

Abstractions et généricité dans *O/eNa* bibliothèque de traitement d'images en C++

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

nov. 2004

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de “classification”
 - Classification et discriminants “orthogonaux”
 - L'héritage par propriétés

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Le LRDE

le labo est jeune :

- créé il y 7 ans
- âge moyen des 8 permanents : 32 ans
- le concept : on forme par la recherche des étudiants
- mais a déjà des couleurs
dominante = calcul numérique scientifique...

La recherche

différentes façon de voir :

- plusieurs domaines de compétences
- deux axes : programmation performante, calcul distribué
- des projets :
 - *OleNA*, bibliothèque de traitement d'images
 - VAUCANSON, un transfert pour les automates
 - TRANSFORMERS, à terme l'outil de traduction
 - FAST, *Fast Algorithms for Scientific Computing through Transformations*, à venir...

Outline

- 1 Avant-propos
 - Présentation
 - **Une histoire**
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

De 1993 à 1995

- ante 1993, deux bibliothèques disponibles :

- une avec chargement ligne par ligne
- une avec chargement total en mémoire

mais :

- limitées aux images 2D et il me fallait traiter des images 3D
- les images de travail ne peuvent pas toutes tenir en mémoire

- post 1993 → TIVOLI :

- pour les images 2D et 3D avec *memory mapping*
- pour différents types de données (bool, entiers / 8 et 16 bit)

- 1995 (pas de chance) :

- je recevais de nouvelles données sur 12 bit...
- je voulais utiliser des images de vecteurs...

En 1997 et depuis

des constatations sur l'équipement des routines :

- toutes ont un mode “debug”
- certaines (itératives) ont un code de visualisation
- quelques unes (par duplication) peuvent être appelées sur un sous-ensemble de points

Depuis 1997

- images de flottants (CEA BIII)
- maillages avec des données quelconques (CEA SHFJ)
- d'autres espaces *vus comme* des images
 - histogramme
 - Fourier
 - etc.
- images en couleur (SWT) et différents codages de la couleur
- images *vraiment* astronomiques
- images à pavage hexagonal (j'en rêve)
- ...

1998, début d'OleNa

- première façon de voir :
 - objectif principal : ne pas être limité par l'outil
 - objectif secondaire : s'affranchir des “détails” et “**variations**” d'implémentation
- autre façon de voir :
 - être général tout en restant performant
 - être ouvert pour affecter les algorithmes de façon **non intrusive**
- grosse difficulté : *mais comment faire ?*

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - **Généralités**
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Environnement et bibliothèque

- partie d'un environnement $\neq$ des autres modules :
 - jeu d'exécutables
 - interpréteur de commandes en ligne
 - visualiseur interactif
 - atelier de programmation visuelle de chaînes de traitement
- cœur de l'environnement :
 - la "puissance" de cet env. dépend d'elle
 - elle ne doit pas être "limitative"

Capacités et fonctionnalités

Définitions :

- puissance = son potentiel, ses **capacités**
par opposition avec :
- ses **fonctionnalités** = ce qui est déjà disponible

première idée clef

avoir une tripotée de fonctionnalités mais être limité à ces fonctionnalités est un échec

Exemple et reformulation

- un exemple :
 - on lit souvent “nous fournissons une érosion” (vrai ?)
 - on s’aperçoit que l’ “érosion” en question ne fonctionne que pour les images 2D, qu’avec certains éléments structurants (cruelle vérité)
- reformulation de l’idée clef :
 - un algorithme est par essence **générique**
 - il impose des propriétés (contraintes) aux données manipulées
 - tout ce qui n’est pas contraint doit être possible

Critères de qualité

- modularité
→ lire (Meyer, 1988)
- adaptabilité
→ possibilité de changer de contexte (de données)
- extensibilité
→ possibilité d'ajouter des fonctionnalités
- efficacité
 - des algorithmes (notation en O)
 - de l'outil utilisé :
 - → le langage de programmation
 - → le paradigme de programmation
- d'autres sont donnés et expliqués par (Fabri, 2000)

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - **Généricité**
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Bibliothèque en trois axes

une bibliothèque est composée :

- d'un catalogue d'algorithmes
ex.: érosion
- de plusieurs types de données
ex.: image 2D et int_u8
- d'outils annexes qui facilitent l'écriture
ex.: point 2D

Bibliothèque en trois axes

Les propriétés de ces axes :

- les axes “algorithmes” et “données” doivent être découplés
- on doit pouvoir étendre la bibliothèque indépendamment :
 - par l'ajout d'algorithmes
 - par l'ajout de types de données
- attention :

ne pas en déduire que l'on s'éloigne du paradigme des objets !

Une question de complexité

une explication que vous verrez partout...

- Soit
 - S le nombre de structures d'images
→ 1D, 2D, 3D, etc.
 - T le nombre de types de données
→ booléen, entier sur 8 bit, sur 16 bit, flottant simple précision, etc.
 - A le nombre d'algorithmes
→ érosion écrite de telle façon, etc.
- Alors
 - on ne veut pas écrire $S \times T \times A$ morceaux de code
 - mais $S + T + A$
⇒ un algorithme est écrit une fois pour toute

...pourtant une erreur se cache derrière cette explication...

