method
        public refBancaires: CompteBancaire,
        public payer (somme: int)
end;
```

Syntaxe O_2C, le corps de la méthode

– très similaire à celle d'un `run body`
– l'objet en question est désigné par `self`

```
method body payer (somme: int) in class Ouvrier {
    self->refBancaires->virer (somme);
}
```

METHODES, SYNTAXE C++

Syntaxe C++

- mêmes principes qu'en O_2C, divers détail différents
- pas besoin de distinguer entre C++ ordinaire et C++ ODMG
- on écrit `this` au lieu de `self`
- mais aussi, on peut accéder directement aux champs et aux methodes de l'objet en question: au lieu de `this->methode(x)` ou `this->champ`, on peut écrire `methode(x)` ou `champ`.
- on écrit `()` pour une méthode sans arguments

```
class Ouvrier {
 public:
    void payer (int somme);
};

void Ouvrier::payer (int somme)
{
    refBancaires()->virer (somme);
}
```

METHODES, UTILISATION EN OQL

Méthodes sans arguments

– Exactement comme s'il s'agissait d'un champ
  Si tous les virements vers la BNP doivent être refaits

```
select p from p in Ouvriers
    where p->refBancaires->nomBanque = "BNP"
```

... on vient d'introduire une notion nouvelle: mélange sans peine de OQL et d'un langage impératif

– cette notion est importante

METHODES, UTILISATION EN OQL, suite

Méthodes avec arguments

```
method body descendDe(ancetre: Personne): Boolean
in class Personne {
    o2 Personne p = self;
    do {
        if (p == ancetre)
            return true;
        if (p == nil)
            return false;
        p = p->mere;
    }
}

select p from p in Personnes

where p->prenom = "Gertrude", p->descendDe(Gertrude);
```

METHODES, TERMINOLOGIE

Membre d'un type objet désigne indistinctiment

- un champ
- une méthode

... cette terminologie est logique puisque les méthodes et les champs s'utilisent d'une façon similaire

CONSTRUCTEURS

Un constructeur ressemble à une méthode (c'est une sorte particulière de membre)

- il sert à créer un objet d'un type donné
- `self` ou `this` a un sens dans le constructeur, et désigne l'objet nouvellement créé
- en C++ et en Java, nous avons l'accès direct (non qualifié) aux membres: il s'agira de membres de l'objet nouvellement créé

Pourquoi utiliser les constructeurs

- pour la modularité (c'est comme dans le cas des méthodes)
- les constructeurs permettent d'assurer le respect des invariants dès le début de la vie d'un objet
  - toute personne a un nom qui n'est pas la chaîne vide
  - toute personne est inscrite dans `ListeDesPersonnes`
- le découpage en types correspond bien au découpage en modules

CONSTRUCTEURS, SYNTAXE C++

Utilisation

```
d_ref<Personne> p = new Personne ("Gertrude",
    "Dubois", 15, 0);
```

Déclaration

– on peut mettre plusieurs constructeurs par type, distingués par le type des arguments

```
class Personne {
 public:
    Personne (char *prenom, char *nom,
        int numSecu, d_ref<Personne> mere);
};
```

CONSTRUCTEURS, SYNTAXE C++, suite

Le corps du constructeur

```
Personne::Personne (char *prenomA, char *nomA,
                    int numSecuA,
                    d_ref<Personne> mereA)
    : prenom(prenomA), nom(nomA), numSecu(numSecuA)
{
    TableDePersonnes += set(this);
}
```

CONSTRUCTEURS, SYNTAXE O_2C

Utilisation

```
o2 Personne p = new Personne ("Gertrude",
    "Dubois", 15, nil);
```

Déclaration

– le nom est toujours `init`
– sinon, les constructeurs se déclarent comme des méthodes ordinaires

Le corps du constructeur

```
method body init (prenomA: string, nomA: string,
                  int numSecuA, Personne mereA)
    in class Personne
{
    prenom = prenomA;
    ...
    ListeDesPersonnes += unique set(self);
}
```

CONSTRUCTEURS, PARTICULARITES

Un constructeur comporte

- une partie automatiquement générée
  - réservation mémoire
  - appel des constructeurs de sous-objets
- une partie que vous avez écrite

Un constructeur n'est pas une fonction comme une autre

On ne peut appeler un constructeur **que** pour créer un objet

- pas pour réinitialiser un objet déjà créé
- pas pour compléter l'action d'un autre constructeur

MEMBRES PRIVES ET PUBLICS

Vu de l'extérieur, un objet

- doit être simple
- ne doit pas montrer de détails de mise en oeuvre qui pourraient changer

A l'intérieur, un objet peut être compliqué

- Ce qui fait partie de l'interface (API) est ***public***
- Les reste est ***privé***: cuisine interne utilisable seulement par les méthodes de l'objet

