SCOOP: a static C++ object-oriented paradigm

Thomas Moulard

LRDE – EPITA Research and Development Laboratory

January 9, 2008
Outline

1. Introduction
2. Common C++ paradigms
3. SCOOP’s approach
4. Conclusion
Outline

1. Introduction
2. Common C++ paradigms
3. SCOOP’s approach
4. Conclusion
What is not SCOOP?

Not a library  SCOOP  is more than a tool as it changes how to engine a library.
Introduction

What is not SCOOP?

Not a library  SCOOP is more than a tool as it changes how to engine a library.

Not a framework  SCOOP can be integrated to existing projects.
Introduction

What is not SCOOP?

Not a library  SCOOP is more than a tool as it changes how to engine a library.

Not a framework  SCOOP can be integrated to existing projects.

Not a language extension  SCOOP uses “pure” C++ 2003 (no extra-preprocessing, nor extension).
Introduction

What is SCOOP?

SCOOP is a paradigm, i.e. a thought pattern [Wikipedia, 2008]. Designed during OLENA’s development in order to write generic libraries [Géraud, 2006].

Consequences

- Impact software design (class hierarchies).
- Introduce a new C++ “dialect”.
- Bring new features in current C++ through a different way of writing pieces of software.
Outline

1. Introduction
2. Common C++ paradigms
3. SCOOP’s approach
4. Conclusion
Object-Oriented programming with C++ 2003

Introduction

Common C++ paradigms

SCOOP’s approach

Conclusion

Figure: Object-Oriented programming

Features

Classes hierarchies Define abstract base classes and specialize them via sub-classing.

Dynamic typing Dynamic typing allows a variable of type $X$ to contain a value of type $Y$ where $Y <: X$ (read “$Y$ is a sub-type of $X$”).

Virtual methods Virtual methods allow implementation refinement.

Image

- fill(Color)

Image2d

- fill(Color)

Image3d

- fill(Color)

Specialized versions of the fill algorithm.
Object-Oriented programming with C++ 2003

Common problems

**Image**
- getPixel(Point)

**Image2d**
- getPixel(Point2d)

**Image3d**
- getPixel(Point3d)

**Argument covariance**
Argument covariance is not present in C++. For example, if `fill` takes an argument of type `Point`, a specialized version of `fill` cannot take an argument of type `Point2d` (where `Point2d <: Point`).

**Figure:** Object-Oriented programming and covariance arguments.
Common problems

Figure: Object-Oriented programming and covariance arguments.

Arguments are not covariant in C++: if `fill` takes an argument of type `Point`, a specialized version of `fill` can not take an argument of type `Point2d` (where `Point2d <: Point`).
Object-Oriented programming with C++ 2003

Common problems

Figure: Object-Oriented programming and multi-methods.

**Multi-methods**

Virtual methods realize a dispatch depending on the *first* argument of the method (*this*).
With multi-methods, the dispatch depends on every argument.

**How to write a generic copy algorithm?**

Impossible in “classic” C++ without run-time overhead.
Generic programming with C++ 2003

Features

Meta-class use  Define meta-class (i.e. templated classes) in order to generate a class which suits specific needs.

Static typing  No class hierarchies (no sub-typing): type checking done at compile-time.

Easy genericity  No need to write plenty of classes, just templating by another type.

Figure: Generic programming.

No inheritance between meta-class and classes.

Vector
- push_back(T)
- sort()

Vector<int>
- push_back(int)
- sort()

Vector<float>
- push_back(float)
- sort()
Generic programming with C++ 2003

Lack of constraints on parameters

Extremely easy to make mistakes as requirements on types can not be expressed in the source code.

It is impossible to restrict the parameters of the meta-class to the only classes which have the features you need (i.e. the attributes, methods... of the parameter used in the meta-class).

Figure: The “sort” algorithm and generic programming.

Meta-class

Vector

push_back(T)

sort()

Requires that operator< is defined in T (documentation only!)

Classes

Vector<Point1d>

push_back(Point1d)

sort()

OK

Vector<Point2d>

push_back(Point2d)

sort()

Error

Vector

T

Thomas Moulard

SCOOP: a static C++ object-oriented paradigm

12 / 27
Generic programming with C++-0X

C++-0X

C++-0X is the next C++ standard. It should be finished in one or two years. One of the major changes is the introduction of Concepts.

Concepts

The concept proposition[D. Gregor, 2006] solves the problem of the lack of constraints. A structural matching is made in order to decide if a type is valid for a specific template parameter.