Objectifs

Dit autrement :

- on veut conserver le caractère général des algorithmes
- on veut écrire peu de code
- on aime la réutilisation

Taxonomie des utilisateurs

4 catégories :

- assembleurs* ils composent une chaîne de traitements à partir d'opérateurs existants
- dessinateurs* ils travaillent de nouveaux opérateurs
- fournisseurs* ils ajoutent de nouveaux types de données
- architectes* ils maintiennent le cœur de l'édifice

Pour les utilisateurs

4 remarques :

- une bibliothèque doit rester facile d'accès
- les requis en informatique augmentent à chaque catégorie
- seul l'architecte doit connaître la **méta-programmation**...
- dans cet exposé, on va considérer que vous pouvez être n'importe lequel de ces utilisateurs

mais avec des probabilités *a priori* !

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 **Une étude de cas**
 - **Un algorithme**
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

map

- sa description :
 - entrées : input, une image et fun, une fonction
 - sortie : l'image d'entrée
 - corps de l'algo :
pour tout point p faire $\text{input}[p] = \text{fun}(\text{input}[p])$
- rappels :
 - sa transcription informatique doit rester **générale**
 - toute **variation** doit être rendue possible

Une écriture abstraite

elle semble assez “naturelle” :

```
void map(image input, function fun)
{
    point p ...
    for_all (p)
        input[p] = fun(input[p])
}
```

Une écriture abstraite

trois constatations **fondamentales** :

- des **abstractions** apparaissent
→ *image*, *function* et *point*
- une solution viendra de la manipulation de “**types**”
 - → le type du “point” est **dépendant** du type de l'image d'entrée
 - → quel paradigme ? quelle modélisation ?
- *on en oublierait presque quelque chose...*

Famille de types et dépendance

vision simpliste mais exacte :

- on est en présence de “familles” de types
 - image 2D $\rightarrow$ point 2D
 - image 3D $\rightarrow$ point 3D
 - ...
- on souhaite déduire des types à partir d'autres types
 - ex.: *image* $\rightarrow$ *point*
 - ...
- les types
 - sont liés et interagissent ensembles
 - doivent être spécifiés ensembles

mais qu'oublie-t-on ?

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 **Une étude de cas**
 - Un algorithme
 - **Variations**
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Définition d'une "variation"

variation

modification du comportement de l'algo sans changement d'écriture de l'algo

Des variations attendues sont :

- triviales
 - la structure de données change (image 1D, 2D...)
 - les données changent (booléens, entiers, rgb...)
- ou moins triviales...

Les variations

- on ne veut pas “écrire” de variations
 - pas de nouveau code
 - pour ne pas multiplier le code
 - pas de modification de code
 - pour éviter de polluer le code existant
- mais on veut les avoir !
 - le code existant doit les supporter
 - ne pas être limité par le code existant
 - elles sont *implicites*
 - à l'écriture de l'algo
 - via le paradigme de programmation

Premier exemple de variation

- on ne souhaite ne traiter qu'un sous-ensemble de points

```
void map(image input, function fun, predicat pred)
{
    point p ...
    for_all (p)
        if (pred[p])
            input[p] = fun(input[p])
}
```

Second exemple de variation

- on souhaite appliquer le traitement sur la couche 'rouge'

```
void map(image input, function fun)
{
  — input contient des valeurs en RGB
  point p ...
  for_all (p)
    input[p].red = fun(input[p].red)
}
```

De “vraies” variations !

le prédicat :

- argument 'contre' :
on peut tourner l'algo sur toute l'image et tant pis pour les calculs inutiles
- arguments 'pour' :
 - → gain en temps d'exécution
connaissiez-vous un chercheur sérieux qui décrirait un algorithme effectuant volontairement des calculs inutiles ?
le programmeur doit-il être plus bête que le chercheur ?
 - → on n'a quelquefois pas le choix !
et si $fun(x) = 1/x$ avec 0 valeur prise dans l'image ?

De façon plus “philosophique”

avec le rouge :

- argument 'contre' :

on peut d'abord isoler la couche rouge pour la passer à l'algo et tant pis pour les calculs inutiles et la mémoire supplémentaire utilisée

- arguments 'pour' :

- → à la fois gain en temps et en mémoire
ces deux aspects peuvent être critiques dans des domaines comme le temps-réel et l'embarqué ;
veut-on alors vraiment se fixer des limitations à l'utilisation de la bibliothèque ?
- → une fonctionnalité en plus
ne pas fournir cette fonctionnalité signifie ne pas laisser à l'utilisateur le choix entre les deux approches ;
serait-ce considérer qu'il n'est pas à même de choisir ?

Penser “variations”

important car cela permet :

- de garder à l'esprit un grand nombre de cas qui doivent être “possibles”
- donc de ne pas limiter la bibliothèque
→ de “moins” la limiter...?
- donc de mieux abstraire
→ de mieux définir les abstractions
- exercice : *imaginez une variation et demandez-vous si (comment) c'est possible...*

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 **Une étude de cas**
 - Un algorithme
 - Variations
 - **Variantes**
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de "classification"
 - Classification et discriminants "orthogonaux"
 - L'héritage par propriétés

Définition d'une "variante"

variante

écriture alternative d'un l'algorithme

- Une variante naît d'entrées de natures différentes :
 - certaines images sont faiblement quantifiés $\Rightarrow$ LUT
 - certains élt's structurants $\Rightarrow$ algos plus efficaces
- *contre-exemple* de variantes :
 - reconstruction séquentiel, parallèle ou hybride...

Une “vraie” variante (?)

