Design Patterns for Generic Programming in C++

Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille
EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France
`{Alexandre.Duret-Lutz,Thierry.Geraud,Akim.Demaille}@lrde.epita.fr`
`http://www.lrde.epita.fr/`

Abstract

Generic programming is a paradigm whose wide adoption by the C++ community is quite recent. In this scheme most classes and procedures are parameterized, leading to the construction of general and efficient software components. In this paper, we show how some design patterns from Gamma *et al.* can be adapted to this paradigm. Although these patterns rely highly on dynamic binding, we show that, by intensive use of parametric polymorphism, the method calls in these patterns can be resolved at compile-time. In intensive computations, the generic patterns bring a significant speed-up compared to their classical peers.

1 Introduction

This work has its origin in the development of Olena [11], our image processing library. When designing a library, one wants to implement algorithms that work on a wide variety of types without having to write a procedure for each concrete type. In short, one algorithm should be *generic* enough to map to a single procedure. In object-oriented programming this is achieved using *abstract types*. *Design Patterns*, which are design structures that have often proved to be useful in scientific computing, rely even more on abstract types and inclusion polymorphism[1].

However, when it comes to numerical computing, object-oriented designs can lead to a *huge performance loss*, especially as there may be a high number of virtual functions calls [7] required to perform operations over an abstraction. Yet, rejecting design patterns for the sake of efficiency seems radical.

In this paper, we show that some design patterns from Gamma *et al.* [10] can be adapted to generic programming. To this aim, virtual functions calls are avoided by replacing inclusion polymorphism by parametric polymorphism.

This paper presents patterns in C++, but, although they won't map directly to other languages because "genericity" differs from language to language, our work does not apply only to C++: our main focus is to devise flexible designs in contexts where efficiency is critical. In addition, C++ being a multi-paradigm programming language [28], the techniques described here can be limited to critical parts of the code dedicated to intensive computation.

In section 2 we introduce generic programming and present its advantages over classical object-oriented programming. Then, section 3 presents and discusses the design of the following patterns: GENERIC BRIDGE, GENERIC ITERATOR, GENERIC ABSTRACT FACTORY, GENERIC TEMPLATE METHOD, GENERIC DECORATOR, and GENERIC VISITOR. We conclude and consider the perspectives of our work in section 4.

2 Generic programming

By "generic programming" we refer to a use of parameterization which goes beyond simple genericity on data types. Generic programming is an abstract and efficient way of designing and assembling components [15] and interfacing them with algorithms.

[1] *Inclusion polymorphism* corresponds to virtual member functions in C++, deferred functions in Eiffel, and primitive functions in Ada.

Generic programming is an attractive paradigm for scientific numerical components [12] and numerous libraries are available on the Internet [22] for various domains: containers, graphs, linear algebra, computational geometry, differential equations, neural networks, visualization, image processing, etc.

The most famous generic library is probably the *Standard Template Library* [26]. In fact, generic programming appeared with the adoption of STL by the C++ standardization committee and was made possible with the addition of new generic capabilities to this language [27, 21].

Several generic programming idioms have already been discovered and many are listed in [30]. Most generic libraries use the GENERIC ITERATOR that we describe in 3.2. In POOMA [12] — a scientific framework for multi-dimensional arrays, fields, particles, and transforms — the GENERIC ENVELOPE-LETTER pattern appears. In the REQUESTED INTERFACE pattern [16], a GENERIC BRIDGE is introduced to handle efficiently an adaptation layer which mediates between the interfaces of the servers and of the clients.

2.1 Efficiency

The way abstractions are handled in the object-oriented programming paradigm ruins the performances, especially when the overhead implied by the abstract interface used to access the data is significant in comparison with the time needed to process the data.

For example, in an image processing library in which algorithms can work on many kinds of aggregates (two or three dimensional images, graphs, etc.), a procedure that adds a constant to an aggregate may be written using the object-oriented programming paradigm as follows.

```
template< class T >
void add (aggregate<T>& input, T value)
{
  iterator<T>& iter = input.create_iterator ();
  for (iter.first(); !iter.is_done(); iter.next())
    iter.current_item () += value;
}
```

Here, `aggregate<T>` and `iterator<T>` are abstract classes to support the numerous aggregates available: parameterization is used to achieve genericity on pixel types, and object-oriented abstractions are used to get genericity on the image structure.

	dedicated C	classical C++	generic C++
`add`	10.7s	37.7s	12.4s
`mean`	47.3s	225.8s	57.5s

Table 1: Timing of algorithms written in different paradigms. (The code was compiled with `gcc 2.95.2` and timed on an AMD K6-2 380MHz machine running GNU/Linux.)

As a consequence, for each iteration the direct call to `T::operator+=()` is drowned in the virtual calls to `current_item()`, `next()` and `is_done()`, leading to poor performances.