Figure: C++-0X concepts.
Comparing paradigms

<table>
<thead>
<tr>
<th>Feature</th>
<th>OOP</th>
<th>GP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>Easy</td>
<td>Hard</td>
</tr>
<tr>
<td>Genericity</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Speed</td>
<td>Slow</td>
<td>Fast</td>
</tr>
</tbody>
</table>

**Table:** Paradigms characteristics.

**Main problem**

How to obtain a paradigm which mixes the advantages of both *Oriented-Object programming* and *Generic programming*?
### Comparing paradigms

<table>
<thead>
<tr>
<th>Feature</th>
<th>OOP</th>
<th>GP</th>
<th>SCOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>Easy</td>
<td>Hard</td>
<td>Medium</td>
</tr>
<tr>
<td>Genericity</td>
<td>Medium</td>
<td>Medium</td>
<td>Easy</td>
</tr>
<tr>
<td>Speed</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
</tr>
</tbody>
</table>

**Table:** Paradigms characteristics.

#### SCOOP’s features
- Static virtual methods
- Covariant arguments
- Static multi-methods
- *Functions from type to type (Morphers).*
**SCOOP core mechanism**

**Curiously recurring template pattern [Coplien, 1995]**

The dynamic type is a class parameter. Abstract, non-final classes are parameterized by the “exact” (dynamic) type, a final (concrete) class is not parameterized (its exact type is itself).

**Benefits and drawbacks**

- The dynamic type is known at **compile time**.
- Easy to dispatch depending on the dynamic type (for virtual methods, argument covariance and multi-methods).
- Less flexible than Oriented Object programming (impossible to change the dynamic type depending on a non-static condition).

**Figure:** A simple SCOOP hierarchy.
**The PIMPL design pattern [Sutter, 2000]**

---

**Description**

A public method is divided into two methods:
- A public method
- Another function which really implements the function.

**Interest**

- Separate interface from implementation.
- Avoid changing the class interface directly.

---

**Figure:** PIMPL design pattern.
Static virtual methods

**Principle**

The dispatch is done manually by the public method.

**Interest**

- No “virtual” keyword: no run-time overhead.
- Easy to implement and to understand.
- Use language overloading mechanism.

**Figure**: Virtual methods in SCOOP.
Argument covariance

**Figure**: Argument covariance in SCOOP.

**Principle**
Dispatch done manually by the public method: the argument's dynamic type can be retrieved through SCOOP's mechanisms.

**Interest**
- No run-time overhead.
- Easy to implement: uses C++ classic overloading.
Multi-methods

**Principle**

Dispatch done manually by the interface function: the arguments’ dynamic types can be retrieved through SCOOP’s mechanisms.

**Interest**

- No run-time overhead.
- Easy to implement: uses C++ classic overloading.

Figure: Multi-methods in SCOOP.

```
Image copy (Image i1, Image i2)
return exact ().impl_copy (i1.exact (), i2.exact ());
```

```
Image2d impl_copy (Image2d i1, Image2d i2)
```

```
Image3d impl_copy (Image3d i1, Image3d i2)
```

```
Image3d impl_copy (Image2d i1, Image3d i2)
```

```
Image2d impl_copy (Image3d i1, Image2d i2)
```

```
Image2d impl_copy (Image3d i1, Image2d i2)
```

```
Image2d impl_copy (Image3d i1, Image2d i2)
```
Virtual types

**Figure:** Virtual types in SCOOP.

**Principle:**
Traits are used to contain types associated to classes. The retrieve algorithm of SCOOP allows its type to be refined in subclasses.
Property based inheritance [Thierry Géraud, 2007]

**Figure:** Property based inheritance.

- **Principle**
  - Three parts:
    - Concepts.
    - Meta-bridge (internal mechanism).
    - Implementation classes.
Morphers

**Delegatee type**

Delegatee is a special virtual type, it delegates the virtual type definition to another class.

**Principle**

Morphers provides type to type functions through the delegatee type. It satisfies concepts of the destination type, takes an instance of the source type as parameter and bridges the two types.

---

**Figure:** Morphers in SCOOP.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Common C++ paradigms</td>
</tr>
<tr>
<td>3</td>
<td>SCOOP’s approach</td>
</tr>
<tr>
<td>4</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>
**Conclusion**

**SCOOP’s road-map**

- Enhance SCOOP’s libraries (Metalic and Static) to make them more user-friendly.
- Prove that the paradigm can scale through the development of OLENA 1.0.
- Enhance code readability through a new mini-language (SCOOL, successor of Metagene [Maes, 2004]) or through a C++ language extension.
Bibliography I