- mais si :
 - l'image à traiter a un nbr de pixels très supérieur au nbr de valeurs prises par ses pixels
 - le calcul lié à la fonction est relativement coûteux
- il est alors plus efficace d'utiliser une LUT (*look-up-table* :

```
void map(image input, function fun)
```

```
{
```

```
  — calcul de la LUT
```

```
  lut l ...
```

```
  value v ...
```

```
  for_all (v)
```

```
    l[v] = fun(v)
```

```
  — calcul de l'image de sortie
```

```
  point p ...
```

```
  for_all (p)
```

```
    input[p] = l[input[p]]
```

```
}
```

- question : est-on en train de parler du **même** algorithme ?

Ou une variation (?)

- oui si :

```
void map(image input, function fun)
{
    function efun(input, fun) — fonction effectivement appliquée
    point p ...
    for_all (p)
        input[p] = efun(input[p])
}
```

- *nota bene* : la fonction “effective” stocke la LUT si besoin.
- du bla-bla vraiment non trivial à comprendre :
 - dans le cas de cet exemple, la variation de l’algorithme est prise en compte par un des “composants” de l’algorithme (efun)
 - c’est exactement ce que nous réalisons par ailleurs avec la généricité grâce à la notion de *point*

Des interrogations

- l'exemple précédent est significatif
- les questions posées sont :
 - quelle est la limite entre variation et variante ?
 - quand doit-on s'arrêter d'être générique ?
- en fait :
 - doit-on poursuivre la quête de l'algorithme unique ?
 - est-on sûr qu'il existera toujours ?
 - n'est-ce pas trop se compliquer la tâche que de vouloir tout faire rentrer dans le même moule ?

Des desiderata raisonnables

on veut les deux comportements :

- une routine générale par algo
 - $\Rightarrow$ pour être générique
 - $\Rightarrow$ supporte les variations
- plusieurs routines pour un même opérateur
 - $\Rightarrow$ pour ne pas compliquer l'écriture générique
 - $\Rightarrow$ gèrent les variantes

fondamentalement

un algo doit supporter plusieurs **formes**

Avant-propos

- pour parler du “comment” il faut :
 - connaître les techniques à notre disposition
 - comprendre leurs différences
- on choisit le langage C++ parce qu'il est :
 - proche du C et répandu
 - connu de tous, accessible donc
 - conçu pour produire des exécutables rapides
 - efficacité des calculs
 - riche en caractéristiques
 - non limitatif, large champ des possibles

Avant-propos

- on va opter pour une vision fonctionnelle
 - un opérateur de TDI est une fonction
 - le plus simple exemple : une procédure en C
 - tout est fonction dans un programme
 - on voit tout bout de code comme une transformation “mathématique”
- approche des langages fonctionnels
 - typer les fonctions
 - comprendre différemment le C++

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 **Un tour de notions fondamentales**
 - **Quatre polymorphismes**
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de “classification”
 - Classification et discriminants “orthogonaux”
 - L'héritage par propriétés

Polymorphisme de coercion

- le type de `double sqr(double d)` est :
“fonction de double $\rightarrow$ double”
- d'un point de vue extérieur,
cette routine peut être appelée :
 - avec un double (copie de l'argument)
 - mais aussi avec un int ou un float
 $\Rightarrow$ coercion = création d'un objet d'un type différent
- elle a plusieurs formes !

Polymorphisme de surcharge

- Autre possibilité :
 - pour une **même** fonction
écrire **plusieurs** routines avec des types différents
 - à l'appel de la fonction
le compilateur détermine quelle routine est appelée

- Avec :

```
int sqr(int i); // 1
float sqr(float f);
double sqr(double d);
```

- l'utilisation `int i = 3; sqr(i);` appelle la routine 1.

Polymorphisme paramétrique

Paramètre

inconnue qui prendra une ou des valeurs à la compilation, suivant les besoins de l'utilisateur

- Exemple :

```
template <typename T>  
T sqr(T v);
```

- ici :
 - 'T' représente un type mais on ne précise pas lequel
 - cette écriture de 'sqr' est une *description* de code
 - les valeurs prises par 'T' sont inconnues...

Polymorphisme paramétrique

- A l'appel de la fonction `sqr` :
 - le compilateur détermine la valeur du paramètre 'T'
 - compile une routine *particulière* avec cette valeur
- Exemple :
 - Si on écrit : `int i = 3; sqr(i);`
⇒ compilation de `sqr<T>` avec T valant **int**
 - Plus loin, le code : `float f = 3.14; sqr(f);`
⇒ autre compilation de `sqr<T>` avec T valant cette fois **float**

Polymorphisme paramétrique

- Au final, on revient au cas de la surcharge :

```
int sqr(int i);  
float sqr(float f);
```

- mais :
 - il n'y a jamais coercion
 - le code de `sqr` a été écrit **une seule fois**

paramétrisme

mode d'écriture de fonctions polymorphes qui offre de la compilation à la demande

Exemple mathématique

- Soit la fonction mathématique :

$$a \in \mathbb{N}^+, f_a : \begin{cases} \mathbb{R} & \rightarrow \mathbb{R} \\ x & \mapsto \sin(ax) \end{cases}$$

- sa traduction est :

```
template <unsigned a>  
float f(float x) { return sin(a*x); }
```

Exemple mathématique

on ne peut (doit) pas confondre :

f_a	fonction paramétrique $f_a(1)$ n'est pas calculable	f	description $f(1)$ ne compile pas
f_2	une fonction (non paramétrique)	$f<2>$	"vraie" fonction, compilable
$f_2(1)$	une valeur	$f<2>(1)$	appel à la fonction, une valeur est renvoyée

