Design Patterns for Generic Programming

Thierry Géraud and Alexandre Duret-Lutz

Thierry.Geraud@epita.fr

EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France
Phone +33 1 44 08 01 77 – Fax +33 1 44 08 01 99
http://www.lrde.epita.fr

Abstract. Generic programming is a recent paradigm, in which most classes and procedures are parameterized, thus leading to the construction of general and efficient software components. In this paper, we demonstrate that some design patterns from Gamma *et al.* can be translated into this paradigm. Although these patterns highly rely on dynamic binding, we show that, in the context of generic programming, the resolution of method calls in these patterns can be performed at compile-time, which is of prime importance for numerical computing. To this end, we make intensive use of parametric polymorphism.

1 Introduction

Thanks to parameterization, generic programming allows to abstractly represent data structures *and* to efficiently implement algorithms. Thus it is a new way of programming and assembling components [6]. The main features of this paradigm are:

- The generic expression of an algorithm only needs few hypothesis on the data it uses.
- A specialized version of an algorithm, e.g. dedicated to a particular data type, can always override the generic implementation. Nevertheless, there are no syntactic differences between using the generic or a specialized form, and no loss in efficiency.
- The translation of an algorithm dedicated to a data type into a generic algorithm does not incur a significant overhead at run-time.

For several years, generic programming was considered an ideal paradigm for libraries of scientific numerical components [5]. Some libraries are available on the Internet [10], for various domains: containers, graphs, linear algebra, computational geometry, differential equations, neural networks, visualization, image processing, etc.

Several generic programming idioms have already been discovered and many are listed in [12]. However, as far as we know, very few design patterns dedicated to this paradigm have been brought to the fore (let us mention the Required Interface pattern [7]). In this paper, we show that some design patterns from Gamma *et al.* [4] can be translated into this paradigm. These patterns, which are some of the most important for modularity and reusability, have a strong dynamic behavior that seems antagonist with generic programming. Nevertheless,

we show that method calls in these patterns can be solved statically, i.e. at compile-time; basically, we avoid dynamic binding by replacing it with parametric polymorphism.

The section 2 explains the limits of the "classical" object-oriented paradigm (including genericity) in the field of numerical computing, and presents the generic programming paradigm. Then, the section 3 presents three patterns: Generic Iterator, Generic Template Method and Generic Decorator, and justifies their design. We conclude and discuss the perspectives of our work in the section 4.

Implementations of the designs given in this paper and of some other patterns are available at `http://www.lrde.epita.fr/download/`.

2 Beyond Genericity: Generic Programming

Before considering design issues for generic programming, we first present this recent paradigm. It is illustrated with implementation samples written in C++, which appears to be the consensual language for object-oriented numerical computing.

2.1 Genericity and its Limits

If genericity can be considered as part of the object paradigm [8], it is mostly used to avoid macros and to build utility classes, where a parameter usually represents a data type. Unfortunately, the "classical" use of genericity is not well suited to numerical computing as demonstrated by the following examples.

First Problem: Efficiency. Let us consider a very simple algorithm: the addition of a constant to the elements of an aggregate. We aim at having a single implementation of the algorithm, which means that this procedure should accept various aggregate types and data types (the types of the elements). We can have a general implementation by the means of the Iterator pattern, parameterized by the data type:

```
template< typename T >
void add( Aggregate<T>& input, T value )
{
  Iterator<T>& iter = input.CreateIterator();
  for ( iter.First(); ! iter.IsDone(); iter.Next() )
    iter.CurrentItem() += value;
}
```

In this algorithm, only abstract classes appear (`Aggregate<T>` and `Iterator<T>`) and each iteration involves polymorphic calls. For numerical computing applications, the penalty of dynamic binding results in an unacceptable overhead at run-time (it can be twice to ten times slower than dedicated handwritten code).

Thus, genericity helps to write the algorithm only once, i.e., for any input type, but the use of the classical object-oriented paradigm prevents us from obtaining efficient computations.

