Advanced Static Object-Oriented Programming Features: A Sequel to SCOOP

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

January 2006

Outline

Outline

Outline

1. Introduction
2. An actual example
    - The running example
    - Variations
    - Specialization of algorithms
3. SCOOP v1
    - About abstractness and OO v. GP
    - SCOOP basic idioms
    - Virtual types in SCOOP
4. Implicit inheritance
    - The need for SCOOP v2
    - Think different
    - Designing with properties
    - The How-To Section

Outline

1. Introduction

2. An actual example
   - The running example
   - Variations
   - Specialization of algorithms

3. SCOOP v1
   - About abstractness and OO v. GP
   - SCOOP basic idioms
   - Virtual types in SCOOP

4. Implicit inheritance
   - The need for SCOOP v2
   - Think different
   - Designing with properties
   - The How-To Section

Objectives of these slides

These slides aim at:

- presenting a static object-oriented programming paradigm featuring:
  - static typing
  - class inheritance in a new (uncommon) way
  - safe covariance
  - multi-methods
- describing our erstwhile work on that subject

http://www.lrde.epita.fr/cgi-bin/twiki/view/Publications/200310-MPOOL

and explaining why we need new programming concepts

Context of our work

- a scientific numerical computing library
  `http://olena.lrde.epita.fr`
- two main features
  - efficiency:
    large amount of data to process; so the faster the better
  - genericity:
    different input types; yet algorithms should be written once
- clients are scientists (not computer science people)
- another main feature
  - simplicity:
    C-like code from the client point of view

Three axis for library entities

Three kinds of entities in libraries:

$\mathcal{D}$ data types

- for use as algorithms input and output
- ex: types of data structures (containers)

$\mathcal{A}$ algorithms

- main objective of libraries = provide a catalogue

$\mathcal{O}$ other (auxiliary miscellaneous) tools

- to ease data manipulation and for use in algorithms
- ex: iterators

Four kinds of users

- assemblers
    - just compose components (algorithms) to solve a problem
    - use axis $\mathcal{A}$ but know about $\mathcal{D}$
- designers
    - write new algorithms
    - extend axis $\mathcal{A}$ and sometimes $\mathcal{O}$
- providers
    - write new data types
    - mainly extend axis $\mathcal{D}$ and often also $\mathcal{O}$
- architects
    - focus on the library core
    - make the three axis work altogether

Problems of an architect

- how to simultaneously get abstractness and efficiency?
- is there a suitable language to implement theory?
- how to ease library extensibility?
- is there a way to avoid modifications
  when we think about a new fundamental feature?

Solution provided

- a static object-oriented paradigm
- a paradigm complying to standard C++
- a more “declarative” approach of programing
    - class hierarchies are not fully explicit
      so they are partially implicit
    - some inheritance relationships are computed at compile-time
      so we have static hierarchies
- a new way of thinking about class design...

A relevant example

from our applicative domain:

- basic image processing operators are very comprehensive
- their effects on images can be expressed visually

a very simple one but:

- it allows us to point out many difficulties
- it is very significant of what we expect from a scientific software

Outline

Some image types (1/2)

a signal (1D image) with integral values:

12	96	51	4

a 2D image with floating values:

1.2	3.4	5.6
7.8	9.1	2.3
4.5	6.7	8.9

a binary 2D image:

●	○	○
●	○	●
○	●	●

where ○ and ● stand for respectively true (white) and false (black).

Some image types (2/2)

a color (red, green, blue) 2D image:

(102, 31, 84)	(221, 93,125)	(90, 18,164)
(208,138,157)	(230,185,182)	(197,124, 35)

a 2D image whose support is not a rectangle:

	3.4	5.6
	9.1	
4.5		

and also we have:

- 2D images on a triangular grid (pixels are hexagons),
- 3D images,
- and so on...

The algorithm

name:	assign
input:	an image (*ima*) and a value (*val*)
action:	for every point of *ima*, set its value to *val*
output:	*ima* is modified in-place

---

pseudo-code:

```
assign(ima : image, val : value)
{
  for_every (p)
    ima[p] := val
}
```

Outline

Some desired variations (1/4)

We also may want this operator to be partially applied (so that the image is only modified on given regions):

```
assign(ima : image, mask : binary_image, val : value) {
  for_every (p)
    if (mask[p])
      ima[p] := val
}
```

For instance:

$$\left.\begin{array}{lcl} \text{ima} & = & \begin{array}{|c|c|c|c|}\hline 2 & 5 & 1 & 3 \\ \hline\end{array} \\[1em] \text{mask} & = & \begin{array}{|c|c|c|c|}\hline \circ & \bullet & \bullet & \circ \\ \hline\end{array} \\[1em] \text{val} & = & 0 \end{array}\right\} \Rightarrow \begin{array}{|c|c|c|c|}\hline 0 & 5 & 1 & 0 \\ \hline\end{array}$$

Some desired variations (2/4)

We may also want to apply this operator to some component of the input values:

```
assign(ima : image, attr : accessor, val : value) {
  for_every (p)
    attr(ima[p]) := val
}
```

For instance:

$$\left.\begin{array}{lcc} \text{ima} & = & \begin{array}{|c|c|c|}\hline (1,2) & (3,4) & (5,6) \\ \hline\end{array} \\ \text{attr} & = & 1^{\text{st}}\ \text{component} \\ \text{val} & = & 0 \end{array}\right\} \Rightarrow \begin{array}{|c|c|c|}\hline (0,2) & (0,4) & (0,6) \\ \hline\end{array}$$

Some desired variations (3/4)

We may also want operators to display graphically their behavior:

```
assign(ima : image, val : value, display : bool)
{
  for_every (p) {
    ima[p] := val
    if (display)
      refresh_display(ima)
  }
}
```

Some desired variations (4/4)

And why not a mix of the previous variations?

```
assign(ima : image,
       mask : binary_image,
       attr : accessor,
       val : value,
       display : bool)
{
  for_every (p)
    if (mask[p])
    {
      attr(ima[p]) := val
      if (display)
        refresh_display(ima)
    }
}
```

About variations (1/2)

If we implement variations as is:

- we get code bloat
    - we pay the expensive price of writing the combination of variations
    - we end up with too much code to maintain
- we obfuscate the code of algorithms
    - we turn code from simple to error-prone
- but the worst is that...