Polymorphisme d'inclusion

Héritage

dans les langages à objets, l'héritage de classes introduit une relation de sous-classage (sémantique = “est-un”)

Exemple

- la classe abstraite *Value* a trois sous-classes :
Integer, Float et Double
- soit la routine suivante : `Value& sqr(Value& v)@`
- elle peut être appelée avec une instance de n'importe laquelle des sous-classes ; ex. : `Integer i = 3; sqr(i);`

Récapitulatif

<i>type de polymorphisme</i>	<i>nbre de routines écrites</i>	<i>nbre de routines compilées</i>
coercition	1	1
surcharge	n	n
paramétrisation	1	m
inclusion	1	1

avec m le nombre de cas différents dans l'appel de `sqr`.

Constatations

On a :

- routine efficace $\Leftrightarrow$ compilation dédiée suivant les données
 $\Rightarrow$ surcharge *ou* paramétrisation
- être général $\Leftrightarrow$ abstraire
 $\Rightarrow$ inclusion *ou* paramétrisation
- Donc...

utilisation des polymorphismes

- (variations $\Leftrightarrow$ une écriture générale) $\Rightarrow$ inclusion *ou* paramétrisation
- (variantes $\Leftrightarrow$ plusieurs écritures) $\Rightarrow$ surcharge

Abstraire pour les variations

- abstraire avec l'inclusion :

```
image& map(image& input, function& fun);
```

- abstraire avec la paramétrisation :

```
template <typename I, typename F>  
I& map(I& input, F fun);
```

- concrètement → présentation (AI, 2001)

Ce qu'il faut retenir

Les 4 points importants :

- il y a dualité entre inclusion et paramétrisation
- les outils aident à l'écriture générale d'algorithmes
- plus d'information à la compilation
⇒ de meilleurs performances
- les mots-clefs dont on raffole : **template**, **typedef**

Une seule voie

variations : abstractions performantes = polym. paramétrique

variantes : plusieurs routines = polym. de surcharge

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 **Un tour de notions fondamentales**
 - Quatre polymorphismes
 - **Gestion des variantes et héritage statique**
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de “classification”
 - Classification et discriminants “orthogonaux”
 - L'héritage par propriétés

Le problème de la surcharge

- Avec l'héritage, la surcharge se passe bien :

```
void maj(fruit& f) { f.murir(); } // v1
void maj(animal& a) { a.vieillir(); } // v2
void test() {
    pomme p;
    maj(p); // v1 est appelee
}
```

- Avec le polymorphisme paramétrique, mal :

```
template <typename FRUIT>
void maj(FRUIT& f) { f.murir(); } // v1

template <typename ANIMAL>
void maj(ANIMAL& a) { a.vieillir(); } // v2
// compilation error: redefinition of maj
```

La nature du problème = le typage

- avec l'inclusion :
 - le type de la v1 est : `"maj : fruit  $\rightarrow$  void void"`
 - le type de la v2 est : `"maj : animal  $\rightarrow$  void void"`
- avec la paramétrisation :
 - les deux fonctions ont pour type
`"maj :  $\forall t, t \rightarrow void$ "`
 - on aimerait avoir
 - pour v1 : `"maj :  $\forall t \subseteq fruit, t \rightarrow void$ "`
 - pour v2 : `"maj :  $\forall t \subseteq animal, t \rightarrow void$ "`
- mais le C++ n'accepte pas les paramètres contraints :

```
template <typename F : fruit> // contrainte : F sous-classe de fruit  
void maj(F& f) { f.murir(); }
```

n'est pas du C++...

Solution

- on **mélange** les deux paradigmes :

```
template <typename F>  
void maj(fruit<F>& f) { f.murir(); } // v1
```

```
template <typename A>  
void maj(animal<A>& a) { a.vieillir(); } // v2
```

- on a :
 - A est le type exact de l'instance d'animal a
 - la classe lapin est une sous-classe de animal<lapin>
 - le typage est récursif
par ex. pour v2 : “maj : $\forall t \subseteq \text{animal} < t >, t \rightarrow \text{void}$ ”

Exemple

- un code d'utilisation :

```
void test() {  
    pomme p;  
    maj(p); // v1 est appelee avec F valant pomme  
    lapin l;  
    maj(l); // v2 est appelee avec A valant lapin  
}
```

- au final :
 - maj<pomme> et maj<lapin> sont les 2 routines compilées
 - d'autres appels pourront déclencher d'autres compilations
ex.: maj<marmotte>
 - on a tiré le meilleur des deux paradigmes !

Comparaison de deux types d'héritages

héritage classique :

```
struct animal {  
    virtual void vieillir() = 0;  
    virtual void manger(bouffe&) = 0;  
};
```

```
struct lapin : public animal {  
    void vieillir() { /*code*/ }  
    void manger(bouffe& b) { /*code*/ }  
};
```

- la liaison est automatique
- un lapin devrait manger des carottes...

héritage statique

```
template <typename A>  
struct animal {  
    void vieillir() { /*liaison*/ }  
    void manger(f.bouffe.type(A)& b) { /*liaison*/ }  
};
```

```
struct lapin : public animal<lapin> {  
    typedef carotte bouffe_type;  
    void impl_vieillir() { /*code*/ }  
    void impl_manger(bouffe_type& c) { /*code*/ }  
};
```

- la liaison est manuelle
- un lapin mange des carottes

Comparaison de deux types d'héritages

dans les deux cas :

- la liaison est transparente pour l'utilisateur
- l'héritage est le même
 - pour l'utilisateur
 - pour le programmeur
 - même modélisation
 - ⇒ on a accès aux tech classiques de génie logiciel ⇒
 - on peut penser “classiquement” puis traduire

Réalisation de la liaison

- ex. de liaison statique :

```
template <typename A>
void animal<A>::manger(f_bouffe_type(A)& b)
{
    ((E*)(this))->impl_manger(b);
}
```

- exercice : *trouvez* des exemples liés au TDI de hiérarchies de classes et de méthodes abstraites...