Table 1 compares classical object-oriented programming and generic programming and shows a speed-up factor of 3 to 4. The `add` test consists in the addition of a constant value to each element of an aggregate. The `mean` test replaces each element of an aggregate by the mean of its four neighbors. The durations correspond to 200 calls to these tests on a two dimensional image of 1024×1024 integers. "Dedicated C" corresponds to handwritten C specifically tuned for 2D images of integers, so the difference with classical C++ is what people call *the abstraction penalty*. While this is not a textbook case —we do have such algorithms in Olena— it is true that usually the impact of object-oriented abstraction is insignificant. High speed-ups are obtained from generic programming compared to object-oriented programming when data processing is cheap relatively to data access. For example for simple list iteration or matrix multiplication.

The generic programming writing of this algorithm, using a GENERIC ITERATOR, will be given in section 3.2.

2.2 Generic programming from the language point of view

Generic programming relies on the use of several programming language features, some of which being described below.

Genericity is the main way of generalizing object-oriented code. Not all languages support both generic classes and generic procedures (e.g., Eiffel features only generic classes).

Nested type names refers to the ability to look up a type as member of a class and allow to link related types (such as `image2d` and `iterator2d`) together.

Constrained genericity is a way to restrict the possible values of formal parameters using signatures (e.g., when using ML functors [19]) or constraining a type to be a subclass of another (as in Eiffel or Ada 95). C++ does not provide specific language features to support constrained genericity, but subclass constraints [29, 23] or feature requirements [18, 25] can be expressed using other available language facilities [27].

Generic specialization allows the specialization of an algorithm (e.g., dedicated to a particular data type) overriding the generic implementation.

Not all languages support these features, this explains why the patterns we present in C++ won't apply directly to other languages.

2.3 Generic programming guidelines

From our experience in building Olena, which is entirely carried out by generic programming, we derived the following guidelines. These rules may seem drastic, but their appliance can be limited to critical parts of the code dedicated to intensive computation.

Guidelines for generic classes:

- Avoid inclusion polymorphism.
  In other words, the type of a variable (static type, known at compile-time) is exactly that of the instance it holds (dynamic type, known at run-time). The main requirement of generic programming is that *the concrete type of every object is known at compile-time*.
- Avoid operation polymorphism.
  Abstract methods are forbidden: dynamic binding is too expensive. Simulate operation polymorphism with either: (i) parametric classes thanks to the *Curiously Recurring Template* idiom (see section 3.4), or (ii) parametric methods, which lead to a form of *ad-hoc* polymorphism (overloading).
- Use inheritance only to factor methods and to declare attributes shared by several subclasses.

Guidelines for procedures which use generic patterns:

- Parameterize the procedures by the types of their inputs, even if the input itself is parameterized.
- Parameterize the procedures by the types of the components used (unless they can be obtained by a nested type lookup in another parameter-type).

3 Generic Design Patterns

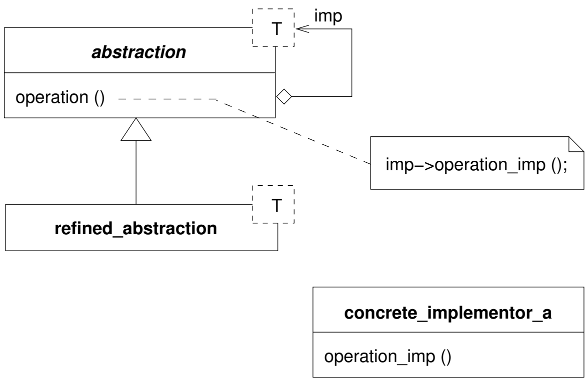
Our generic design patterns exposition is Gamma *et al.*'s description of the original, abstract version of the patterns [10]. We do not repeat the elements that can be found in this book.

3.1 Generic Bridge

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Structure



Participants

An `abstraction` class is parameterized by the `Implementation` used. Any (low-level) operation on the abstraction is delegated to the implementation instance.

Consequences

Because the implementation is statically bound to the abstraction, you can't switch implementation at run-time. This kind of restriction is common to generic programming: configurations must be known at compile-time.

Known Uses

This pattern is really straightforward and broadly used in generic libraries. For example the `allocator` parameter in STL containers is an instance of GENERIC BRIDGE.

The POOMA team [5] use the term *engine* to name implementation classes that defines the way matrices are stored in memory. This is also a GENERIC BRIDGE.

The Ada 95 rational [14, section 12.6] gives an example of GENERIC BRIDGE: a generic empty package (also called signature) is used to allow multiple implementation of an abstraction (here, a *mapping*).

As in the case of the original patterns, the structure of this pattern is the same as the GENERIC STRATEGY pattern. These patterns share the same implementation.