About variations (2/2)

...

- we have lost an important property of algorithms:
    - algorithms are intrinsically abstract
    - put differently,
      they should be free from implementation details
- we have broken an important software engineering rule:
    - feature addition should be a non intrusive extension
    - clearly,
      we cannot foresee what the next desired variations will be!

A step towards a solution

- an algorithm is written once in its “simple” form
- we modify input data to provide the algorithm with different particular behaviors:
  - for instance

    ima’ := add_mask(ima, mask)
    assign(ima’, val)

  - idem with

    ima’ := first_component(ima)
    ima’ := add_display(ima)

  - and—now why not—with

    ima’ := first_component(add_mask(add_display(ima), mask))

Recap

We want:

- to preserve abstractness in implementing algorithms
  - ⇝ to keep code clean and clear
- to write efficient algorithms
  - ⇝ to have an effective scientific library
- to externally “modify” the behavior of algorithms
  - ⇝ to get flexibility in using algorithms

and as a consequence:

- to provide an easy way to define “modified” data types
  - ⇝ e.g., a masked image is an image + a mask

Outline

Re-considering the notion of algorithm

- an image processing operator sometimes translate into several distinct algorithms
- input act as a selector of the right (or more appropriate) algorithm
- having several algorithms for a functionality:
  - is sometimes mandatory
    (Example: the 'erosion' operator should use respectively '*and*' and '*min*' when input have Boolean and scalar values.)
  - or just allows for enhancing efficiency

An another image type

A very common image type is the association of data with a look-up-table (LUT); for instance:

$$\text{ima} = \left\{ \text{data} = \begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 1 & 1 & 2 \\ \hline 2 & 2 & 2 \\ \hline \end{array}\,,\ \text{lut} = \begin{array}{|lcr|}\hline 1 & \rightarrow & (102, \ 31, \ 84) \\ \hline 2 & \rightarrow & (221, \ 93, 125) \\ \hline 3 & \rightarrow & (208, 138, 157) \\ \hline \end{array} \right\}$$

which means that this image actually is:

$$\text{ima} = \begin{array}{|c|c|c|}\hline (102, \ 31, \ 84) & (208, 138, 157) & (102, \ 31, \ 84) \\ \hline (102, \ 31, \ 84) & (102, \ 31, \ 84) & (221, \ 93, 125) \\ \hline (221, \ 93, 125) & (221, \ 93, 125) & (221, \ 93, 125) \\ \hline \end{array}$$

A second algorithm (1/2)

The ’assign’ functionality is better written like:

```
assign(ima : image_with_lut, val : value)
{
  for_every (v) // values of ima’s lut
    v := val
}
```

the call “assign(ima, black)” ends up with:

$$\text{ima} \;=\; \left\{ \text{data} = \begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 1 & 1 & 2 \\ \hline 2 & 2 & 2 \\ \hline \end{array},\; \text{lut} = \begin{array}{|cccccc|}\hline 1 & \rightarrow & (& 0, & 0, & 0) \\ \hline 2 & \rightarrow & (& 0, & 0, & 0) \\ \hline 3 & \rightarrow & (& 0, & 0, & 0) \\ \hline \end{array} \right\}$$

A second algorithm (2/2)

this second algorithm also accepts variations so the call "assign(first_component(ima), 0)" ends up with:

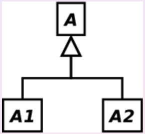
$$\text{ima} = \left\{ \text{data} = \begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 1 & 1 & 2 \\ \hline 2 & 2 & 2 \\ \hline \end{array}, \ \text{lut} = \begin{array}{|cccr|}\hline 1 & \rightarrow & (& 0, \ 31, \ 84) \\ \hline 2 & \rightarrow & (& 0, \ 93, 125) \\ \hline 3 & \rightarrow & (& 0, 138, 157) \\ \hline \end{array} \right\}$$

finally we have both:

assign(ima : image, val : value); *// general case*
assign(ima : image_with_lut, val : value); *// special case*

Use cases of specializations (1/2)

Consider the abstract class hierarchy:



class A { /∗ ... ∗/ };

class A1 : **public** A { /∗ ... ∗/ };
class A2 : **public** A { /∗ ... ∗/ };

such as $A = A_1 \cup A_2$, which means that:

- there cannot be another sub-class of A
- an object of type A is either a A_1 or a A_2.

Use cases of specializations (2/2)

There are two different ways of defining specializations:

```
// bar
void bar(A1& a) { /∗ ... ∗/ }
void bar(A2& a) { /∗ ... ∗/ }
```

```
// baz
void baz(A & a) { /∗ ... ∗/ }
void baz(A1& a) { /∗ ... ∗/ }
```

both bar and baz are functionalities of *A* but

- bar is defined on every disjoint subsets of *A*,
- whereas baz is defined
    - by a (default) general implementation
    - and a specialized impl for a particular subset of *A*

Recap

We want:

- to specialize algorithms
  - ⇝ to get the higher efficiency we can
- to show a facade (one functionality) to the client
  - ⇝ to keep specializations transparent for the client

and as a consequence:

- to be able to write multi-methods
  - ⇝ e.g., an operation that dispatches w.r.t. its arguments

Outline

OO and genericity

Object-orientation and genericity are great

- having classes means:
  - encapsulation
  - information hiding
- having genericity means:
  - define a class with universal quantification
  - e.g., image2d$<$T$>$ is a 2D image (a container)
    it is defined once, for all T,
    T being the type of contained data

An alternative to handle abstractions (1/4)

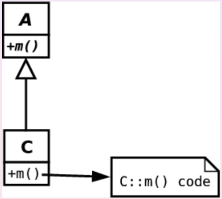
A duality exists between

- class inheritance:
  - named typing
  - inheritance relationship is explicit
  - abstractions = abstract classes (or interfaces)
  - method binding is often solved at run-time
- parametric polymorphism:
  - structural typing
  - no inheritance is required
  - abstractions = parameters
  - method binding can be solved at compile-time

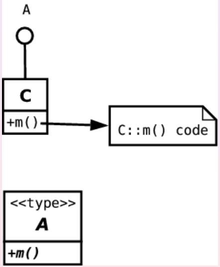
An alternative to handle abstractions (2/4)

The following couple of class designs

with class inheritance:



and without:



are translated into C++ by...