```
class Ouvrier {
 public:
    void payer (int somme);
 private:
    enum {
        virement, chequeDomicile
    } commentPayer;
    union {
        d_ref<CompteBancaire> refBancaires;
        char *adresse;
    };
};
```

HERITAGE : SES RAISONS D'ETRE, premier exemple

Un étudiant est une personne

L'objet représentant un étudiant a toutes les propriétés d'un objet représentant une personne, plus d'autres propriétés.

A chaque fois qu'une personne doit être utilisée dans un programme, un étudiant doit pouvoir être utilisé à sa place

```
o2 Etudiant e = new Etudiant (...);
o2 Personne maman;
ListeDesPersonnes += e;
printf ("%s", e->prenom);
maman = e->mere;


o2 OuvrierSpecialise os;
...
os->payer (2000);
```

HERITAGE : DEFINITION

Un type `B` hérite d'un type `A`

- lorsque les `B` forment un sous-unsemble, une spécialisation de `A`
- lorsque chaque `B` a toutes les propriétés d'un `A`
- alors `B` possède tous les champs de `A`, plus d'autres
- alors `B` possède autant de méthodes que `A`, ayant les mêmes signatures, plus d'autres
  - `B` peut avoir les mêmes méthodes que `A` ou des méthodes différentes, juste avec les mêmes signatures

Syntaxe C++

```
class Etudiant: public Personne {
 private:
    d_set<d_ref<Cours>> listeCours;
 public:
    void inscrire (d_ref<Cours> c);
    int estInscrit (d_ref<Cours> c);
};
```

HERITAGE : SES RAISONS D'ETRE, second exemple

Un ensemble d'objets qui

– partagent certaines propriétés, et ont donc une API commune (en partie ou en totalité)
– divergent sur certains points, donc les détails internes doivent diverger

```
class FigurePlane {
 public:
    double surface ()=0;
    void dessiner ()=0;
    int xmin()=0; int xmax()=0;
    int ymin()=0; int ymax()=0;
    double diametre()=0;
};

class Disque: public FigurePlane {
 private:
    int centrex, centrey, rayon;
 public:
    double surface ();
    void dessiner ();
    int xmin()=0; int xmax()=0;
    int ymin()=0; int ymax()=0;
    double diametre()=0;
};
```

```
float disque::surface()
{
    return 3.1416*rayon*rayon;
}
```

Il ne peut pas exister d'objets de type `FigurePlane`, seulement de type `Disque`, `Rectangle` etc.

- `FigurePlane` est une ***classe abstraite***
- `FigurePlane` groupe des objets divers, ayant des propriétés en commun

HERITAGE : SES RAISONS D'ETRE, second exemple, suite

On peut faire

```
name MesFigures: list(FigurePlane);

run body {
    o2 FigurePlane f;
    for (f in MesFigures) {
        f->dessiner();
    }
};
```

PLAN DU TROISIEME COURS

Des sujets divers

- Eléments de syntaxe O_2C
  - Listes
  - D'autres collections
  - Mélange OQL/O_2C
- Les index dans O_2
- Détails de mise en oeuvre
  - Fonctionnement en client-serveur
  - Relation entre les objets dans le SGBD et les objets vus par C++ ou Java
  - Où on mémorise les méthodes
- Des systèmes à objets persistants (des SGBD incomplets)
- Les limites de l'approche objet

LA SYNTAXE DES LISTES EN O_2C

Liste vide

```
list()
```

Liste explicite

```
list(1,2,5,3)
```

Liste explicite avec des répétitions

```
list(7,4:0,2:15,2:9) == list(7,0,0,0,0,15,15,9,9)
```

Concaténation

```
l1+l2   list(1,2,3)+list(4,5,6) == list(1,2,3,4,5,6)
```

Accès à un élément, en lecture ou écriture,
comme dans un tableau en Pascal ou C ou Java

`l[5]` (l'élément de tête porte le numéro 0)

`l[5] = 7`

LA SYNTAXE DES LISTES EN O_2C, suite

Accès à un intervalle, en lecture ou écriture

`l[5:7]` une liste de 3 éléments
`l[:7]` la liste composée des 8 premiers éléments de `l`
`l[7:]` la liste composée de tous les éléments de `l`, sauf les 7 premiers
`l[5:7] = list (1,2,3,4)` remplace un intervalle de l par un autre (au total, `l` a maintenant un élément de plus)

Notez la différence

`l[5:5]` une liste contenant un seul élément
`l[5]` ceci n'est pas une liste

Appartenance : retourne l'index de la première apparition de l'élément dans la liste, sinon un nombre négatif