Second Problem: Genericity is not Enough. A typical implementation of the computation of the mean value of the elements of an aggregate is:

```
template< typename T >
T mean( const Aggregate<T>& input )
{
  /* a type */ sum = /* an initial value */;
  Iterator<T>& iter = input.CreateIterator();
  for ( iter.First(); ! iter.IsDone(); iter.Next() )
    sum += iter.CurrentItem();
  return sum / input.size();
}
```

The variable `sum` should have a type that allows to store larger values than the ones of type `T`; the `sum` type depends on the unknown type-parameter `T`. Moreover, the initial value of the variable `sum` is not always the scalar `0` since `T` can be non-scalar.

At this stage, using classical genericity does not provide solutions to implement such a simple algorithm. As a consequence, most of the available scientific libraries leave genericity aside and are restricted to few data types in order to limit the tedious "copy, paste, and modify" process.

2.2 Towards Generic Programming

Generic programming addresses the problems pointed out in the previous section. Two key ideas in this paradigm are:

- the procedures should be parameterized by the types of the input, even if the input itself is parameterized ;
- the types of the algorithmic tools should be deduced from the input types (to this end, type aliases are declared within classes).

Let us consider again the latter example. The parameter is now the type of the procedure argument `input`, say `A`. The algorithmic tools are an iterator and a summation value. The iterator type can also be deduced from `A` thanks to the alias `iterator_t`; the type of the aggregate elements, say `T`, can be deduced from `A` thanks to the alias `data_t`; and the summation type can be deduced from `T` thanks to the alias `sum_t`. Last, the sum initial value is the constant `zero`, defined as a class attribute in `T`. The resulting code[1] is:

```
class int_u8
{
  typedef int_u16 sum_t;
  static const int_u8 zero;
  //...
};
```

```
template< typename T >
class buffer
{
 public:
  typedef T data_t;
  typedef buffer_iterator<T> iterator_t;
  //...
};
```

[1]The keyword `typename` used in the procedure `mean` (printed on next page) before type deductions is required by the language C++ to fix ambiguities.

```
template< typename A >
typename A::data_t mean( const A& input )
{
  typedef  typename A::data_t  T;
  typename T::sum_t sum = T::zero;
  typename A::iterator_t
                   iter = input.CreateIterator();
  for ( iter.First(); ! iter.IsDone(); iter.Next() )
    sum += iter.CurrentItem();
  return sum / input.size();
}
```

```
int main()
{
  buffer<int_u8> buf;
  //...
  int_u8 value = mean( buf );
}
```

When a procedure `mean` is instantiated at compile-time for a given type (`A` is set to `buffer<int_u8>`, in the code above), no type within the procedure is abstract, and thus no method call is polymorphic. Moreover, each call can be inlined by the compiler. The executable has about the same performances as dedicated code.

Generic programming simultaneously addresses two issues: writing a single procedure per algorithm, and having efficient numerical procedures. Moreover, generic programming allows to express algorithms that genericity cannot.

2.3 Generic Programming Rules

We propose two rules that help to better define generic programming.

1. Operation polymorphism (keyword `virtual` in C++) is excluded because dynamic binding is too expensive. In other words, abstract methods are forbidden.

   As a consequence, inheritance is only used to factor method implementation and to declare attributes that can be shared by several subclasses.

2. Inclusion polymorphism is excluded. In other words, the type of a variable (static type) is exactly that of the instance it holds (dynamic type).

   As a consequence, each container manages exclusively objects with the same dynamic type. For instance, we can handle lists of cars or lists of trucks but not lists of vehicles.

These rules may seem drastic; however, C++ is a multi-paradigms language and the use of generic programming can be limited to some critical parts of code, dedicated to intensive calculi (the other pieces of the software can still have a classical object-oriented design).

3 Generic Design Patterns

The way we present design patterns differs from current fashions. Since these patterns have already been described in the book of Gamma *et al.* [4], we do not repeat the elements that can be found in this book.

3.1 Generic Iterator

Intent

Exactly the same as the original pattern.