An alternative to handle abstractions (3/4)

with class inheritance:

```
class A { // ...
  virtual void m() = 0;
};

class C : public A { //...
  virtual void m() {
    // C::m code
  }
};
```

and without:

```
class C { // ...
  void m() {
    // C::m code
  }
};
```

An alternative to handle abstractions (4/4)

and the main difference appears in the writing of algorithms

with class inheritance:

```
void foo(A& a) {
  a.m();
}
```

where A is an abstract class

and without:

```
template <class A>
void foo(A& a) {
  a.m();
}
```

where A is now a parameter

Pros for object-orientation

- Pros for classes:
  - they provide a good way to think about domain entities
  - and a proper “abstraction-like” level
- Pros for class inheritance:
  - a practitioner already has names for the domain objects
    $\leadsto$ so abstractions and concrete entities can be named
  - she definitely knows the definitions of abstractions,
    $\leadsto$ so abstract classes are perfect for that
  - she always knows the “is-a” relationship between objects
    $\leadsto$ so inheritance is (seems) trivial

Class inheritance versus generic programming

OO means “class inheritance” and GP stands for “generic programming”

- efficiency
  - is great in GP but poor in OO
  - the abstraction cost of OO is a $\times\alpha$ at execution-time
- overloading
  - comes easily thanks to OO abstractions but is limited in GP
  - is featured by many mainstream OO languages
- multi-methods
  - look intuitive in the OO context but are difficult to get in GP
  - however they are not featured by mainstream OO langs

Temporary conclusion

We want the best of both worlds (OO and GP):

- abstract classes
  - ⇝ so interfaces are clearly defined
- class inheritance
  - ⇝ so classes are explicitly related to each other
- parameterization
  - ⇝ so programs are efficient at run-time
- static typing
  - ⇝ so errors are pointed out at compile-time

so we have defined a Static Object-Oriented Paradigm (SCOOP), version 1.

Different approaches of abstractness

abstractness:

// OO-style
void foo(abstraction& a);

⇝ through abstract classes

here the class “A” is renamed as “abstraction”

// GP-style
template <**class** A>
void foo(A& a);

⇝ through parameters

so on this slide “A” is always a parameter

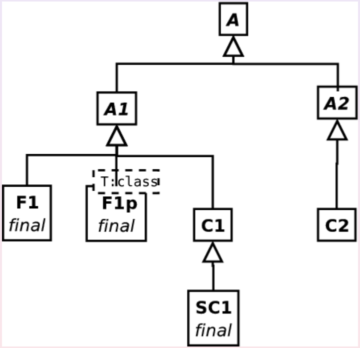
// SCOOP-style
template <**class** A>
void foo(abstraction<A>& a);

⇝ simultaneously through both abstract classes and parameters

Outline

A class hierarchy translated in SCOOP

Let us consider this class hierarchy:



we want to translate this hierarchy into a static one...

Abstract classes (1/3)

To achieve (strong) static typing, the exact type of an object should never be forgotten.

Example:

- an elephant (concrete class) is an animal (abstract class)
- the concept of animal translates into a class parameterized by its exact type:

  template <**class** E> **class** animal { /∗...∗/ };
- an object whose type is elephant derives from animal<elephant>

In the following, E always denotes the “exact type”.

Abstract classes (2/3)

The abstract class at the top of a hierarchy derives from any<E> to inherit some equipment.

More precisely:

- the 'any' class provides a couple of methods, named exact, that performs a downcast of the target object toward its exact type
- we have

```
template <class E>
class any {
public:
  E& exact() { return *(E*)(void*)this; }
  const E& exact() const { // likewise...
};
```

Abstract classes (3/3)

Classes propagate the exact type through inheritance.

More precisely

- starting a static hierarchy in SCOOP from a top class A:

```
template <class E>
class A : public any<E> { // ...
};
```

- setting inheritance between two abstract classes:

```
template <class E>
class A1 : public A<E> { // ...
};
```

Final concrete classes

Defining a final concrete class follows a particular idiom.

Precisely

- between a final concrete class and an abstract class:

```
class F1 : public A1< F1 > { // ...
};
```

- even when the final concrete class is parameterized:

```
template <class T>
class F1p : public A1< F1p<T> > { // ...
};
```

Non final concrete classes

Defining a non-final concrete class follows a particular idiom.

Precisely

- C1 is a non-final concrete class deriving from A1:

```
template <class E = itself> // "itself" is a special type
class C1_ : public A1< C1_<E> > { // ...
};
```

- and the client can literally write “C1” thanks to:

```
typedef C1_<itself> C1;
```

- sub-classing C1 is then possible:

```
class SC1 : public C1_< SC1 > { //...
}; // here SC1 is a final class
```

Methods

An abstract method is statically bound to its proper implementation.

More precisely:

- the programmer should manually code the binding

```
template <class E>
class abstraction { // ...
  int meth(int args) {
    return this->exact().impl_meth(args);
  }
};
```

- method implementation should use the impl_ prefix

Putting all together

OO:

```
class A { // ...
  virtual void m() = 0;
};

class B : public A { //...
  virtual void m() {
    // B::m code
  }
};


void foo(A& a) {
  a.m();
}
```

SCOOP:

```
template <class E>
class A : public any<E> { // ...
  void m() { this->exact().impl_m(); }
};

class B : public A<B> { //...
  void impl_m() {
    // B::m code
  }
};

template <class E>
void foo(A<E>& a) {
  a.m();
}
```

About algorithms in SCOOP

An algorithm is turned into a procedure (C-like function):

- their variations are handled through inheritance
  - $\leadsto$ the procedure behavior changes with the input types
- their specializations can be handled through multi-methods
  - $\leadsto$ several procedures share the same name but not the same code

Just like a regular method,
a multi-method is statically bound to its proper implementation.