3.2 Generic Iterator

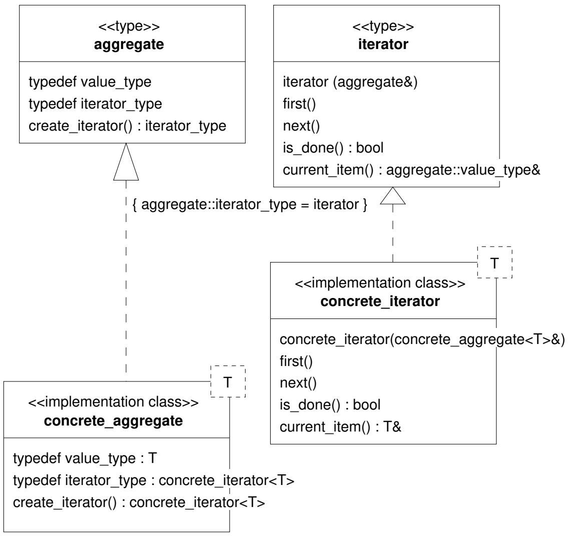
Intent

To provide an *efficient* way to access the elements of an aggregate without exposing its underlying representation.

Motivation

In numeric computing, data are often aggregates and algorithms usually need to work on several types of aggregate. Since there should be only one implementation of each algorithm, procedures must accept aggregates of various types as input and be able to browse their elements in some unified way; *iterators* are thus a very common tool. As an extra requirement compared to the original pattern, iterations must be efficient.

Structure



We use `typedef` as a non-standard extension of UML [24] to represent type aliases in classes.

Participants

The term *concept* was coined by M. H. Austern [1], to name a set of requirements on a type in STL. A type which satisfies these requirements is a *model* of this concept. The notion of concept replaces the classical object-oriented notion of abstract class.

For this pattern, two concepts are defined: *aggregate* and *iterator*, and two concrete classes model these concepts.

Consequences

Since no operation is polymorphic, iterating over an aggregate is more efficient while still being generic. Moreover, the compiler can now perform additional optimizations such as inlining, loop unrolling and instruction scheduling, that virtual function calls hindered.

Efficiency is a serious advantage. However we lose the dynamic behavior of the original pattern. For example we cannot iterate over a tree whose cells do not have the same type[2].

[2] A link between an abstract aggregate and the corresponding generic procedures can be achieved using lazy compilation and dynamic loading of generic code [8].

Implementation

Although a concept is denoted in UML by the stereotype <<type>>, in C++ it does not lead to a type: a concept only exists in the documentation. Indeed the fact that concepts have no mapping in the C++ syntax makes early detection of programming errors difficult. Several tricks have been proposed to address this issue by explicitly checking that the arguments of an algorithm are models of the expected concepts [18, 25]. In Ada 95, concept requirements (types, functions, procedures) can be captured by the formal parameters of an empty generic package (the *signature* idiom) [9].

For the user, a type-parameter (such as `Aggregate_Model` in the sample code) represents a model of *aggregate* and the corresponding model of *iterator* can then be deduced statically.

Sample Code

```
template< class T >
class buffer
{
  public:
    typedef T data_type;
    typedef buffer_iterator<T> iterator_type;
  // ...
};
```

```
template< class Aggregate_Model >
void add(Aggregate_Model& input,
         typename Aggregate_Model::data_type value)
{
 typename Aggregate_Model::iterator_type&
  iter = input.create_iterator ();

 for (iter.first(); !iter.is_done(); iter.next())
  iter.current_item () += value;
}
```

Known Uses

Most generic libraries, such as STL, use the GENERIC ITERATOR.

Variations

We translated the Gamma *et al.* version, with methods `first()`, `is_done()`, and `next()` in the iterator class. STL uses another approach where pointers should also be *models* of iterators: as a consequence, iterators cannot have methods and most of their operators will rely on methods of the container's class. This makes implementation of multiple schemes of iteration difficult: for example compare a forward and a backward iteration in STL:

```
container::iterator i;
for (i = c.begin(); i != c.end(); ++i)
     // ...

container::reverse_iterator i;
for (i = c.rbegin(); i != c.rend(); ++i)
     // ...
```

First, the syntax differs. From the STL point of view this is not a serious issue, because iterators are meant to be passed to algorithms as instances. For a wider use, however, this prevents parametric selection of the iterator (i.e., passing the iterator as a type). Second, you have to implement as many `xbegin()` and `xend()` methods as there are schemes of iteration, leading to a higher coupling [17] between iterators and containers.

Another idea consists in the removal of all the iterator related definitions, such as `create_iterator()` or `iterator_type`, from `concrete_aggregate<T>` in order to allow the addition of new iterators without modifying the existing aggregate classes [32]. This can be achieved using *traits classes* [20] to associate iteration schemes with aggregates: the iterated aggregate instance is given as an argument to the iterator constructor. For example we would rewrite the `add()` function as follows.

```
template< class Aggregate_Model >
void add(Aggregate_Model& input,
         typename Aggregate_Model::data_type value)
{
 typename forward_iterator< Aggregate_Model >::type
  iter (input);

 for (iter.first(); !iter.is_done(); iter.next())
  iter.current_item () += value;
}
```

This eliminates the need to declare iterators into the aggregate class, and allows further additions of iteration schemes by the simple means of creating a new traits class (for example `backward_iterator<T>`).

