Improving Object-Oriented Generic Programming

Alexandre Duret-Lutz and Thierry Géraud

TECHNICAL REPORT 0001
APRIL 2000

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

Document

Improving Object-Oriented Generic Programming

Alexandre Duret-Lutz and Thierry Géraud

Technical Report 0001
April 2000

EPITA Research and Development Laboratory
14-16, rue Voltaire – F-94276 Le Kremlin-Bictre cedex – France
Phone +33 1 44 08 01 01 – Fax. +33 1 44 08 01 99
email {Alexandre.Duret-Lutz,Thierry.Geraud}@lrde.epita.fr

Abstract

Great efforts have gone into building scientific libraries dedicated to particular application domains and a main issue is to manage the large number of data types involved in a given domain. An ideal algorithm mplementation should be general: it should be written once and process data in an abstract way. Moreover, it should be efficient. For several years, some libraries have been using generic programming to address this problem. In this report, we present two major improvements of this paradigm.

In generic libraries, a concept 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 problem is that, until now, concepts are only defined in the documentation. We managed to make them explicit in the program by representing a concept by a type; moreover, (multiple) concept inheritance is fully supported. The procedure signature thus evolves from `void foo( AbstractA& )` for classic object-orientation, and from `template<class A> void foo( A& )` for classic generic programming, towards `template<class A> void foo( ConceptA<A>& )`. This results in a better support of procedure overloading, which is of prime importance for libraries where each algorithm implementation can have numerous variations.

Another problem is that the generic programming paradigm suffers from a lack of appropriate design patterns. We observed that classical patterns of Gamma et al. can be translated into this paradigm while handling operation polymorphism by parametric polymorphism. In these patterns, method calls can be solved statically, because the inferior type of each object in generic programming is known at compile-time. We thus preserve their modularity and reusability properties but we avoid the performance penalty due to their dynamic behavior, which is a critical issue in numerical computing. This results in better design capabilities for object-oriented generic libraries.

Keywords

Object-oriented programming, generic programming, parametric polymorphism, constrained genericity, design patterns.

Contents

Foreword

This report presents the work that we have done about *generic programming* from January 1999 to April 2000. This "foreword" section has been added in September 2000 in order to mention that papers have been accepted whose contents is close to the work presented in this report:

- Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-Franois Mangin. **Obtaining Genericity for Image Processing and Pattern Recognition Algorithms**. In the Proceedings of the *15th International Conference on Pattern Recognition (ICPR'2000)*, IEEE Computer Society, vol. 4, pages 816-819, Barcelona, Spain, September 2000.
  ↪ section 2.

- Thierry Géraud and Alexandre Duret-Lutz (shepherd: Andreas Rüping). **Generic Programming Redesign of Patterns**. In the Proceedings of the *5th European Conference on Pattern Languages of Programs (EuroPLoP'2000)*, Irsee, Germany, July 2000.
  ↪ subsection 4.1

- Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille (shepherd: Bjarne Stroustrup). **Generic Design Patterns in C++**. In the Proceedings of the *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, Texas, USA, January-February 2001.
  ↪ section 4

Lastly, section 3 is still unpublished.

Warning. Since we are still working on generic programming, the contents of this report could be partially obsolete; please, refer to recent papers for getting fresher explanations, further details, and extra references. Do not hesitate to browse our web site from `http://www.lrde.epita.fr/publications/` and to contact us.

1 Introduction

Great efforts have gone into building scientific libraries dedicated to particular application domains and a main issue is to manage the large number of data types involved in a given domain. An ideal algorithm mplementation should be general: it should be written once and process data in an abstract way. Moreover, it should be efficient.

For instance, in image processing, an algorithm implementation should indistinctly accept 2D and 3D images (isotropic or not), regions, region adjacency graphs, image and graph pyramids, sequences, collections and so forth; the types of data contained in these structures is scalar (Boolean, integer or float), complex, composed (e.g. RGB). Practically, existing libraries are usually dedicated to a particular data structure (mostly 2D images) and the implementations of algorithms are restricted to few data types (mostly unsigned 8 bit integers) [13].

For several years, some libraries of scientific numerical components have been using generic programming to address this problem [16, 14]. Some libraries are available on the Internet [27], for various domains: containers, graphs, linear algebra, computational geometry, differential equations, neural networks, visualization, etc.

Thanks to parameterization, generic programming allows to abstractly represent data structures *and* to efficiently implement algorithms [22]. As defined in [21], the main features of this paradigm are given by the following three statements.

- 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.

Please note that *generic programming* should not be confused with *genericity*: generic programming is an intensive use of genericity for a software architectural purpose (and especially for the design of algorithm implementation), whereas the use of genericity in oriented-object programming is usually restricted to utility classes and procedures.

Generic programming highly relies on object-orientation since the notions of class, encapsulation, information hiding, inheritance, overloading and genericity are required. However, inclusion polymorphism [6] is excluded from the generic programming paradigm because, in the context of scientific numerical computing, implementation of algorithms has to be efficient and because dynamic bindings would cause an unacceptable overhead at run-time. The first section, section 2, gives clues to understand the generic programming paradigm and its context, i.e., object-oriented numerical computing.

We then present two major improvements of the object-oriented generic programming paradigm, and we illustrate these two improvements on the design of a highly generic image processing library.

The section 3 explains how to make explicit in programs the notion of *concept* that appears, until now, only in the documentation of generic libraries. This first improvement results in a better support of procedure overloading, which is of prime importance for libraries where each algorithm implementation can have numerous variations.