Multi-methods (1/2)

For instance, for bar and baz multi-methods:

- first provide their implementation sets

```
namespace impl
{
  // bar
  template <class E> void bar(A1<E>& a) { /* code dedicated to subset A1... */ }
  template <class E> void bar(A2<E>& a) { /* code dedicated to subset A2... */ }
  // baz
  template <class E> void baz(A<E>& a) { /* general code (default)... */ }
  template <class E> void baz(A1<E>& a) { /* specialized code... */ }
}
```

- then the multi-method facades, which perform the binding

```
// bar
template <class E> void bar(A<E>& a) { impl::bar(a.exact()); }
// baz
template <class E> void baz(A<E>& a) { impl::baz(a.exact()); }
```

Multi-methods (2/2)

this multi-method idiom naturally

- works with multiple arguments
- allows the compiler to point out potential ambiguities such as in:

```
namespace impl {
  template <class T, class U> void oops(A1<T>& t, A<U>& u) { /∗ ... ∗/ }
  template <class T, class U> void oops(A<T>& t, A2<U>& u) { /∗ ... ∗/ }
}
template <class T, class U>
void oops(A<T>& t, A<U>& u) { impl::oops(t.exact(), u.exact()); }

int main() {
  C1 c1; C2 c2; // with C1 and C2 respectively deriving from A1 and A2
  oops(c1, c2);
}
```

What have we done?

we have

- static class hierarchies
  - ⇝ meaning that abstractions keep track of object exact type
- parametric routines with constrained genericity
  - ⇝ so mixing overloading and genericity is now easy

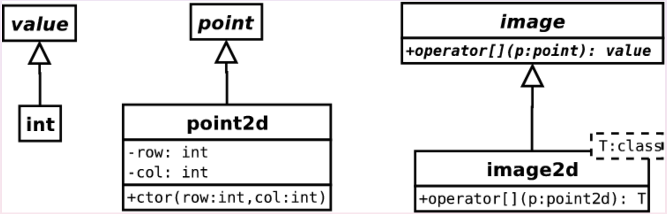
considering **template**<**class** T> **void** routine(A<T>& arg)

- arg can be of any type T being a subclass of A
  - ⇝ more precisely, T is a subclass of A<T>
- this kind of recursive bound is theorically sound
  - ⇝ it is known as F-bounded parametric polymorphism

Outline

Covariant methods (1/2)

The following design seems reasonable:



that’s because...

Covariant methods (2/2)

...many methods are expected to behave in a covariant way!

for instance in:

```
class image { // ...
  virtual value& operator[](const point& p) = 0;
};

template <class T>
class image2d : public image { // ...
  virtual T& operator[](const point2d& p) { /* impl... */ }
};
```

the type of p is point2d in the operator implementation, whereas it is point (base class of point2d) in the abstract interface.

Covariance

- C++, such as many languages, does not support covariant methods
  - ⇝ such feature is proven to be not type-safe!
- the covariant behavior can be emulated but a run-time test is required:

```
T& image2d<T>::operator[](const point& p)
{
  const point2d* ptr = dynamic_cast<const point2d*>(&p);
  if (ptr == 0) throw covariance_error;
  const point2d& p2 = *ptr;
  // here p2 has the proper type
  // ...
}
```

- however, covariance can be safe in a static context
  - ⇝ since types are known at compile-time, covariance can be type-checked

Extended C++ (1/3)

A solution, based on “virtual types”, is here expressed with an extended C++ syntax

an abstract class declares virtual types and thus can use them in methods:

```
class image
{
public:
  // virtual types declarations:
  virtual typedef value value_vt;
  virtual typedef point point_vt;

  // a method using virtual types:
  virtual value_vt& operator[](const point_vt& p) = 0;
};
```

Extended C++ (2/3)

The former declaration:

virtual typedef value value_vt;

means that the value virtual type should be a subclass of the value abstraction.

Another way to extend C++ could be to define abstract virtual types with the ”`= 0`” syntax:

virtual typedef value_vt = 0;

and a constrain upon a virtual type, depending on inheritance, could be expressed with the ”`: public`” syntax, such as in:

virtual typedef value_vt = 0 : **public** value;

Extended C++ (3/3)

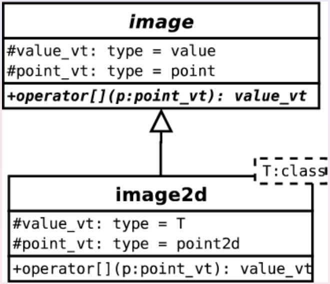
a subclass should provide definitions for abstract virtual types and/or override inherited definitions:

```
template <class T>
class image2d : public image
{
public:
  // virtual types definitions:
  virtual typedef T value_vt;
  virtual typedef point2d point_vt;

  // method implementation:
  virtual value_vt& operator[](const point_vt& p) {
    // here the type of p is point2d
    // ...
  }
};
```

virtual types substitution follows subclassing

OO diagram with virtual types

Finally we end up with:



and the polymorph method now looks invariant
(yet still behaves in a covariant way)

Attempt in standard C++

The natural translation into SCOOP gives:

```
template <class E>
class image : public any<E> {
public:
  typedef typename E::value_vt value_vt;
  typedef typename E::point_vt point_vt;
  value_vt& operator[](const point_vt& p) { return this->exact().impl_op(p.exact()); }
};

template <class T>
class image2d : public image< image2d<T> > {
public:
  typedef T value_vt;
  typedef point2d point_vt;
  value_vt& impl_op(const point_vt& p) { /* impl... */ }
};
```

which does not work since these classes are mutually recursively defined.