3.3 Generic Abstract Factory

Intent

To create families of related or dependent objects.

Motivation

Let us go back over the different iteration schemes problem discussed previously. We want to define several kind of iterators for an aggregate, and as so we are candidates for the ABSTRACT FACTORY pattern. The STL example can be rewritten as follows to make this pattern explicit: iterators are *products*, built by an aggregate which can be seen as a *factory*.

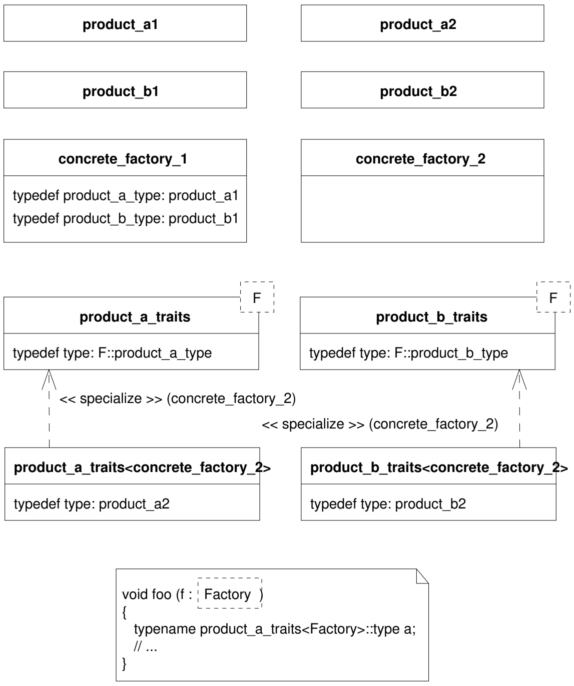
```
factory_a::product_1 i;
for (i = c.begin(); i != c.end(); ++i)
     // ...

factory_a::product_2 i;
for (i = c.rbegin(); i != c.rend(); ++i)
     // ...
```

Implementing a GENERIC ABSTRACT FACTORY is therefore just a matter of defining the product types in the classes that should be used as a factory. This is really simpler than the original pattern. Yet there is one significant difference in usage: an ABSTRACT FACTORY returns an *object* whereas a GENERIC ABSTRACT FACTORY returns a *type*, giving more flexibility (e.g. constructors can be overloaded).

We have shown that if we want to implement multiple iteration schemes, it is better to use traits classes, to define the schemes out of the container. A trait class if a GENERIC ABSTRACT FACTORY too (think of `trait::type` as `factory::product`). But one issue is that these two techniques are not homogeneous. Say we want to add a new iterator to the STL containers: we cannot change the container classes, therefore we define our new iterator in a traits, but now we must use a different syntax whether we use one iterator or the other.

The structure we present here takes care of this: both internal and external definitions of products can be made, but the user will always use the same syntax.

Structure



Here, we represent a parametric method by boxing its parameter. For instance, `Factory` is a type-parameter of the method `Accept`. This does cannot conform to UML since UML lacks support for parametric methods.

Participants

We have two factories, named `concrete_factory_1` and `concrete_factory_2` which each defines two products: `product_a_type` and `product_b_type`. The first factory define the products intrusively (in it's own class), while the second do it externally (in the product's traits).

To unify the utilization, the traits default is to use the type that might be defined in the "factory" class. For example the type `a` defined in `foo<Factory>`, defined as `product_a_trait<Factory>::type` will equal to `concrete_factory_1::product_a_type` in the case `Factory` is `concrete_factory_1`.

Consequences

Contrary to the pattern of Gamma, inheritance is no longer needed, neither for factories, nor for products. Introducing a new product merely requires adding a new parametrized structure to handle the types aliases (e.g., `product_c_traits`), and to specialize this structure when the alias `product_c_type` is not provided by the factory.

Known Uses

Many uses of this pattern can be found in STL. For example all the containers whose contents can be browsed forwards or backwards[3] define two products: forward and backward iterators.

The actual type of a list iterator never explicitly appears in client code, as for any class name of concrete products. Rather, the user refers to `A::iterator`, and `A` is an STL container used as a concrete factory.