Qu'avons-nous ?

- on ne perd jamais d'information de type *tout le typage est statique*
- on peut à la fois abstraire *et* rester performant
- et surtout *on est dans le domaine de la modélisation OO*
- mieux :
 - on a des **“virtual typedef”**
 - on a de la **covariance**
 - on a des **multi-méthodes**

c'est le paradigme “SCOOP” (MPOOL, 2003)

Les problèmes rémanents

Que se passe-t-il avec ce code ?

```
template <typename E>
struct animal : public any<E> { // 2
    void mange(E::aliment m) { // 3
        // ...
    };
    struct lapin : public animal <lapin> { // 1
        typedef carotte aliment; // 4
        // ...
    };
};
```

Qu'en conclure ?

Les problèmes rémanents

Et avec celui-là ?

```
struct petit_lapin : public lapin {  
    void impl_vieillir() {  
        std::cout << "petit_lapin::vieillir" << std::endl;  
    }  
};  
void test_bis() {  
    petit_lapin l;  
    maj(l);  
}
```

Qu'en conclure ?

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 **Un tour de notions fondamentales**
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - **Les paradigmes successifs d'olena**
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images
 - Tentatives de “classification”
 - Classification et discriminants “orthogonaux”
 - L'héritage par propriétés

De 1999 à 2001

0. “miLeNa”, pour se faire les crocs, images 2D seulement :

```
template <typename T>
void op(image2d<T>& ima) {
    point2d p;
    std::cout << ima[p];
}
```

1. *generic programming* classique (ex.: STD:: et VIGRA)

aucune containte sur les paramètres (aucun héritage) :

```
template <typename I>
void op(I& ima) {
    I::point_type p;
    std::cout << ima[p];
}
```

De 2002 à 2004

2. yo-yo, contrainte par héritage (c'est tout) :

```
template <typename I>
void op(image<I>& ima_) {
    I& ima = exact(ima_);
    I::point_type p;
    std::cout << ima[p];
}
```

3. SCOOP, modèle “objet” statique (avec des bonus) :

```
template <typename I>
void op(image<I>& ima) {
    point_type(I) p;
    std::cout << ima[p];
}
```

2005

4. *OleNa* 1.0, héritage implicite

on ne change rien à la façon d'écrire les algo :

```
template <typename I>  
void op(image<I>& ima) {  
    point_type(I) p;  
    std::cout << ima[p];  
}
```

La spécificité d'OleNa 1.0

- manifestement pas dans l'écriture d'algorithmes
- pour l'instant :
 - on est “pauvre” en abstractions
→ *image*, *point*...
 - on a des classes “évidentes” d'images
→ `image2d<T>`, `image3d<T>`
 - on a toujours pas expliqué la mise en œuvre des variantes
→ `for_all(p) if (pred[p])`
→ `input[p].red` remplace `input[p]`
- on a seulement une solution aux variantes dites “triviales”

La spécificité d'OleNa 1.0

OleNa 1.0

- plus d'abstractions
- plus de classes d'images
- de nombreuses variantes
- l'héritage est implicite

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 **Un tour de notions fondamentales**
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - **Méta-programmation et vision fonctionnelle**
- 5 Les abstractions liées aux images
 - Tentatives de “classification”
 - Classification et discriminants “orthogonaux”
 - L'héritage par propriétés

Buts de cette section

- mieux distinguer compilation et exécution
- entrevoir les possibilités du C++
- comprendre le métier de l'architecte
- s'initier à la méta-programmation
- ne pas être étonné des capacités d'Oléna 1.0

Un bon début en méta-programmation

- Considérons une écriture telle que celle-là :

```
template <typename T>
struct image2d {
    typedef T value_type;
    // ...
};
```

- Soit I un type d'image, on peut définir une fonction qui nous donne le type de ses valeurs :

```
#define f_value_type(I) typename I::value_type
```


Un bon début en méta-programmation

- Cela permet d'écrire :

```
template <typename I>  
void foo(image<I>& ima, f.value_type(I) value) { /* ... */ }
```

et :

```
image2d<float> ima;  
foo(ima, 3); // la coercion est ok
```

- Plus généralement :
une fonction f qui prend P et renvoie R s'écrit :

```
namespace impl {  
  template <nature_de_P P> struct f { /* ici R est édfini */ };  
}  
#define f(P) impl::f<P>::R
```

Les traits

Deux idées :

- regrouper les propriétés d'un type, T , dans une structure *traits*
 - donc disposer d'un (1) type, *traits*, pour représenter des propriétés
 - donc pour les stocker, les encapsuler
- définir la structure *traits* pour chaque T
 - donc *traits* est paramétrée par T
 - donc on dispose au final d'une base de données, les instances *traits* $\langle T \rangle$