Equipment for virtual types

To break recursion

- virtual types are defined separately from their corresponding class
- a traits class is used to encapsulate virtual types definitions.

for that, a tiny equipment is provided:

```
struct undefined;
template <class T> struct traits;
#define vtype(T,V) typename traits<T>::V##_vt
```

where vtype is a macro to resolve virtual type value; for instance:

“vtype(E, value)” means “**typename** traits<E>::value_vt”

Virtual types in SCOOP (1/3)

- first the class to be defined is declared:

```
template <class E> class image; // forward declaration
```

- then virtual types are declared by a traits class:

```
template <class E>
struct traits < image<E> > // specialization
{
  typedef undefined value_vt;
  typedef undefined point_vt;
};
```

- at that point, the virtual types are not yet defined

  an (abstract) image cannot tell what its effective value_vt and point_vt are

Virtual types in SCOOP (2/3)

Last the class can be defined:

```
template <class E>
class image : public any<E> {
public:
  vtype(E, value)& operator[](const vtype(E, point)& p) {
    return this−>exact().impl_op(p);
  }
};
```

where

- the calls vtype(E,something) are substituted at compile-time by the proper types
- these types are expected to be provided by subclasses of image

Virtual types in SCOOP (3/3)

The same scheme is used for the derived class:

```
// forward declaration:
template <class T> class image2d;

// traits specialization:
template <class T>
struct traits < image2d<T> > : public traits< image< image2d<T> > >
{
  typedef T value_vt;
  typedef point2d point_vt;
};

// class definition:
template <class T>
class image2d : public image< image2d<T> > {
public:
  T& impl_op(const point2d& p) { /* impl... */ }
};
```

Conclusion

Several remarks:

- for virtual type definitions to be inherited, the traits should reproduce the same inheritance tree than their corresponding classes
  it works because in C++ `typedefs` are inherited!
- in our example, image2d is a final class so its interface can directly use the virtual type values (and avoid calling vtype)

however SCOOP v1, as presented here, does not fulfill all our requirements...

Outline

A quick refresh(1/2)

Remember that:

- we are in a static context
  - ⤳ all types are known at compile-time
- we define class hierarchies like in classical OO
  - ⤳ with abstract classes, their interface, and inheritance
- ***we want to design classes built over other classes***
  - ⤳ e.g., a masked image is an image + a mask
  - ⤳ e.g., an image with a display attached to
  - ⤳ ...

A quick refresh (2/2)

Generic programming (such as in the standard library of C++ and so on) is a solution to this combinatorial problem:

an algorithm should work on many data types
yet it should be written once and be efficient at run-time

with

- $\mathcal{A}$ algorithms
- $\mathcal{D}$ data types = $\mathcal{S}$ structure types $\times$ $\mathcal{T}$ value types

it comes that

- one should only define ($\mathcal{A} + \mathcal{S} + \mathcal{T}$) entities
- and then $1\ \mathcal{A} \Leftrightarrow \mathcal{S} \times \mathcal{T}$

Introducing morphers

Let us call morpher a class defined from another class put differently, a morpher is a generic class built upon another class

with

- $\mathcal{M}$ morphers

it comes that

- one should only define $(\mathcal{A} + \mathcal{S} + \mathcal{T} + \mathcal{M})$ entities
- and then $1\ \mathcal{A} \Leftrightarrow (\mathcal{S} \times \mathcal{T})^{\mathcal{M}^*}$

The case of morphers (1/3)

First let us introduce an abstract subclass of image:

```
// top class of the image hierarchy
class image { /∗ ... ∗/ };

// the new abstract class for 2d images
class image_2d : public image { /∗ ... ∗/ };

// a concrete image class
template <class T>
class image2d : public image_2d { /∗ ... ∗/ };
```

having abstract subclasses means:

- extended interfaces
- somehow specialized behaviors
- concepts more precise than just “image”

The case of morphers (2/3)

Let us introduce a morpher that works by delegation:

```
class masked_image : public image
{ //...
  value& operator[](const point& p) {
    assert(mask[p]); // test and...
    return this->ima[p]; // delegate
  }
  image& ima; // object to delegate to
  image& mask;
};
```

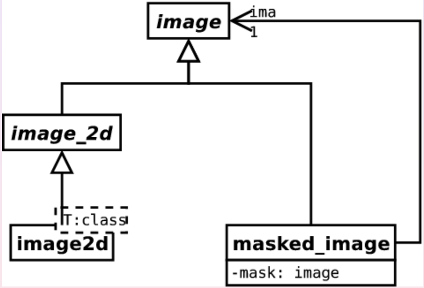
```
// routine to associated a mask
// with an image:

masked_image&
add_mask(image& ima, image& mask)
{
  return *new masked_image(ima, mask);
}
```

with that design we can have:

```
image2d<int> i_2d; image2d<bool> m_2d; //...
image& ima = add_mask(i_2d, m_2d);
point2d p(5,1);
cout << ima[p] << endl; // ok
```

The current design (1/2)

The corresponding UML class diagram is the following:



so an “masked image” does not derive from the 2D image abstraction

The current design (2/2)

thus the following sample code

image_2d& ima = add_mask(i_2d, m_2d);

is not valid...

yet, in that case, the result of “add_mask” should be a 2d image!

The case of morphers (3/3)

Actually

- we want to translate a morpher into one single class
- in the static context:
  - the masked image class looks like

```
template <class I, class M>
class masked_ /* here some code has been deleted */ { //...
  I& ima;
  M& mask;
};
```

  - e.g., we have `masked_< image2d<int>, image2d<bool> >`

we want to say that:
when `I` is 2D then `masked_<I,M>` is 2D

The problem with morphers

the facts are:

- morphers should be implemented by delegation
  - ⇝ because using mixins cannot work property (just trust me on that!)
- when `I` has a specific property, then `a_morpher_based_on<I>` should not ignore it
  - ⇝ a “2D image plus a mask” should be a 2D image...
- delegation does not transfer “properties”
  - ⇝ so does not transfer inheritance (in our example `image_2d`)

A solution for morphers

for morphers we want a mechanism:

- that relies on delegation
- that acts like mixins
- that is close to type inference
- that is easily extendable without intrusion
- that can be written in static OO C++