3.4 Generic Template Method

Intent

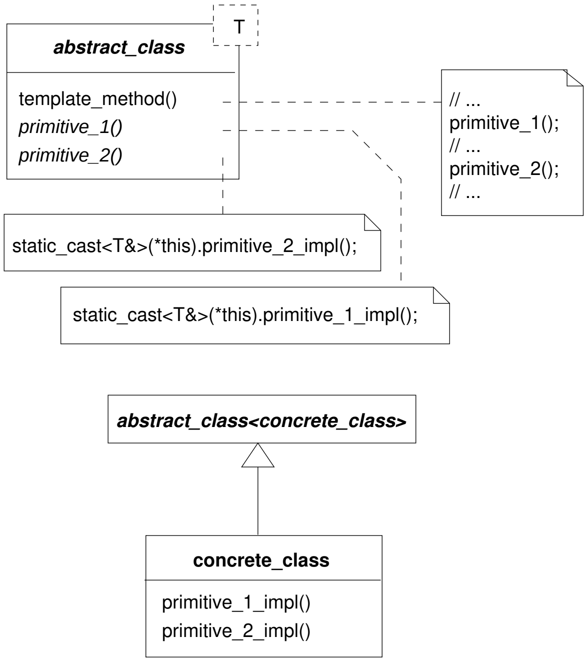
To define the canvas of an *efficient* algorithm in a superior class, deferring some steps to subclasses.

Motivation

In generic programming, we limit inheritance to factor methods [section 2.3]; here, we want a superior class to define an operation some parts of which (primitive operations) are defined only in inferior classes. As usual we want calls to the primitive operations, as well as calls to the template method, to be resolved at compile-time.

[3] vectors, doubly linked lists and dequeues are models of this concept, named *reversible containers*

Structure



Participants

In the object-oriented paradigm, the selection of the target function in a polymorphic operation can be seen as a search for the function, browsing the inheritance tree upwards from the dynamic type of the object. In practice, this is done at run-time by looking up the target in a table of function pointers.

In generic programming, we want that selection to be solved at compile-time. In other words, each caller should statically know the dynamic type of the object from which it calls methods. In the case of a superior class calling a method defined in a child class, the knowledge of the dynamic type can be given as a template parameter to the superior class. Therefore, any class needing to know its dynamic type will be parameterized by its leaf type.

The parametric class `abstract_class` defines two operations: `primitive_1()` and `primitive_2()`. Calling one of these operations leads to casting the target object into its dynamic type. The methods executed are the implementations of these operations, `primitive_1_impl()` and `primitive_2_impl()`. Because the object was cast into its leaf type, these functions are searched for in the object hierarchy from the leaf type up as desired.

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

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 code, including the template method itself. Hence, this design is more efficient.

Implementation

The methods `primitive_1()` and `primitive_2()` do not contain their implementation but a call to an implementation; they can be considered as *abstract methods*. Please note that they can also be called by the client without knowing that some dispatch is performed.

This design is made possible by the typing model used for C++ template parameters. A C++ compiler has to delay its semantic analysis of a template function until the function is instantiated. The compiler will therefore accept the call to `T::primitive_1_impl()` without knowing anything about `T` and will check the presence of this method later when the call to the `A<T>::primitive_1()` is actually performed, if it ever is. In Ada [13], on the contrary, such postponed type checking does not exist, for a function shall type check even if it is not instantiated. This pattern is therefore not applicable *as is* in this language.

One disadvantage of this pattern over Gamma's implementation is directly related to this: the compiler won't check the actual presence of the implementations in the subclasses. While a C++ compiler will warn you if you do not supply an implementation for an abstract function, even if it is not used, that same compiler will be quiet if pseudo-virtual operations like `primitive_1_impl()` are not defined and not used. Special care must thus be taken when building libraries not to forget such functions since the error won't come to light until the function is actually used.

We purposely added the suffixes `_impl` to the name of primitives to distinguish the implementation functions. One could image that the implementation would use the same name as the primitive, but this require some additional care as the abstract primitive can call itself recursively when the implementation is absent.[4]

Sample Code

The following code shows how to define a `get_next()` operation in each iterator of a library of containers. Obviously, `get_next()` is a template method made by issuing successive calls to the `current_item()` and `next()` methods of the actual iterator.

We define this method in a superclass `iterator_common` parametrized by its subtype, and have all iterators derive from this class.

```
template< class Child, class Value_Type >
class iterator_common
{
 public:
  Value_Type& get_next () {
   // template method
   Value_Type& v = current_item ();
   next ();
   return v;
  }
  Value_Type& current_item () {
   // call the actual implementation
   static_cast<Child&>(*this).current_item_impl();
  }
  void next () {
   // call the actual implementation
   static_cast<Child&>(*this).next_impl();
  }
};


// sample iterator definition
template< class Value_Type >
class buffer_iterator: public
  iterator_common< buffer_iterator< Value_Type >,
                   Value_Type >
{
  public:
    Value_Type current_item_impl () { ... };
    void next_impl () { ... };
    void first () { ... };
    void is_done () { ... };
    // ...
};
```

Known Uses

This pattern relies on an idiom called *Curiously Recurring Template* [4] derived from the *Barton and Nackman*

[4]You can ensure at compile-time that two functions (the primitive and its implementation) are different by passing their addresses to a helper template specialized in the case its two arguments are equal.