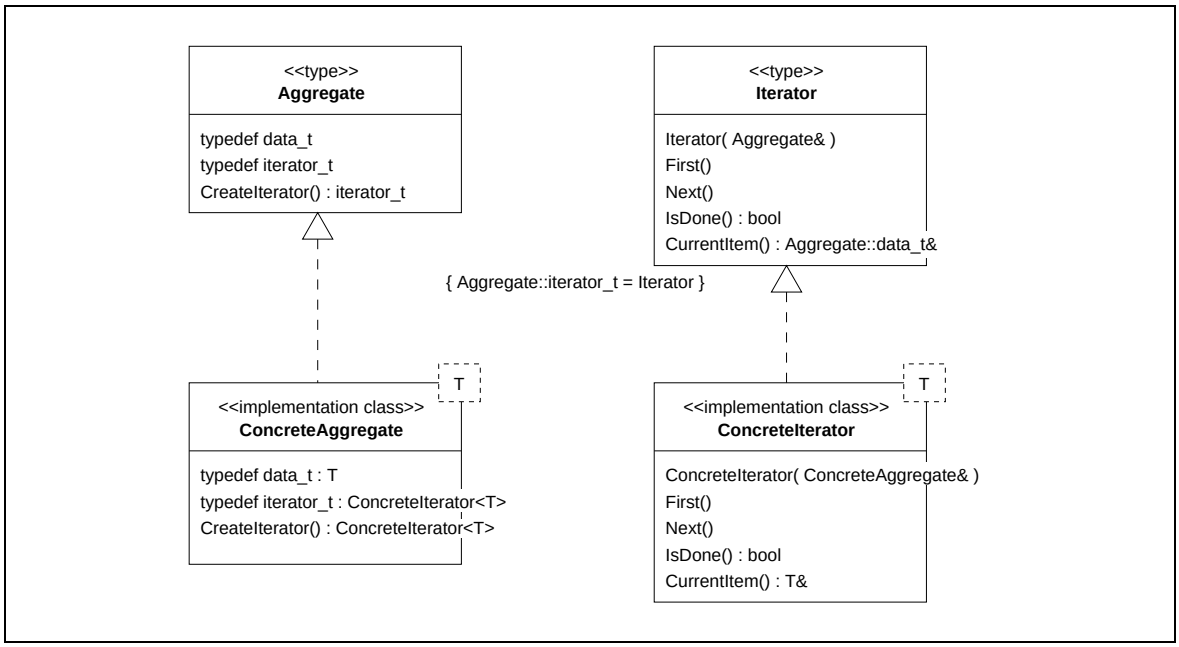

Figure 1: GENERIC ITERATOR pattern.
We used a non-standard extension of UML to represent type aliases in classes.

Motivation

In scientific computing, data are often aggregates and algorithms usually accept various aggregate types as input and browse the aggregate elements. The notion of iterator is thus a very common tool and a major requirement is that iterations are expected to be efficient (this is an extra requirement compared to the original pattern).

Structure

It is given in figure 1.

Participants

In generic programming, according to Austern [1], the description of a set of requirements on a type is a *concept*, and a type which satisfies these requirements is a *model* of this concept. For this pattern, we defined two concepts: *aggregate* and *iterator*, and two concrete classes, models of these concepts.

Consequences

Contrary to the original pattern, the generic design does not use inheritance. No operation is polymorphic.

Implementation

Although a concept is denoted in UML by the stereotype <<type>>, a concept does not lead to a C++ type; a concept only exists in program documentation.

For the user, a type-parameter represents a model of aggregate and the corresponding model of iterator can then be deduced statically, i.e. at compile-time. Moreover, each call of a method of the iterator can be inlined and iterations are very efficient.

Sample Code

A sample code is given in the section 2.2.

Known Uses

Most generic libraries use the generic iterator pattern, eg. STL, the C++ *Standard Template Library* [11]. In fact, generic programming appeared with the adoption of the STL by the C++ standardization committee and was made possible with the addition to this language of new generic capabilities [9].

3.2 Generic Template Method

Intent

Exactly the same as the original pattern.

Motivation

In the section 2.3, we said that inheritance is used in generic programming to factor methods. Here, we want a superior class to define an operation some parts of which (primitive operations) are defined only in inferior classes, in addition we want method calls to be solved at compile-time. It concerns calls of the primitive operations as well as calls of the template method.

Structure

It is given in figure 2.

Participants

In the object-oriented paradigm, the resolution of a polymorphic operation call on a target object consists in finding a method that implements the operation, while searching bottom-up in the class hierarchy from the object dynamic type. In generic programming, let us consider a leaf class; if its superior classes are parameterized by the type of this class, they always know the dynamic type of the object.