Example

in the following pseudo-C++ “SCOOP v2”-like code:

```
class image { //...
};

class image_2d : public image { //...
  value& operator()(int row, int col) = 0;
};

class masked_image
  : public image_entry { //...
  // no operator()(int, int) is written here
  // since this class is generic
};
```

```
masked_image&
add_mask(image& ima, image& mask) {
  return *new masked_image(ima, mask);
}

int main() {
  image2d<int> i; image2d<bool> m; //...
  image_2d& ima = add_mask(i, m); // (a)
  cout << ima(5,1) << endl; // (b)
}
```

the class `masked_image` automatically

- inherits from `image_2d` so line (a) is ok
- and delegates the operator call of line (b)

SCOOP v2 in a few words

the cornerstone of SCOOP v2 is:

inheritance is not fully explicit
(so inheritance is partially implicit)

more precisely:

- we declare that a concrete class belong to a hierarchy
  - ⇝ `masked_image` derives from a special class, `image_entry`
- we do not explicitly precise the abstract image subclasses from which it derives
  - ⇝ we cannot explicitly write from which class derives `masked_image`
  - ⇝ but a masked 2d image will derive from the `image_2d` abstract class

Outline

1. Introduction
2. An actual example
   - The running example
   - Variations
   - Specialization of algorithms
3. SCOOP v1
   - About abstractness and OO v. GP
   - SCOOP basic idioms
   - Virtual types in SCOOP
4. Implicit inheritance
   - The need for SCOOP v2
   - Think different
   - Designing with properties
   - The How-To Section

Programing with properties (1/3)

in SCOOP v2

- a class is defined along with a collection of types, the so-called properties
  - ⇝ a property is not just a trait associated to a class
- a concrete class can enter a hierarchy
  - ⇝ for that, the class should derive from the hierarchy entry

    `image_entry` for the `image` hierarchy
- inheritance for this class is automatically plugged from its properties
  - ⇝ so inheritance is not fully explicit

Programing with properties (2/3)

before:

```
class image { //...
  typedef point_type = 0;
};

class image_2d : public image { //...
  typedef point2d point_type;
};

template <class T>
class image2d : public image_2d { //...
  // point_type is already defined here
  // but we explicitly write inheritance
};
```

with properties:

```
class image { //...
  typedef point_type = 0;
};

class image_2d : public image { //...
  // optional: check point_type == point2d;
};

template <class T>
class image2d : public image_entry { //...
  typedef point2d point_type;
  // we define point_type
  // but now inheritance can be implicit
};
```

Programing with properties (3/3)

`image_entry` has to define how to solve inheritance:

```
class image_entry : public
  image_2d when point_type == point2d
  // and so on for other inheritance rules...
{};
```

and now we can easily write:

```
template <class I, class M>
class masked_ : public image_entry { //...
  typedef I::point_type point_type;
};
```

thus, when `I` is 2D, `masked_<I,M>::point_type` is `point2d`
so `masked_<I,M>` inherits from `image_2d`

remember that the inheritance mechanism is performed at compile-time!

First conclusion on properties

we now do **NOT** say:

`image2d<T>` works with `point2d` because it derives from `image_2d`

but conversely we do say:

`image2d<T>` derives from `image_2d` **because** it works with `point2d`

using properties:

- allows to just roughly draw inheritance
    - we just have to write “`image2d<T>`” is an image
    - so we can get rid of inheritance details
    - and we can have morphers work properly
- reverses the way we think about inheritance

A few remarks

yet this solution remains partially unsatisfactory

- a type is manually transfered (and that's really bad!)

  in the previous code the designer explicitly writes that

  the value of `::point_type` is transfered from `I` to `masked_<I,M>`

- we definitely cannot know the list of types to transfer

  an extension will introduce some `::new_type`...

so we need a solution

- to **express** the notion of "set of properties (SoP) of a type"
- to **transfer** a SoP from one type to another
- to **extend or modify** a SoP in a non-intrusive way

Outline

Hierarchy design (1/3)

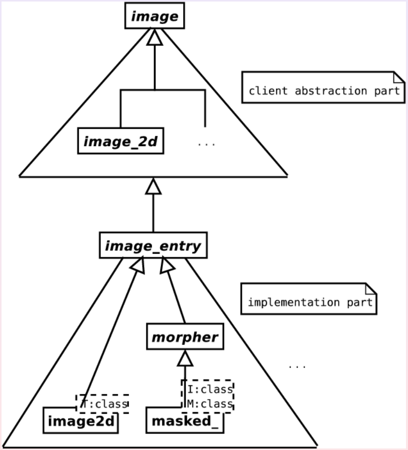
A class hierarchy has two important classes:

- the top abstract class
  - ⇝ `image` in our example
- the hierarchy entry class
  - ⇝ `image_entry` in our example

The other classes belong to one of these categories:

- client abstractions
  - ⇝ for instance `image_2d`
- concrete classes
  - ⇝ for instance `image2d<T>` or `masked_<I,M>`
- implementation abstract classes...

SCOOP v2 hierarchy design



Hierarchy design (2/3)

client abstractions:

- are defined in-between respectively the hierarchy top and entry classes
- are part of the application domain
- can use but do not define properties

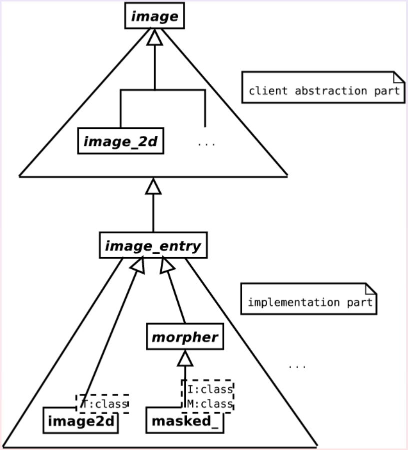
concrete (implementation) classes:

- are subclasses of the entry class
- are also used by the client (the assembler)

implementation abstract classes:

- are subclasses of the entry class and base classes for concrete classes
- are used to factor code and definitions of properties
  ⇝ so they shall be understood as implementation details
- are for provider and architect eyes only

the return of SCOOP v2 hierarchy design



Hierarchy design (3/3)

so we have two parts:

- the client abstraction part

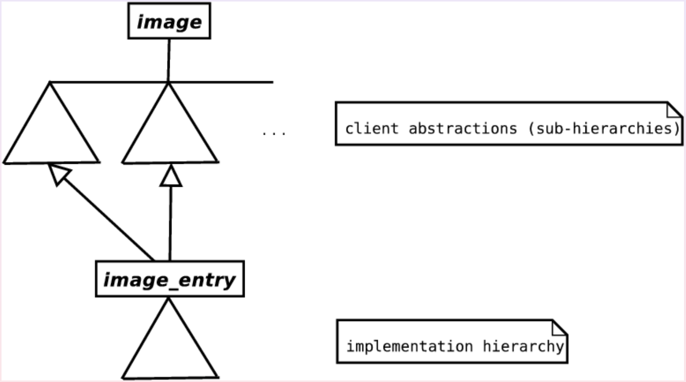
  ⇝ can be organized into "parallel sub-hierarchies"

- the hierarchy entry class as separator

- and the implementation part
  - with implementation abstract classes
  - and concrete classes

  ⇝ this part can be organized into a judicious "implementation hierarchy"

A typical class diagram in SCOOP v2



Conclusion on properties and hierarchy

properties

- are defined in the implementation part
- behave as virtual types in the implementation hierarchy

the client abstraction part and the implementation part

- address two well-separated issues (domain v. design)

both parts are extendable independently,
whatever the extension is horizontal or vertical
(and that's great!)

The extension process (1/2)

extending a class hierarchy means:

- adding a new concrete class
  - this new class has to implement abstract methods
  - and to set the values of properties
- adding a new property to this hierarchy
  - all concrete classes have to value this property
  - setting values is performed in the implementation part
- adding something in a client abstract class
  - either a new property or a new method
  - this new entity is thus not defined by all concrete classes

The extension process (2/2)

extending a class hierarchy also means:

- adding a new sub-hierarchy of abstract classes
  - these new classes derive from the hierarchy top class
  - this new sub-hierarchy can be orthogonal to existing ones
  - a new inheritance rule has then to be defined
- adding a method definition
  - corresponding to an abstract method
  - for any class of the hierarchy

all these extensions are non-intrusive
(so that's great!)

Outline

1. Introduction
2. An actual example
   - The running example
   - Variations
   - Specialization of algorithms
3. SCOOP v1
   - About abstractness and OO v. GP
   - SCOOP basic idioms
   - Virtual types in SCOOP
4. Implicit inheritance
   - The need for SCOOP v2
   - Think different
   - Designing with properties
   - The How-To Section

Forewords

the solution we present conforms to standard C++

and it's not such hard core C++...

(however the following slides are rated R)

About functions from type(s) to type (1/4)

Just realize that:

- writing the following C++ code

  ```
  template <class T> struct foo { typedef undefined ret; };
  ```

  means that `foo` is a function
  - taking a type as argument (`T`)
  - and returning a type (`foo<T>::ret`)
- for instance,
  getting the value type from an image type is a function
  - with `image2d<int>` the value type is `int`
  - in that case, the name of the `foo`-like function is `value_type`

About function from type(s) to type (2/4)

Also just realize that:

- the specialization

  ```
  template <> struct foo <A> { typedef float ret; };
  ```

  defines the result of the function for the input type A

- and the following structure

  ```
  template <>
  struct types <A> {
    typedef float bar;
    typedef bool  baz;
  };
  ```

  means

  - that `bar` can be considered as a function (from type to type)
  - and that we pack several **definitions** together

About functions from type(s) to type (3/4)

but

- do **not** confuse function definitions with function results
  - for virtual types, definitions are subject to substitution...
  - calling a function is performed by a particular syntax
- so calling a function from type(s) to type can have different behaviors:
  - the basic matching imposed by C++ template specialization
    ⇝ and this kind of matching is rather limited
  - a client-defined pattern matching for each function
    ⇝ just like in a functional language
  - and the "virtual type" mechanism that relies on inheritance
    ⇝ that's the one we are interested in for properties

About functions from type(s) to type (3/4)

for example:

- with

  ```
  template <class I>
  struct set_types < image<I> > {
    typedef undefined value;
    //...
  };
  ```

- when `I` is `image2d<int>`
- the **definition** of value type for `image<I>` gives `undefined`
- but the value type **result** provided by `typeof(image<I>, value)` is `int`

C++ contraction (1/2)

in the following we use some syntactical contractions:

standard C++	*shortened C++*
`template<class T> struct foo;`	`decl ’T foo;`
`template<class T>` `struct foo {...};`	`’T foo<T> {...};`
`template<class T>` `struct foo < image2d<T> > ...`	`’T foo< image2d<T> > ...`
`: public base1, public base2`	`: base1, base2`
`and_< eq<T1,T2>, is_a(T3,T4) >`	`T1 == T2 and T3 <# T4`
`predicate::ensure();`	`check predicate;`
`current`	the class we are currently defining
`typeof(current, value)`	value@
`{this->exact().impl_m();}`	`= 0;`
`...`	some code has been deleted

C++ contraction (2/2)

in the following we use some syntactical contractions:

standard C++	*shortened C++*
`typedef float alias_type;`	`alias = float;`
`{ typedef float ret; };`	`= float;`
`typename foo<T>::alias_type`	`foo(T).alias`
`typename foo<T>::ret`	`foo(T)`

understand that

- `foo<T>` is the structure type
- `foo(T).alias` and `foo(T)`
    - are access to the structure contents (a `typedef`)
    - but are **not** the function resolution of the virtual type foo