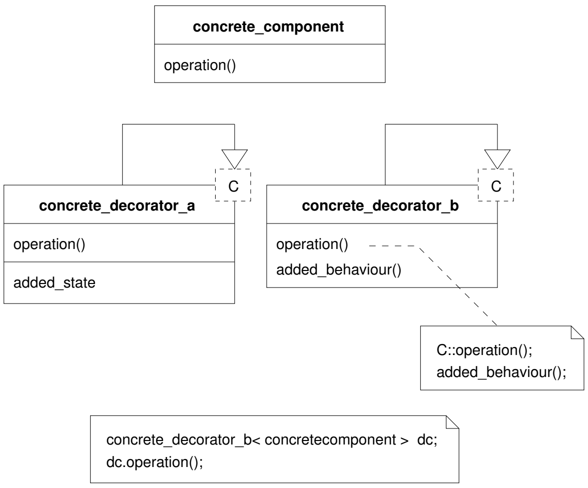
Trick [2]. In [2] this idiom is used to define a binary operator (for instance +) in a superior class from the corresponding unary operator (here +=) defined in an inferior class. Further examples are given in [30].

3.5 Generic Decorator

Intent

To *efficiently* define additional responsibilities to a set of objects or to replace functionalities of a set of objects, by means of subclassing.

Structure



We use a special idiom: having a parametric class that derives from one of its parameters. This is also known as *mixin inheritance*[5] [3].

Participants

A class `concrete_component` which can be decorated, offers an operation `operation()`. Two parametric decorators, `concrete_decorator_a` and `concrete_decorator_b`, whose parameter is the decorated type, override this operation.

[5] Mixins are often used in Ada to simulate multiple inheritance [14].

Consequences

This pattern has two advantages over Gamma's. First, any method that is not modified by the decorator is automatically inherited. While Gamma's version uses composition and must therefore delegate each unmodified operation. 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 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 redefines the data access operator `operator*()` of the iterator.

```
// A basic red-green-blue struct
template< class T >
struct rgb
{
  typedef T red_type;
  red_type red;

  typedef T green_type;
  green_type green;

  typedef T blue_type;
  blue_type blue;
};
```

```
// An accessor class for the red field.
template< class T >
class get_red
{
  public:
    typedef T input_type;
    typedef typename T::red_type output_type;

    static output_type&
    get (input_type& v) {
      return v.red;
    }

    static const output_type&
    get (const input_type& v) {
      return v.red;
    }
};
```

Note how the `rgb<T>` structure exposes the type of each attribute. This makes cooperation between objects easier: here the `get_red` accessor will look up the `red_type` type member and doesn't have to know that fields of `rgb<T>` are of type `T`. `get_red` can therefore apply to any type that features `red` and `red_type`, it is not limited to `rgb<T>`.

```
// A decorator for any iterator
template< class Decorated,
          template< class > class Access >
class field_access: public Decorated
{
 public:
  typedef typename Decorated::value_type value_type;
  typedef Access< value_type >           accessor;
  typedef typename accessor::output_type output_type;

  field_access () : Decorated () {}
  field_access (const Decorated& d) : Decorated (d) {}

  // Overload operator*, use the given accessor
  // to get the proper field.
  output_type& operator* () {
    return accessor::get (Decorated::operator* ());
  }

  const output_type& operator* () const {
    return accessor::get (Decorated::operator* ());
  }
};
```

`field_access` is a decorator whose parameters are the types of the decorated iterator, and of a helper class which specifies the field to be accessed. Actually, this second parameter is an example of the GENERIC STRATEGY pattern [6, 30].

```
int main ()
{
  typedef std::list< rgb< int > > A;
  A input;
  // ... initialize the input list ...

  // Build decorated iterators.
  field_access< A::iterator, get_red >
    begin = input.begin (),
    end   = input.end ();
  // Assign 10 to each red field.
  std::fill (begin, end, 10);
}
```

The `std::fill()` procedure is a standard STL algorithm which assigns a value to each element of a range (specified by two iterators). Since `std::fill()` is here given decorated iterators it will only assign red fields to 10.

Note that the decorator is independent of the decorated iterator: it can apply to any STL iterator, not only `list<T>::iterator`. The `std::fill()` algorithm will use methods of `field_access` inherited from the decorated iterator, such as the assignment, comparison, and pre-increment operators.

Known Uses

Parameterized inheritance is also called *mixin inheritance* and is one way to simulate multiple inheritance in Ada 95 [14]. This can also be used as an alternate way for providing *template methods* [6].

3.6 Generic Visitor

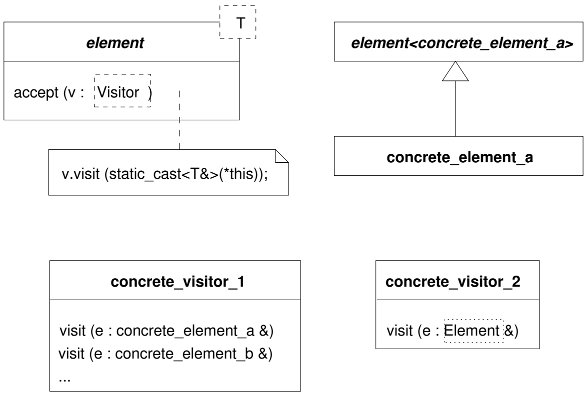
Intent

To define a new operation for the concrete classes of a hierarchy without modifying the hierarchy.