The parametric class `AbstractClass` defines two operations: `PrimitiveOperation1()` and `PrimitiveOperation2()`. Calling one of these operations leads to transtyping the target object to its dynamic type, thanks to the method `Self()`, inherited from the parametric class `SuperiorOf`. The methods that are executed are the implementations of these operations, respectively `PrimitiveOperation1_impl()` and `PrimitiveOperation2_impl()`. These implementations are searched for from the dynamic object type.

When the programmer later defines the class `ConcreteClass` with the primitive operation implementations, the method `TemplateMethod()` is inherited and a call of this method leads to the execution of the proper implementations.

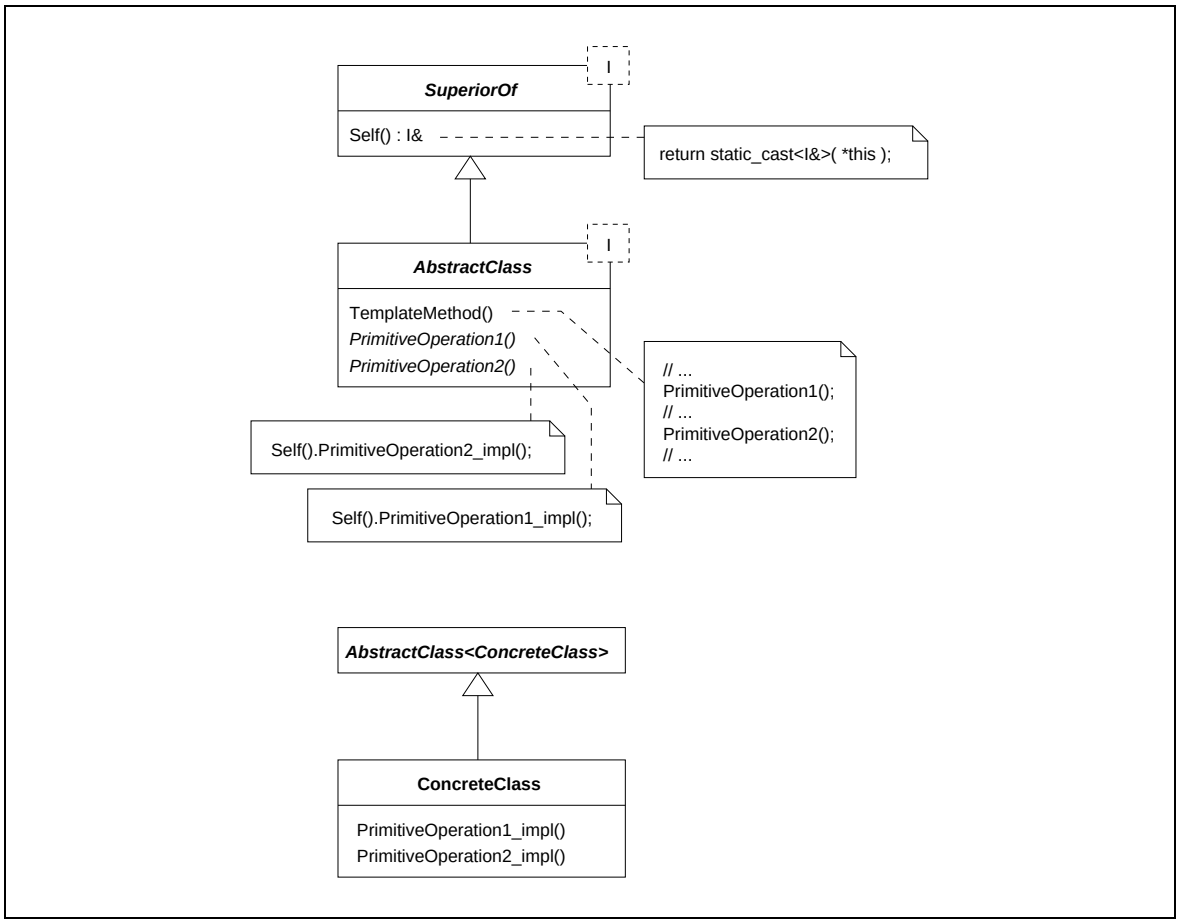

Figure 2: Generic Template Method pattern.