Exemple de traits

- La déclaration : **template <typename T> struct traits;**
- Une instance (entrée de la base de données) :

```
template <typename T>  
struct traits < image2d<T> >  
{  
    typedef point2d point_type;  
    typedef T value_type;  
    // ...  
};
```

- Une macro :
#define f_point_type(I) typename traits<I>::point_type

Factoriel n calculé à l'exécution

- La fonction :

```
unsigned fact(unsigned n) {  
    unsigned ret;  
    if (n != 1) ret = n * fact(n - 1);  
    else ret = 1;  
    return ret;  
}
```

- et un appel : `fact(7)`
- c'est dommage car :
 - 7 est connu à la compilation
 - `fact(7)` pourrait être calculé à la compilation
 - on perd donc du temps à l'exécution

Factoriel n calculé à la compilation

- L'implémentation de la fonction :

```
namespace impl {  
    // if ( $n \neq 1$ )  
    template <unsigned n> fact { enum { ret = n * fact<n-1> }; };  
    // else  
    template <> fact<1> { enum { ret = 1 }; };  
}
```

- la fonction : `#define fact(n) typename impl::fact<n>::ret`
- la syntaxe des appels reste inchangée

Changement de la valeur d'un paramètre

- Entrées :
 - un type paramétré $V\langle n, T \rangle$, par ex. `vec<3, float>`
 - un nouveau type de données D , par ex. `int`
- Sortie :
 - le nouveau type paramétré, par ex. `vec<3, int>`
- La fonction à exprimer : `ch_value_type(l, D)`

Changement de la valeur d'un paramètre

Solution :

```
namespace impl {  
  
    template <typename I, typename D>  
    struct ch_value_type;  
  
    template <unsigned n, typename T, typename D>  
    struct ch_value_type < vec<n,T>, D >  
    {  
        typedef vec<n,D> ret;  
    };  
  
    // ...  
}
```

exercice : trouvez des exemples d'application en TDI

L'héritage conditionnel

- On a une classe de vecteurs $\text{vec}\langle n, T \rangle$ avec n la dimension, on veut que cette classe dérive respectivement de `odd` ou de `even` suivant la parité de n .
- On a le code à trous suivant :

```
struct odd {};  
struct even {};
```

```
// ...
```

```
template <unsigned n, typename T>  
struct vec // ...  
{};
```


L'héritage conditionnel : une (première) solution

Une fonction de parité avec héritage de even et/ou odd :

```
template <unsigned> struct parity;  
template <> struct parity<0> : public even {};  
template <> struct parity<1> : public odd {};
```

L'héritage par vec de la parité :

```
template <unsigned n, typename T>  
struct vec : public parity<(n % 2)>  
{};
```

L'héritage conditionnel : une bien meilleure solution

- l'utilisateur écrit la relation d'héritage naturellement :
 - un `vec` est-un `vector`
 - il a ni à écrire ni à entrevoir une once de méta-prog
 - des classes (cachées) vont gérer les difficultés
- d'où :

```
template <unsigned n, typename T>
struct vec : public abstract::vector< vec<n,T> > // so... odd or even
{
    enum { dim = n };
    // ...
};
```

exercice : trouvez des exemples d'application en TDI

Interopérabilité par généralisation

- On veut une fonction générale de *downcast*, `exact(obj)` :
 - qui fonctionne avec des instances d'une hiérarchie statique
 - mais qui fonctionne aussi avec toute autre instance

- Un code "simpliste" qui ne marche pas :

```
template <typename E> E& exact(any<E>& obj) { return obj.exact(); }  
template <typename E> E& exact(E& obj) { return obj; }
```

- On veut écrire :

```
template <typename T>  
exact_type(T)& exact(T& obj) {  
    if (obj "édridge de" any<qqch>)  
        return obj.exact();  
    else  
        return obj;  
}
```

Interopérabilité par généralisation

- Il faut pouvoir tester l'héritage à la compilation...

```
template <typename T> struct make_ptr { static T* ret(); };  
// make_ptr<T>::ret() retourne un pointeur sur T
```

```
template <unsigned n> struct value { char nada[n]; };  
// on a : sizeof(value<n>) == n
```

```
struct selector {  
    template <class E> static value<1> on(any<E>*);  
    static value<8> on(...);  
};  
// selector::on(make_ptr<T>::ret()) ne renvoie pas tjs le même type
```

- La fonction finale :

```
#define is_any(T) sizeof(selector(make_ptr<T>::ret())) == 1
```

Conclusion temporaire

seconde idée clef

on peut (à peu près) tout faire

- on a dépassé des limites du langage / du paradigme OO :
 - paramètres contraints
 - covariance
 - types virtuels
 - + **multi-méthodes**
- *on veut maintenant définir une modélisation OO pour Olena...*

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 **Les abstractions liées aux images**
 - **Tentatives de “classification”**
 - Classification et discriminants “orthogonaux”
 - L'héritage par propriétés

Tentative 1 de classification

Une image peut être :

- 1D (signal)
- 2D (classique, maillage carré)
- 2D avec pavage hexagonal
- 3D
- et pourquoi pas un graphe (de régions)
- ...

Tentative 2 de classification

ou bien (on recommence), une image peut être :

- une image d'étiquettes
 - cas particulier : une image binaire (objet ou fond)
- une image en niveaux de gris
- une image couleur
- une image de champ de déformation
- ...