Motivation

In the case of the VISITOR pattern, the operation varies with the type of the operation target. Since we assume to know the exact type as compile-time, a trivial design is thus to define this operation as a procedure overloaded for each target. Such a design, however, does not have the advantages of the translation of the VISITOR pattern proposed in the next section.

Structure



Participants

In the original Gamma's pattern the method `accept` has to be defined in each `element`. The code of each of these `accept` method can be the same[6], only type of the

[6]This is not actually the case in Gamma's book, because the name of the visiting method to call is dependent on the element type; how-

this pointer changes. Here we use the same trick as the GENERIC TEMPLATE METHOD to factor accept in the superclass.

Each visitor defines a method visit, for each element type that it must handle. visit can be either an overloaded function (as in concrete_visitor_1) or a function template (as in concrete_visitor_2). In both case, the overload resolution or function instantiation is made possible by the exact knowledge of the element type.

One advantage of using a member template (as in concrete_visitor_2), over an overloaded function (as in concrete_visitor_1) is that the concrete_visitor_2 class does need to be changed when new type are added: the visitor can be specialized externally should the default be inadequate.

Consequences

The code is much closer to the one of Gamma than the trivial design presented before, because the visitor is here an object with all its advantages (state, life duration).

While accept and visit does not return anything in the original pattern, they can be taught to. It the GENERIC ITERATOR they can even return a type dependent on the visitor's type. As the following example shows.

Sample Code

Let's consider an image2d class the pixels of which should be addressable using different kind of positions (Cartesian or polar coordinates, etc.). For better modularity, we don't want the image2d to known all position types. Therefore we see positions as visitors, which the image accepts. accept returns the pixel value corresponding to the supplied position. The image will provide only one access method, and it is up to the visitor to perform necessary conversion (e.g. polar to Cartesian) to use this interface.

A position may also refer to a particular channel in a color image. The accept return type is thus dependent on the visitor. We will use a traits to handle this.

```
template< class Visitor, class Visited >
struct visit_return_trait;
```

For each pair (Visitor, Visited) visit_return_trait<Visitor,Visited>::type is the return type of access and visit.

```
// factor the definition of accept for all images
template < class Child >
class image {
public:
  template < typename Visitor >
  typename visit_return_trait< Visitor, Child >::
  type accept (Visitor& v) {
    return v.visit (static_cast< Child& > (*this));
  }
  // ... likewise for const accept
};
```

```
template< typename T >
class image_2d : public image< image_2d< T > > {
public:
  typedef T pixel_type;
  // ...
  T& get_value (int row, int col){...}
  const T& get_value const (int row, int col){...}
};
```

Here is one possible visitor, with it's corresponding visit_return_trait specialization.

```
class point_2d {
public:
  point_2d (int row, int col) { ... }

  template < typename Visited >
  typename Visited::pixel_type&
  visit (Visited& v) {
    return v.get_value (row, col);
  }
  // ...
  int row, col;
};

template< class Visited >
struct visit_return_trait< point_2d, Visited > {
  typedef typename Visited::pixel_type type;
};
```

channel_point_2d is another visitor, which must be parametered to access a particular layer (as in the decorator example).

ever, using the same name (visit) for all these methods make no problem in any language as C++ which support function overloading.

```
template< template< class > class Access >
class channel_point_2d {
public:
  channel_point_2d (int row, int col) { ... }

  template < typename Visited >
  typename Access< typename Visited::pixel_type >::
  output_type& visit (Visited& v) {
    return Access< typename Visited::pixel_type >::
      get (v.get_value (row, col));
  }
  // ...
};

template< template< class > class Access,
          class Visited >
struct visit_return_trait
  < channel_point_2d< Access >, Visited > {
  typedef typename
    Access< typename Visited::pixel_type >::
      output_type type;
};
```

Finally, the following hypothetical main shows how the return value of `accept` differ according to the visitor used.

```
int main () {
  image_2d< rgb< int > > img;
  point_2d p(1, 2);
  channel_point_2d<get_red> q(3, 4);

  int v      = img.accept (p);
  rgb<int> w = img.accept (q);
}
```

In our library, `accept` and `visit` are both named `operator[]` so we can write `img[p]` or `p[img]` at will.

4 Conclusion and Perspectives

Based on object programming, generic programming allows to build and assemble reusable components [15] and proved to be useful where efficiency is required.

Since generic programming (or more generally *Generative programming* [31, 6]) is becoming more popular and because much experience and knowledge have been accumulated and assimilated in structuring the object-oriented programming, we believe that it is time to explore the benefits that the former can derive from well-proven designs in the latter.