Consequences

In generic programming, operation polymorphism can be simulated by "parametric polymorphism through inheritance" and then be solved statically. The cost of dynamic binding is avoided; moreover, the compiler is able to inline all the pieces of code, including the template method itself. Hence, this design does not penalize efficiency.

Implementation

`SuperiorOf` and `AbstractClass` can behave like abstract classes; to this end, their constructors are protected. The methods `PrimitiveOperation1()` and `PrimitiveOperation2()` do not contain an operation implementation but a call of an implementation; they can be considered as abstract methods. Please note that they can also be called by the client (the fact that these methods are polymorphic-like is hidden because the call `Self` is encapsulated).

Known Uses

This pattern relies on an idiom (the *Curiously Recurring Template*) given in [3] and based on [2]. In this idiom, an binary operator, for instance `+`, is defined in a superior class from the corresponding unary operator, here `+=`, defined in an inferior class.

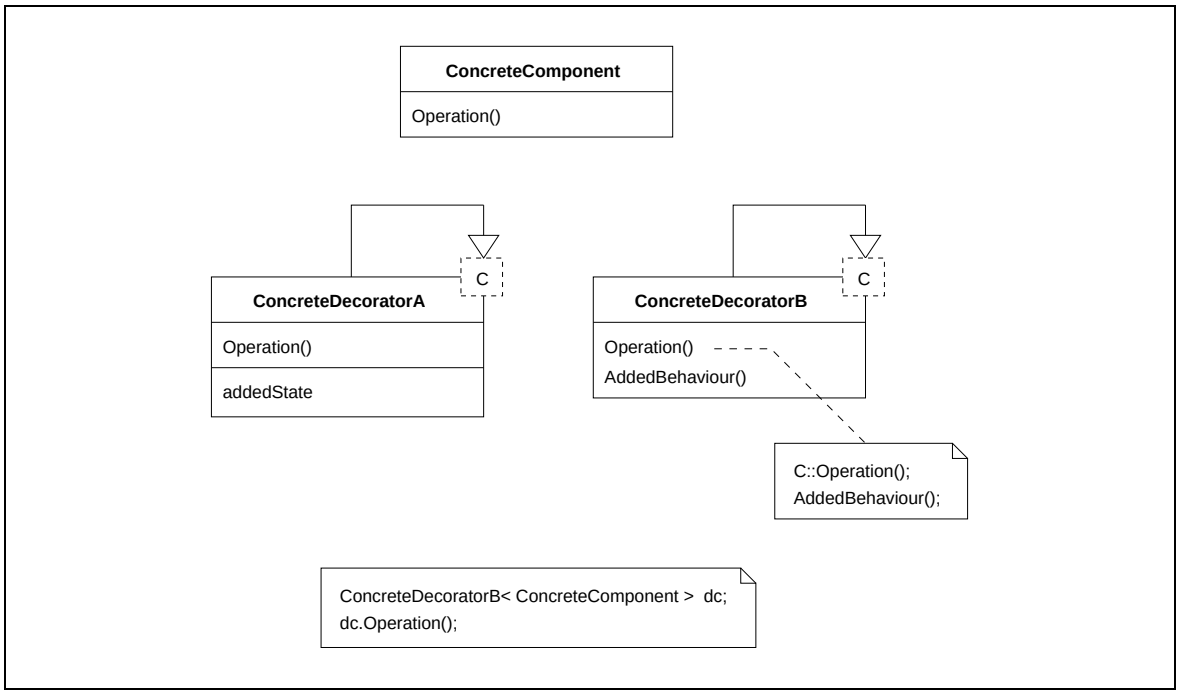

Figure 3: Generic Decorator pattern.

3.3 Generic Decorator

Intent

Exactly the same as the original pattern.

Structure

It is given in figure 3. We used a special idiom: having a parametric class that derives from one of its parameters. As far as we know, this is a contribution.

Participants

A class `ConcreteComponent` which can be decorated, offers an operation `Operation()`. Two parametric decorators, `ConcreteDecoratorA` and `ConcreteDecoratorB`, whose parameter is the decorated type, override this operation. It is in effect a substitution since the decorators are inheriting from their parameter.