```
(5 in list(0,1,5)) == 2
(3 in list(0,1,5)) < 0
```

LA SYNTAXE DES COLLECTIONS EN O_2C, compléments divers

Nombre d'éléments

```
count(coll)
```

Extraction de l'élément unique d'un `set` ou d'un `unique set` : comme en SQL

`element(s)` (si l'ensemble n'est pas un singleton, il y a erreur)

Itération sur n'importe quelle collection

```
for (p in coll) {...}
for (p in coll where <condition>) {...}
```

MELANGE ENTRE O_2C ET OQL

Appeler du O_2C à partir de OQL

– les methodes

Appeler du OQL à partir de O_2C

```
o2query (resultat, "select p from p in ListeDesPersonnes \

where p->nom = $1, p->prenom=$2", nomChoisi, prenomChoisi);
```

On peut mélanger très finement O_2C et OQL
Les index fonctionnent seulement en OQL

LES INDEX : UTILITE

Les indexes permettent de rendre très rapides certaines selections qui, autrement, auraient été très lentes.

```
class Personne {
    char prenom[30];
    char nom[30];
    char NumSecu[16];
    d_ref <Personne> mere, pere;
};

select p from p in ListeDesPersonnes
    where p->nom = "Dubois"

select p from p in ListeDesPersonnes
    where Gertrude=p->mere

select p from p in ListeDesPersonnes
    where p->prenom = "Pierre", p->mere=Gertrude
```

LES INDEX : CHEMINS

Exemples de chemins

```
p->nom
p->mere
p->mere->nom
p->voiture /* ici, voiture est un petit tuple inclus dans un
plus grand, ce n'est pas un objet */
p->voiture.plaque
```

L'index posé **dans une collection, sur un chemin**, permet de retrouver dans cette collection les objets selon la valeur de l'attribut accessible par ce chemin.

LES INDEX : CHEMINS, suite

Les attributs sur lesquels nous savons poser un index

- les scalaires (entiers chaines de caractères, réels, booléens)
- les pointeurs

```
select p from p in ListeDesPersonnes
    where Gertrude=p->mere
```

Les choses sur lesquelles nous ne savons pas poser un index

- Valeurs composées (limitation de O_2)

  `p->voiture` /* ici, `voiture` est un petit tuple inclus dans un plus grand, ce n'est pas un objet */

- Valeurs retournées par des methodes (problème fondamental)

  `p->refBancaires` /* ici, `refBancaires` est un appel de methode */

LES INDEX : CHEMINS, suite

Restriction sur les chemins : dans O_2, le chemin ne peut pas traverser de pointeurs (bien qu'il peut se terminer par un pointeur)

– Chemin se terminant par un pointeur

  `p->mere`

  Pas de problème

  – si le champ `mere` change dans `p`, les index concernant `p` seront modifiés.
  – si le contenu de l'objet `p->mere` change, il n'y a rien à faire.

– Chemin traversant un pointeur

  `p->mere->nom`

  Problème

  – si le contenu de `p->mere` change, il faudra modifier les index concernant `p` , et nous ne savons pas le faire.

SYNTAXE O_2

```
create index ListeDesPersonnes on nom;
create index ListeDesPersonnes on voiture.plaque;
delete index ListeDesPersonnes on nom;
delete index ListeDesPersonnes on voiture.plaque;
```

LE FONCTIONNEMENT CLIENT-SERVEUR

Principe essentiel: *data shipping* (serveur de données)

- service de pages
- service d'objets

A opposer au *function shipping* traditionnel (serveur d'exécution)

- surtout en BD relationnelles
- certains SGBD à objets font les deux (*data shipping* et *function shipping*)

LES MECANISMES DE MEMORISATION PERSISTANTE D'OBJETS

SGBD à objets restreints

- but: faire de la programmation comme d'habitude, mais avec des objets persistants
- accès à l'aide d'un langage habituel (C++, Java)
- certaines fonctionnalités de base de données
  - accès à l'aide d'un langage impératif toujours présent
  - requêtes et index limités ou absents
  - concurrence limitée, voire absente

Exemples

- Object Design International: ObjectStore et divers produits
- Université de Glasgow et SunSoft: PJava

REMARQUES FINALES SUR L'APPROCHE OBJET

Limitations

- Jeunesse
- Grande liberté, rendant difficiles les optimisation

Point fort

- Les systèmes à objets se programment d'une façon logique, conforme au bon sens, conforme aux principes du génie logiciel.