Tentative 3 de classification

ou encore (on recommence de nouveau...), une image peut être :

- une image d'étiquettes (hum... déjà dit ?!)
- une image de scalaires
- une image de vecteurs
- ...

exercice : imaginez la suite “ou...”

Premier exemple

une instance de `image2d<rgb>` est une :

- image 2D classique
- image couleur
- image de vecteurs ?

oui si **struct** `rgb` : **public** `vec<3,int_u8>` !

Second exemple

- supposons que :
 - on a une image de niveaux de gris issue d'un multi-seuillages
 - ses valeurs sont des entiers, donc des étiquettes
 - une LUT associe un niveau de gris à chaque étiquette
- c'est une :
 - une image en niveaux de gris ?
 - une image d'étiquettes ?
- réponse :
 - une image en niveaux de gris
il suffit de la visualiser pour s'en persuader
 - une image dont les valeurs sont encodées avec une LUT
en mémoire, les valeurs sont stockées par des étiquettes

Des classifications

- Une image peut être :
 - une image d'étiquettes $\rightarrow a::label_image$
 - une image en niveaux de gris $\rightarrow a::greylevel_image$
 - etc.
- Une image (ima) stocke ses valeurs sous la forme de :
 - une image d'étiquettes avec une LUT $\rightarrow a::lut_image$
 - une image de valeurs $\rightarrow a::raw_image$
 - etc.
- Une image du point de vue de ses valeurs peut être :
 - une image de scalaires $\rightarrow a::scalar_image$
 - une image de vecteurs $\rightarrow a::vector_image$
 - une image de données structurées $\rightarrow a::struct_image$
 - etc.

Les variantes

pour “map”, on obtient alors

variante 1 :

```
point p ... — de input
for_all (p)
  input[p] = fun(input[p])
```

variation 3 :

```
lut_elt e ... — de input.lut()
for_all (e)
  e.value = fun(e.value)
```

variante 2 :

```
lut l ... — obj temp
value v ... — de l
for_all (v)
  l[v] = fun(v)
point p ...
for_all (p)
  input[p] = l[input[p]]
```

Les variantes

variante		type abstrait
1	$\Leftrightarrow$	<code>abstract::image<E></code>
2	$\Leftrightarrow$	<code>abstract::lowq_image<E></code>
3	$\Leftrightarrow$	<code>abstract::lut_image<E></code>

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 **Les abstractions liées aux images**
 - Tentatives de “classification”
 - **Classification et discriminants “orthogonaux”**
 - L'héritage par propriétés

Situation classique en modélisation OO

- des discriminants sont orthogonaux :
 - la dimension du support
1D, 2D, 3D...
 - la nature des données
niveaux de gris, couleur...
 - leur encodage
données bruts, LUT...
 - ...
- *exercice : essayez de dessiner un arbre (graphe ?)
d'héritage...*

La solution OO

- On orthogonalise vraiment :
 - on a une hiérarchie séparée par discriminant
 - on n'a plus qu'une classe concrète : image
 - elle possède un attribut par discriminant
 - elle délègue l'implémentation de ses méthodes
- mais ça ne marche pas :
 - une image 2d ne pourrait pas avoir d'opérateur “(row,col)” !
 - ...

Une piste

un exemple de ce que l'on veut :

- la bibliothèque définit la classe *image2d* $\langle T \rangle$
- si T est un type de niveaux de gris
⇒ cette classe dérive de *abstract::greylevel_image*
sinon
⇒ cette classe dérive d'une autre abstraction

troisième idée clef

le programmeur ne peut pas écrire explicitement l'héritage

question : du déjà-vu ?

Suite de la piste

- un autre exemple que ce que l'on veut :
 - on veut pouvoir “masquer” une image
 - limiter les traitements à un masque binaire
Cf. la variation avec “if (pred[p])”
 - et ce pour n'importe quelle image
 - ex.: *image2d*<*T*>, *image3d*<*T*>...
- raisonnement :
 - rappel : on aime les routines génériques
 - idée : on aimerait que le masquage aussi soit générique
 - conclusion :
on ne veut écrire qu'une (1) classe d'image pour cela !

Suite du raisonnement

- objectif double :
 - le moins de code possible (programmeur = fainéant)
 - le plus de fonctionnalités possible (utilisateur = exigeant)
- artefact :

cette classe dérivera de *abstract::image2d* ssi l'image ciblée dérive cette même abstraction

Terminologie

cette classe est une occurrence de **morpheur**,
une classe qui ressemble à une image

D'où vient le paradigme d'OleNa 1.0

- issu des limites de SCOOP
 - d'obscures problèmes de définitions récursives de types...
- issu des limites de l'objet
 - si on n'écrit plus d'héritage, qu'écrit-on ?
- issu de l'idée de “morpheur”
 - un objet qui n'est pas une “vraie” image
- issu de génie logiciel pour le calcul scientifique
 - la transformation statique de *design patterns* → (COOTS, 2001)

Outline

- 1 Avant-propos
 - Présentation
 - Une histoire
- 2 A propos de bibliothèques
 - Généralités
 - Généricité
- 3 Une étude de cas
 - Un algorithme
 - Variations
 - Variantes
- 4 Un tour de notions fondamentales
 - Quatre polymorphismes
 - Gestion des variantes et héritage statique
 - Les paradigmes successifs d'olena
 - Méta-programmation et vision fonctionnelle
- 5 Les abstractions liées aux images**
 - Tentatives de “classification”
 - Classification et discriminants “orthogonaux”
 - L'héritage par propriétés**