Some equipment (1/3)

flags to handle the result of functions from type(s) to type:

flag	*meaning*
`undefined`	the type is not defined yet (so it has only been declared... as being "undefined")
`no_type`	there is no type (no relevant type can be returned)
`not_found`	the type has not been found (it cannot be retrieved)

this sample code:	gives:
decl A; decl B;	
'T foo<T> = undefined; foo<A> = float;	check foo(B) == undefined; check foo(A) == **float**;
'T types<T> {}; types<A> { bar = **double** };	check types(A).baz == not_found; check types(A).bar == **double**;

Some equipment (2/3)

A key tool is implicitly used when we write:

```
check types(A).baz == not_found;
```

actually

- trying to read the typedef `baz_type` in the structure `types<A>` shall compile even if this type definition does not exist!
- for that we rely on the C++ SFINAE rule
  - ⇝ you should know that “Substitution Failure Is Not An Error”!
- a piece of meta-program is behind the writings like `foo(T)` and `foo(T).alias`
  - ⇝ the meta-function is `typedef_of(type, alias)`

Some equipment (3/3)

Some functions (for any type T) are proposed as an equipment for the architect and the provider:

function	*meaning*
set_super(T)	to declare the immediate base class of T
super(T)	to get the immediate base class of T
set_types(T)	to define the properties of T
types(T)	to **get** the properties set of T
set_ext_type(T, P)	to define an extra property P for T for extending the properties set without intrusion
typeof(T, P)	to **get** the property P of T

and also

function	*meaning*
set_impl(T)	to define a default impl for the interface of T
set_inherits(A, E, i)	to define the i^{th} inheritance rule for E in the A hierarchy

The magic of typeof

setting a property P for a type T can be performed

- either within the bundles of types associated with T
  - ⇝ one should then use `set_types(T)`
- or via the non-intrusive extension process
  - ⇝ one should then use `set_ext_type(T, P)`

`typeof(T, P)` retrieves from any type T its property P

practically

- the property is defined either in the bundle `types(T)` or, as a stand-alone extension, by `type(T, P)`
- both structures `types(T)` and `type(T, P)` follows inheritance to provide virtual types
- the property should not be twice '`not_found`' nor '`undefined`'

Ready?

so let’s rock!

and that’s not so hard...

Hierarchy top class

A hierarchy has a top abstract class.

// first declare the class:

```
decl 'E image;
```

// before setting the types related to it:

```
'E set_types<current> {
  value = undefined;
  point = undefined;
};
```

// last define the class:

```
'E image<E> : any<E>, impl<current> {
  @value& operator[](const @point& p) = 0;
};
```

Hierarchy entry class

A SCOOP v2 hierarchy has an entry class.

// the entry point of the image hierarchy:

```
decl ’E image_entry;

’E image_entry<E> : inherits<image, E> {
};
```

the class “`inherits`”, provided in the equipment, allows for sub-classes to implicitly inherit from client abstractions

A concrete class

Then we can add a concrete class.

// first declare:

```
decl 'T image2d;
set_super<current> = image_entry<current>;
```

// then set types:

```
set_types<current> {
  value = T;
  point = point2d;
};
```

// last define:

```
'T image2d<T> : super<current> {
  @value& operator[](const @point& p) { ... }
  ...
};
```

Adding a sub-hierarchy

A first sub-hierarchy is defined (discriminant = grid dimension).

// start with the sub-hierarchy:

```
'E image_2d<E> : image<E>, impl<current> {
  @value& operator()(int row, int col) {
    return (*this)[@point(row, col)];
  }
};

//...
```

// and end with the corresponding inheritance rule:

```
'E set_inherits<image, E, 1> =
   if typeof(E, point) == point2d
   then image_2d<E>
   // elseif ...
   ;
```

Making a room for morphers

Let us introduce the abstract impl. class for image morphers.

// declare:

```
decl 'I 'E morpher;
set_super<current> = image_entry<current>;
```

// fetch the properties from I:

```
set_types<current> : types<I> { // for the packed ones
  delegated = I; // extra property
};
'P set_type<current, P > = type<I, P>; // for the stand-alone ones
```

// define:

```
'I 'E morpher<I,E> : super<current> {
  morpher(I& ima) : ima_(ima) {}
  @delegated impl_delegate() { return ima_; }
  I& ima_;
};
```

Adding a morpher

We can add an image morpher: the class for “image + mask”.

declare:

```
decl 'I 'M masked_;
set_super<current> = morpher<I, current>;
```

set a new type:

```
set_type<current, mask > = M; // extra property
```

define:

```
'I 'M masked_<I, M> : super<current> {
  masked_(I& ima, M& mask) : super(ima), mask_(mask) {}
  M& mask() { return mask_; }
  M mask_;
};
```

Generalization

The “mask” property becomes global (defined for all image types):

```
'I set_type< image<I>, mask > = no_type; // default value
```

and a second sub-hierarchy takes advantage of this new property:

```
'E masked_image<E> : image<E>, impl<current> {
  @mask& mask() = 0;
};

'E set_inherits<image, E, 2> =
   if typeof(E, mask) != no_type
   then masked_image<E>
   ;
```

Method default implementation

To handle morphers, one should be able to automatically delegate methods.

```
'E set_impl< image<E> > {
  // the abstract image class is thus equipped:
  @delegated& delegate() = 0;
  E& impl_delegate(); // no default impl
  // delegation is implemented for the image class interface:
  @value& impl_operator[](const @point& p) {
    return delegate().operator[](p);
  }
}
// we proceed likewise for each client abstract class:
'E set_impl< masked_image<E> > {
  @mask& impl_mask() { return delegate().mask(); }
}
```

Conclusion

- this document is nothing but an introduction to SCOOP v2
- a technical report with much more details will be published on our web site `http://olena.lrde.epita.fr`
- a perspective of our work is to provide a language
    - based on the concepts presented in those slides
    - dedicated to efficient object-oriented scientific programing

Thanks

I'd like to thank

- the SCOOP v1 team, which is at the origin of this work: Nicolas Burrus, Alexandre Duret-Lutz, David Lesage, and Raphaël Poss
- Roland Levillain for fruitful discussions
- Akim Demaille and all the people that are supporting the OLENA project
- and every contributors to OLENA

just mail me if you have some comments or questions:

`theo@lrde.epita.fr`