Generic Programming Redesign of Patterns

Thierry Géraud and Alexandre Duret-Lutz

Thierry.Geraud@lrde.epita.fr
Alexandre.Duret-Lutz@lrde.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

Problem

How to improve the performance of design patterns when they are involved in intensive algorithms?

Context

Implementation of algorithms for scientific object-oriented numerical computing.

Example

Let us consider a very simple algorithm: the addition of a constant to each element 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 aggregate elements). A general implementation is possible thanks to the ITERATOR pattern [5], 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.CurrentElem() += value;
}
```

In this algorithm, there are only abstract classes (`Aggregate<T>` and `Iterator<T>`) and each iteration involves polymorphic calls, thus, dynamic bindings. The ITERATOR pattern results in a run-time overhead: the algorithm is about twice as slow as a code dedicated to a given aggregate type (a list for instance).

Forces

Abstraction. There should be one procedure for each algorithm. Because algorithms should work for different types, procedures must feature a certain degree of abstraction: they must accept input of various types. Most of the algorithmic entities have to be represented in an abstract way.

Efficiency. However, abstraction should not lead to a computational burden, since scientific computing requires efficient implementations. These abstract implementations should be about as fast as implementations dedicated to a particular data type.

Design. Still, object-oriented modeling of abstraction typically relies on operation polymorphism, which imposes an efficiency penalty due to the huge number of dynamic bindings that usually occur in scientific computing. In particular, design patterns widely use operation polymorphism.

Design quality. Yet, the core ideas captured in many design patterns are design structures that have often proved useful in scientific computing. Sacrificing design patterns simply because there might be efficiency problems is not justified.

Solution

A solution for avoiding operation polymorphism, which is a sort of "run-time genericity", is to heavily rely on parametric polymorphism, i.e. (compile-time) genericity. In particular, abstractions are handled by parameters and/or deduced from parameters. In order to transform the structure of a usual design pattern into its generic version, we rely on several rules.

General rules:

- Inclusion polymorphism is forbidden. 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 assumption of generic programming is that the concrete type of every object is known at compile-time.
- Operation polymorphism (keyword `virtual` in *C*$^{++}$) is excluded because dynamic binding is too expensive. In other words, abstract methods are forbidden. To replace operation polymorphism, we can resort to:
  - parametric classes through the *Curiously Recurring Template* idiom, later described in this paper ;
  - parametric methods, which leads to a form of *ad-hoc* polymorphism (overloading).
- Inheritance is only used to factor methods and to declare attributes that can be shared by several subclasses.

Rules for procedures which use generic patterns:

- A procedure should be parameterized by the types of the procedure input, even if the input itself is parameterized.
- The types of the algorithmic tools used in a procedure should be given as a parameter-type of deduced from a parameter-type.

Example Resolved

Since the type of the procedure argument input is abstract in the classic object-oriented version of the example (given in the "problem" section), this type is now a procedure parameter, say A. The only algorithmic tool is an iterator and its type can be deduced from A thanks to the alias iterator_type. The type of the procedure argument value, which is the type of the aggregate elements, can be deduced from A thanks to the alias data_type. The resulting code of the generic procedure is very close to the previous version[1] :

```
template< typename A >
void add( const A& input, typename A::data_type value )
{
  typename A::iterator_type iter = input.CreateIterator();
  for ( iter.First(); ! iter.IsDone(); iter.Next() )
    iter.CurrentElem() += value;
}
```

When the procedure add is instantiated at compile-time for a given type, no class within the procedure is abstract. For instance, in the code below, the procedure add is instantiated with A set to buffer<int>:

```
int main()
{
  buffer<int> buf;
  //...
  add( buf, 7 );
}
```

and the type aliases defined in the class buffer:

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

allows the compiler to know that the type of the argument value is int and to define the iterator iter with the proper type.

Since all the types used in the procedure are known at compile-time, any method call does not require dynamic binding. Moreover, each call can be inlined by the compiler. The resulting executable has about the same performance as dedicated code.

Finally, the procedure remains an "abstract-like" representation of the algorithm (but without abstract classes) and the ITERATOR pattern has been transformed to become efficient.

[1]The keyword typename before type deductions is required by the language C^{++} to fix ambiguities [13].

Resulting Context

Building classes and procedures with an intensive use of both genericity and type deduction is a quite recent paradigm called “generic programming” [8]. This paradigm addresses two issues:

- having a single abstract procedure per algorithm,
- and having efficient numerical procedures.

As a consequence, for several years, generic programming has been adopted by the object-oriented numerical computing community to build libraries of scientific components. Some of these “generic libraries” are available on the Internet [10] and address various domains (graphs, linear algebra, computational geometry, etc.). Some generic programming idioms have already been discovered and many are listed in [14].

The resulting context is then the implementation of algorithms for scientific object-oriented numerical computing *within* the generic programming paradigm[2]. So, usual design patterns should have a different structure than their known one, and a lot of design patterns from Gamma *et al.* [5] can be translated into this paradigm.

Known Uses

Several generic versions of usual design patterns already appear in generic libraries.

- Most generic libraries use the GENERIC ITERATOR pattern that we have previously described in this paper. Let us mention the *C++ Standard Template Library* [12], *STL* for short. In fact, the generic programming paradigm became popular with the adoption of *STL* by the *C++* standardization committee and was made possible with the addition to this language of new generic capabilities [9].
- In POOMA [6], a scientific framework for multi-dimensional arrays, fields, particles, and transforms, the GENERIC ENVELOPE-LETTER pattern appears. A parametric array type is defined with a parameter-type, called an engine type, to specify the manner of indexing and the types of the indices. An array object, the envelope, defers data lookup to the engine object, being the letter.
- In the REQUESTED INTERFACE pattern [7], a GENERIC BRIDGE is introduced to handle efficiently an adaptation layer which mediates between the offered interfaces of the servers and the requested interfaces of the clients. To this end, a generic bridge class is parameterized by a server class, implements the requested interface and delegates the requests to a server object.

[2] Please note that *generic programming* should not be confused with the notion of *genericity*: generic programming is an intensive use of genericity for software architecture purposes, whereas the common use of genericity in oriented-object programming is dedicated for utility tools (procedures that behave like macros and container classes).

Related Patterns

As we are presenting a pattern of design re-engineering from the "classical" object-oriented paradigm towards the generic programming paradigm (put differently, a pattern of design pattern transformation), this kind of pattern is related to refactoring patterns [4].

In another sense, related patterns are the initial and resulting patterns themselves.

Examples

We now give the description of three generic versions of design patterns, originally from Gamma *et al.* [5]: the GENERIC ITERATOR, the GENERIC TEMPLATE METHOD and the GENERIC DECORATOR. The last two patterns are, as far as we know, original.

The implementations of the designs given in this paper and of some other GOF patterns, translated into the generic programming paradigm, are available at:
`http://www.lrde.epita.fr/download/`.

Name

GENERIC ITERATOR

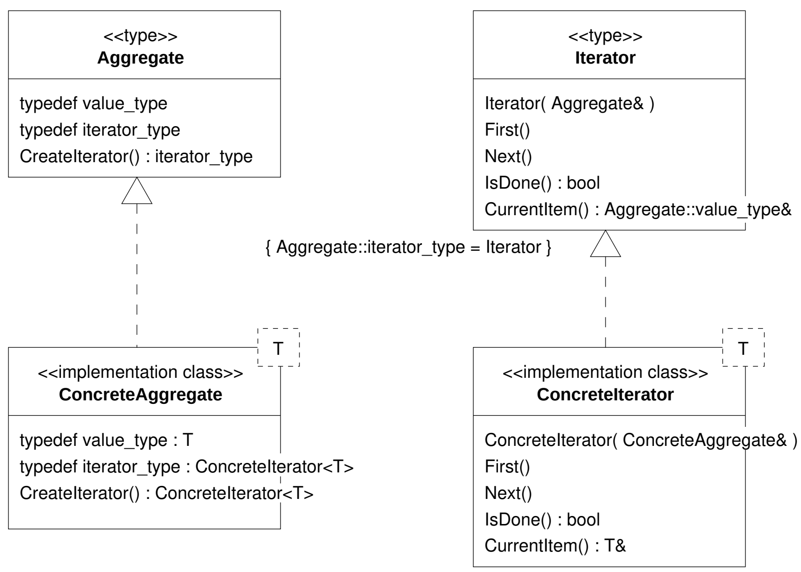
Intent

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

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. A major requirement is that iterations are expected to be efficient (this is an extra requirement compared to the original pattern).

Structure



In this diagram, we use a non-standard extension of UML [11] to represent type aliases in classes.

Participants

In generic libraries, a *concept* [1] is the description of a set of requirements on a type that parameterizes an algorithm implementation (the notion of concept replaces the classical object-oriented notion of abstract class); a type which satisfies these requirements is a *model* of this concept. For this pattern, two concepts are defined: *aggregate* and *iterator*, and two concrete classes, models of these concepts.

Consequences

This design is efficient and allows to implement in an abstract way algorithms which iterate over the elements of an aggregate.

Implementation

An implementation is given in the "example resolved" section.

Known Uses

Most generic libraries use the GENERIC ITERATOR pattern (they can be found on the Internet from the page [10]). Aggregates are, for instance, graphs with various topologies, or matrices with various sparse and dense formats.

Name

GENERIC TEMPLATE METHOD

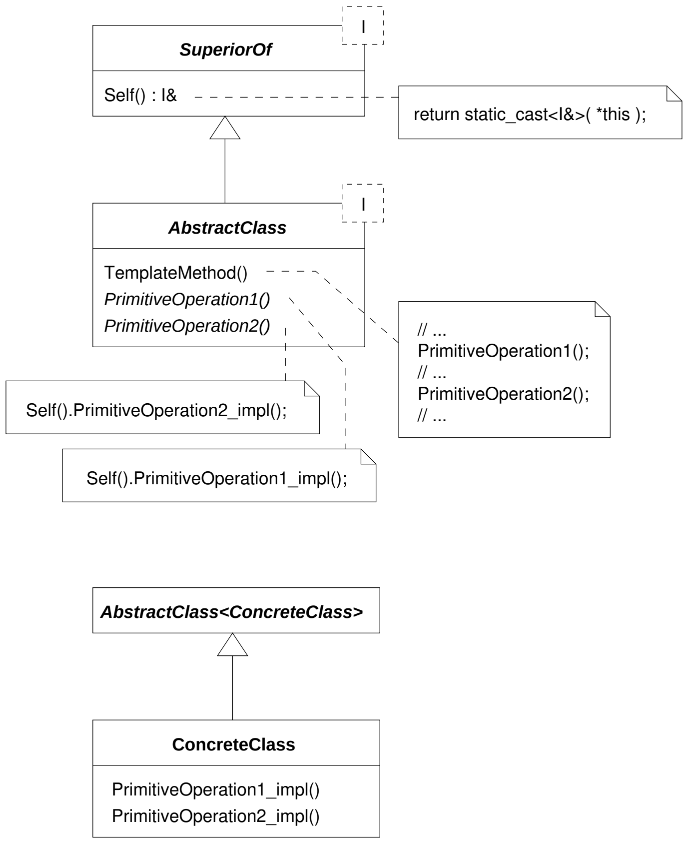
Intent

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

Motivation

We have 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 itself.

Structure



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 starting 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.

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; a template method can be called within an algorithm implementation body.

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 individually 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.

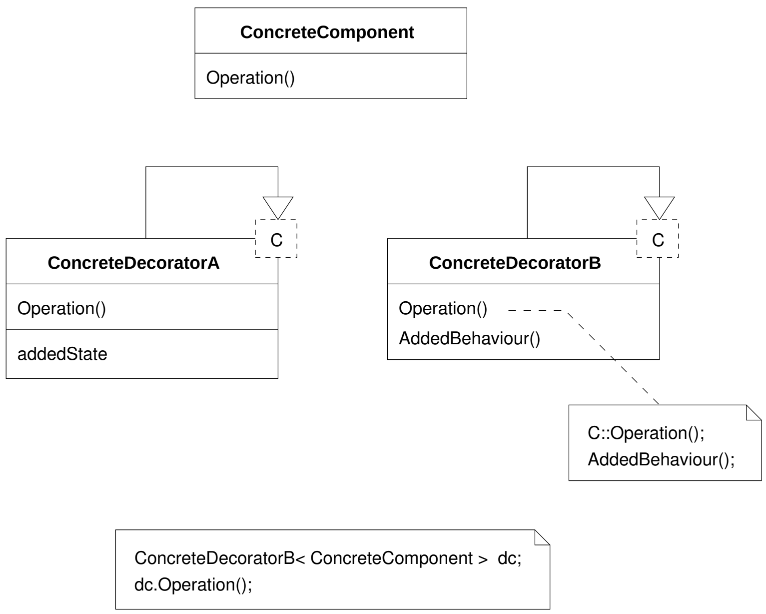
Name

GENERIC DECORATOR

Intent

Efficiently define additional responsibilities to an object or replace functionalities of an object, by the means of subclassing.

Structure



In this structure, 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 a substitution since the decorators are inheriting from their parameter.

Consequences

This pattern has two advantages over Gamma's one. First, any method that is not modified by the decorator is automatically inherited. Not only does this free the decorator from having to define these operations, but in addition, any specific method of a decorated object 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 and efficient.

Conversely, in the generic version, we lose the capability of dynamically adding a decoration to an object.

Known Uses

As we have said, the idiom on which the GENERIC DECORATOR relies is original, so we do not know any use of this generic version. However, decorating an iterator of *STL* is truly useful when a container holds structured data, and when one wants to perform operations only on a single field of these data. In order to access this field, the decorator redefines 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}`.

Acknowledgments. The authors would like to thank Andreas Rüping and Philippe Laroque for their fruitful suggestions on this pattern.

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] Brian Foote and William F. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, volume 1 of *Software Patterns Series*, chapter 14. Addison-Wesley, 1995.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns – Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1994.

[6] 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`

[7] Ullrich Köthe. Requested interface. 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] David R. Musser, editor. *Dagstuhl seminar on Generic Programming*, SchloßDagstuhl, Wadern, Germany, April-May 1998.
`http://www.cs.rpi.edu/~musser/gp/dagstuhl/`

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

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

[13] Bjarne Stroustrup. *The C++ Programming Language (Special Edition)*. Addison Wesley, February 2000.

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