Mode d’emploi

- pour chaque type d’images
 - on déclare une liste de propriétés
 - mieux : on récupère des propriétés !
 - des ensembles de propriétés sont déjà définis !
 - on dérive de la classe : *abstract::image_entry*
- l’héritage est alors construit automatiquement
- dans le cas d’un morpheur :
 - on déclare une liste de propriétés
 - on dérive d’une sous-classe de *abstract::image_entry*

Un exemple

Une image 2D classique :

- ses propriétés :

```
template <typename T>
struct props < image2d<T> > : public default_props < topo_2d >,
                               public default_props < buffer2d<T> >
{
    typedef mlc::no_type delegated_type;
    typedef T value_type;
    enum {
        morpher_tag = image_tag::case_unmorphed,
    };
};
```

- le corps de sa définition :

```
template <typename T>
struct image2d : public abstract::image_entry< image2d<T> >
{
    //
```


Mode d’emploi d’un morpheur

- dans le cas d’un morpheur :
 - on déclare les propriétés qu’il modifie
 - on précise sur quel type il s’appuie
- par défaut :
 - un morpheur délègue ses méthodes
 - un morpheur s’apparente à l’identité

Le morpheur identité

- ses propriétés :

```
template <typename I>
struct props < id_morpher<I> > : public props <I> {
    typedef I delegated_type;
};
```

- le corps de sa définition :

```
template <typename I>
struct id_morpher : public abstract::image_morpher< id_morpher<I> >
{
    typedef abstract::image_morpher< id_morpher<I> > super;
    id_morpher(const I& ref) : super(ref) { exact_ptr = this; }
};
```

Le morpheur identité

on fournit du sucre :



```
id_morpher<I> id(abstract::image<I>& ref) {  
    return id_morpher<I>(ref);  
}
```

● on a vraiment : $\text{id}(\text{ima}) \Leftrightarrow \text{ima}$

Une première modification

on veut définir une image dont les valeurs sont
“vues à travers une fonction” :

- cela change le type des valeurs de l’image :

```
template <typename I, typename F>
struct props < image_through_fun<I,F> > : public props <I> {
    typedef I delegated_type;
    typedef typename mlc::fun_info<F>::result_type value_type;
    // ...
};
```

Une première modification

- ainsi que l’accès aux données :

```
template <typename I, typename F>
struct image_through_fun // ...
{
    // ...
    oln_value_type(self)& impl_op_sqbr(const oln_point_type(self)& p)
    {
        return fun(ref[p]);
    }
};
```

- hypothèse : *O/eNa* fournit du sucre...

Une première modification

- l'accès à la composante rouge peut être une fonction (*red*)
- on peut écrire facilement des fonctions :

```
float ma_fonction(int i)
{
    return exp(i * i) / 3.14;
}
```

- donc pourquoi ne pas avoir le code suivant ?

```
image2d<rgb_8> ima("test.ppm");
image2d<float> out = fun(ma_fonction)(red(ima));
```

Une première modification

- parce qu’il y a mieux !

```
image2d<float> out = exp(sqr(red(ima)) / 3.14;
```

- mais que se passe-t-il ?

Une deuxième modification

on veut compter les accès à l'image :

- les propriétés ne changent pas
 - car on ne modifie pas vraiment l'image mais...
- toute méthode définit un espace supplémentaire
 - pour écrire un code qui va s'exécuter avant (*ante*)
 - pour écrire un code qui va s'exécuter après (*post*)
 - par défaut, il ne se passe rien
- il faut réaliser le comptage :

```
template <typename I>
struct count_morpher : public abstract::image_morpher< count_morpher<I> >
{
    // ctor similar to id_morpher plus :
    void impl_op_sqbr_ante() const { ++count; }
    unsigned count;
};
```


Une deuxième modification

exemple d’utilisation :

```
count_image< image2d<rgb> > > c_ima(ima);  
morpho::opening(c_ima //...  
//...  
std::cout << c_ima << std::endl;
```

Conclusion

- Que pouvez-vous seulement imaginer ?
- *O/eNa* 1.0, sortie pour ISMM 2005

Références

- **Obtaining Genericity for Image Processing and Pattern Recognition Algorithms.**

T. Géraud, Y. Fabre, A. Duret-Lutz, D. Papadopoulos-Orfanos, and J.-F. Mangin. In the proceedings of the 15th *International Conference on Pattern Recognition* (ICPR'2000), IEEE Computer Society, vol. 4, pages 816-819, Barcelona, Spain, Sep 2000.

- **Design Patterns for Generic Programming in C++.**

A. Duret-Lutz, T. Géraud, and A. Demaille. In the proceedings of the 6th *USENIX Conference on Object-Oriented Technologies and Systems* (COOTS'2001), pages 189-202, San Antonio, Texas, USA, Jan-Feb 2001.

- **Generic Implementation of Morphological Image Operators.**

J. Darbon, T. Géraud, and A. Duret-Lutz. In the proceedings of the *International Symposium on Mathematical Morphology VI* (ISMM'2002), pages 175-184, Sydney, Australia, Apr 2002.

- **A Static C++ Object-Oriented Programming (SCOOP) Paradigm Mixing Benefits of Traditional OOP and Generic Programming.**

N. Burrus, A. Duret-Lutz, T. Géraud, D. Lesage, and R. Poss. In the Proceedings of the *Workshop on Multiple Paradigm with OO Languages* (MPOOL'03)