Consequences

This pattern has two advantages over Gamma's. First, any method that is not modified by the decorator is automatically inherited. Not only does this frees the decorator from having to define these operations, but in addition, any specific method of a decorated is in its decorator. Second, decoration can be applied to a set of classes that are not related via inheritance. Therefore, a decorator becomes truly generic.

On the other hand, in generic programming, we lose the capability of dynamically adding a decoration to an object.

Sample Code

Decorating an iterator of STL is useful when a container holds structured data, and one wants to perform operations only on a field of these data. In order to access this field, the decorator only has to redefine the data access operator `operator*()` of the iterator.

```
typedef std::list< RGB<int> > A;
A input;
// ...
FieldAccess< A::iterator, Get_red > i;
for ( i = input.begin(); i != input.end(); ++i )
{
  *i = 0;
}
```

The example given above uses a decorator `FieldAccess`. Its parameters correspond to the type of the decorated iterator, and to a function object [1] which specifies the field to be accessed. A loop sets to `0` the red field of a list of red-green-blue colors. Without decoration, the iterator would have set all the colors to `{0,0,0}`.

The decorator uses methods inherited from the decorated iterator, such as the operators of assignment, comparison, and pre-increment.

4 Conclusion and Perspectives

Generic programming is a new paradigm which will soon be applied to other domains than scientific numerical computing. Based on object programming, this paradigm allows to build and assemble efficient reusable components [6]. In addition, the generic components are compatibles with traditional components via generative programming [13]. This contrasts with the previous image of generic programming, seen as incompatible with dynamic approaches because of its static orientation.

In this paper, we have highlighted the limits of "classical" genericity and we have shown that generic programming provides us with means to write efficient algorithms that accept various input. As an original contribution, we have demonstrated what has become possible: design patterns can be used to better modularize and reuse parts or all of generic softwares. As an example, we have presented the generic versions of three fundamental patterns from Gamma *et al.* [4]: the Generic Iterator, the Generic Template Method and the Generic Decorator. Due to the lack of room, we do not give four more GoF patterns that we also have translated into generic programming: the Generic Bridge, the Generic Visitor, the Generic Abstract Factory, and the Generic Chain of Responsibility.

We believe that it is time to explore the benefits that generic programming can derive from "classical" object-oriented software design.

Acknowledgments. The authors would like to thank Akim Demaille and Philippe Laroque for their fruitful comments on this paper.

References

[1] Matthew H. Austern. *Generic programming and the STL – Using and extending the C++ Standard Template Library.* Professional Computing Series. Addison-Wesley, 1999.

[2] John Barton and Lee Nackman. *Scientific and engineering C++.* Addison-Wesley, 1994.

[3] James Coplien. Curiously recurring template pattern. In Stanly B. Lippman, editor, *C++ Gems.* Cambridge University Press & Sigs Books, 1996. `http://people.we.mediaone.net/stanlipp/gems.html`

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns – Elements of reusable object-oriented software.* Professional Computing Series. Addison Wesley, 1995.

[5] Scott Haney and James Crotinger. How templates enable high-performance scientific computing in C++. *IEEE Computing in Science and Engineering*, 1(4), 1999. `http://www.acl.lanl.gov/pooma/papers.html`

[6] Mehdi Jazayeri. Component programming – a fresh look at software components. In *Procedings of the 5th European Software Engineering Conference (ESEC'95)*, pages 457–478, September 1995.

[7] Ullrich Köthe. Requested interface. In *In Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97)*, Munich, Germany, 1997. `http://www.riehle.org/events/europlop-1997/#ww1p2`

[8] Bertrand Meyer. *Object-oriented software construction.* Prentice Hall, 2 edition, 1997.

[9] Nathan C. Myers. Gnarly new C++ language features, 1997. `http://www.cantrip.org/gnarly.html`

[10] The object-oriented numerics page. `http://oonumerics.org/oon`

[11] Alex Stepanov and Meng Lee. *The Standard Template Library.* Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, February 1995.

[12] Todd L. Veldhuizen. Techniques for scientific C++, August 1999. `http://extreme.indiana.edu/~tveldhui/papers/techniques/`

[13] Todd L. Veldhuizen and Dennis Gannon. Active libraries – Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98).* SIAM Press, October 1998. `http://extreme.indiana.edu/~tveldhui/papers/oo98.html`