We showed how design patterns can be adapted to the generic programming context by presenting the generic versions of three fundamental patterns from Gamma *et al.* [10]: the GENERIC BRIDGE, GENERIC ITERATOR, the GENERIC ABSTRACT FACTORY, the GENERIC TEMPLATE METHOD, the GENERIC DECORATOR, and the GENERIC VISITOR. We hope that such work can provide some valuable insight, and aid design larger systems using generic programming.

Acknowledgments

The authors are grateful to Philippe Laroque for his fruitful remarks about the erstwhile version of this work, to Colin Shaw for his corrections on an early version of this paper, and to Bjarne Stroustrup for his valuable comments on the final version.

Availability

The source of the patterns presented in this paper, as well as other generic patterns, can be downloaded from `http://www.lrde.epita.fr/download/`.

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] Gilad Bracha and William Cook. Mixin-based inheritance. In *procdings of ACM Conference on Object-Oriented Programming: System, Languages, and APPLICATION (OOPSLA) 1990*. ACM, 1989. URL `http://java.sun.com/people/gbracha/oopsla90.ps`.

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

[5] James A. Crotinger, Julian Cummings, Scott Haney, William Humprey, Steve Karmesin, John Reynders, Stephen Smith, and Timothy J. Williams. Generic programming in POOMA and PETE. In *Dagstuhl seminar on Generic Programming*, April 1998. URL

http://www.acl.lanl.gov/pooma/papers/GenericProgrammingPaper/dagstuhl.pdf.

[6] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming. Methods, Tools, and Applications*. Addison Wesley, 2000. URL http://www.generative-programming.org/.

[7] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *11th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, 1996. URL http://www.cs.McGill.CA/ACL/papers/oopsla96.html.

[8] Alexandre Duret-Lutz. Olena: a component-based platform for image processing, mixing generic, generative and oo programming. In *symposium on Generative and Component-Based Software Engineering, Young Researchers Workshop*, 10 2000. URL http://www.lrde.epita.fr/publications/.

[9] Ulfar Erlingsson and Alexander V. Konstantinou. Implementing the C++ Standard Template Library in Ada 95. Technical Report TR96-3, CS Dept., Rensselaer Polytechnic Institute, Troy, NY, January 1996. URL http://www.adahome.com/Resources/Papers/General/stl2ada.ps.Z.

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

[11] Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-François Mangin. Obtaining genericity for image processing and pattern recognition algorithms. In *15th International Conference on Pattern Recognition (ICPR'2000)*, September 2000. URL http://www.lrde.epita.fr/publications/.

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

[13] *Ada95 Reference Manual*. Intermetrics, Inc., December 1994. URL http://adaic.org/standards/95lrm/. Version 6.00 (last draft of ISO/IEC 8652:1995).

[14] *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, Massachusetts, January 1995. URL ftp://ftp.lip6.fr/pub/gnat/rationale-ada95/.

[15] 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.

[16] Ullrich Köthe. Requested interface. In *In Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97)*, Munich, Germany, 1997. URL http://www.riehle.org/events/europlop-1997/p16final.pdf.

[17] John Lakos. *Large-scale C++ software design*. Addison-Wesley, 1996.

[18] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000. URL http://oonumerics.org/tmpw00/.

[19] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[20] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5):32–35, June 1995. URL http://www.cantrip.org/traits.html.

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

[22] OON. The object-oriented numerics page. URL http://oonumerics.org/oon.

[23] Esa Pulkkinen. Compile-time determination of base-class relationship in C++, June 1999. URL http://lazy.ton.tut.fi/~esap/instructive/base-class-determination.html.

[24] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference manual*. Object Technology Series. Addison-Wesley, 1999.

[25] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000. URL http://oonumerics.org/tmpw00/.

[26] Alex Stepanov and Meng Lee. *The Standard Template Library*. Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, October 1995. URL `http://www.cs.rpi.edu/~musser/doc.ps`.

[27] Bjarne Stroustrup. *The Design and Evolution of C++*. Addision Wesley, June 1994. URL `http://www.research.att.com/~bs/dne.html`.

[28] Bjarne Stroustrup. Why C++ isn't just an Object-Oriented Programming Language. In *OOPSLA'95*, October 1995. URL `http://www.research.att.com/~bs/papers.html`.

[29] Petter Urkedal. Tools for template metaprogramming. web page, March 1999. URL `http://matfys.lth.se/~petter/src/more/metad/index.html`.

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

[31] 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. URL `http://extreme.indiana.edu/~tveldhui/papers/oo98.html`.

[32] Olivier Zendra and Dominique Colnet. Adding external iterators to an existing Eiffel class library. In *32nd conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'99)*, Melbourne, Australia, November 1999. IEEE Computer Society. URL `http://SmallEiffel.loria.fr/papers/tools-pacific-1999.pdf.gz`.