The section 4 shows that classical patterns of Gamma *et al.* [10] can be translated into this paradigm while handling inclusion and operation polymorphisms by parametric polymorphism. We thus preserve the modularity and reusability properties of these patterns but we avoid the performance penalty due to their dynamic behavior, which is a critical issue in numerical computing. This results in better design capabilities for object-oriented generic libraries.

Last, we conclude and discuss the perspectives of our work in the section 5.

2 Beyond Genericity: Generic Programming

Before considering language and design issues for generic programming, we first present this paradigm. It is illustrated with implementation samples written in C++, which appears to be the consensual language for object-oriented numerical computing (see [27]).

2.1 Genericity and its Limits

If, according to Meyer [19], genericity can be considered as part of the object paradigm, it is mostly used to replace macros by procedures, such as in:

```
template< typename T >
T min( const T& val1, const T& val2 )
{
   return val1 < val2 ? val1 : val2;
}

int main()
{
  int i1 = 1, i2 = 2;
  int i = min( i1, i2 );  // calls min<int>
}
```

and to build utility classes, where a parameter usually represents a data type, such as in:

```
template< typename T >
class list
{
  public:
    void push_front( const T& elt );
  //...
};

int main()
{
  list<string> l;
  l.push_front( "astring" );
}
```

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, given in [10], and 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 three polymorphic calls. For numerical computing applications, the penalty of dynamic binding results in an unacceptable overhead at run-time. For this example, on a PC platform at 333Mhz with LINUX, the execution time is twice longer than the one of a dedicated handwritten code, when the aggregate is a 2D lattice.

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: Simplicity. 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. In order to make the procedure works properly, it should be re-designed such as the version below:

```
template< typename T2, typename T >
T mean( const Aggregate<T>& input )
{
  T2 sum = T2::zero;
  //...
}

int main()
{
  buffer<int_u8> buf;
  //...
  int_u8 value = mean<int_u16>( buf );
}
```

Considering the procedure call in `main`, the compiler is able to deduce from the type of the argument `buf` that the parameter-type `T` is `int_u8` (unsigned integer coded with 8 bit). On the other hand, the parameter-type `T2` has to be given explicitly by the user at the procedure call.

Unfortunately, we do not want that the user has to set a type at procedure call: in order to keep a generic library user-friendly, we want that each call to a procedure remains as simple as in *C*.

At this stage, using genericity in a classic way does not provide solutions to implement a simple algorithm with both properties of efficiency and simplicity. 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_type`; the type of the aggregate elements, say `T`, can be deduced from `A` thanks to the alias `value_type`; and the summation type can be deduced from `T` thanks to the alias `larger_type`. Last, the sum initial value is the constant `zero`, defined as a class attribute in `T`. The resulting code[1] is given below:

[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.

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

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

template< typename A >
typename A::value_type mean( const A& input )
{
  typedef  typename A::value_type  T;
  typename T::larger_type  sum = T::zero;
  typename A::iterator_type  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 without requiring the user to learn heavy call syntaxes [12].

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 computation (the other pieces of the software can still have a classical object-oriented design).

3 Making Concepts Be Explicit

3.1 Concepts and Models

In the "classic" object-oriented paradigm, an algorithm that has to run on several kinds of input is mapped into such a procedure:

```
void foo( AbstractA& data ) { data.meth(); }
```

Inclusion polymorphism is used to pass data to the procedure; method calls in the procedure body are based on operation polymorphism [7]. When the procedure contains intensive computations, this programming paradigm leads to a huge penalty due to method bindings. Even in C++ where all bindings are static, the code can be up to four times slower than dedicated C [12].

In this "classic" approach, the abstract class `AbstractA` declares the interface that the inferior classes implement: requirements are explicit, which helps the programmer to fulfill her task, and helps the compiler to generate accurate error messages.

In generic programming, we still want to express algorithms in an abstract way but without compromising efficiency, i.e., we want to avoid inclusion polymorphism: the concrete type of every object must be known at compile-time. Procedures then can be based on parametric polymorphism [7]:

```
template< typename TypeA >
void foo( TypeA& data ) { data.meth(); }
```

In the procedure body, method calls are plain function calls, which makes code inlining possible, thus allowing further optimization by the compiler.

This kind of software design is algorithm-oriented. Specifically, we call a *concept* a set of requirements on a parameter-type and we call a *model* a type that fulfills these requirements [3] (this is also referred to as the *Generic Liskov Substitution Principle* in [30]). In the sample code above, the concept related to the formal parameter `TypeA` is a class that holds a method `void meth()`. As we reject abstract classes, concepts can only be expressed in comments or in the documentation. The compiler can no longer assist the programmer, and can produce rather cryptic error messages [23] whose "positions" are not even related to the real occurrence of the concept mismatch.

The reason for this is that C++ does not support bounded parametric polymorphism [2], also known as constrained genericity [19]. In Eiffel [18] or Ada 95 [15], a formal parameter can be constrained to be a subclass of another; in ML [20] the possible values of a functor formal parameter are restricted thanks to signatures; signatures can also be simulated in Ada 95 using formal parameters of generic empty packages [9]. To make concepts explicit in C++, we need bounded parametric polymorphism. We want to express algorithms by parameterized procedures with constrained parameter-types like in the pseudo-codes below (the syntax is inspired by a proposal of a generic extension for Java [24]):

```
template< typename TypeA > where TypeA { void meth(); }
void foo( TypeA& data ) { data.meth(); }
```

or:

```
interface ConceptA { void meth(); };

template< class ModelA > where ModelA implements ConceptA
void foo( ModelA& data ) { data.meth(); }
```

but without modifying C++! In this section, we exhibit C++ idioms for the *explicit definition of concepts*, for their *presence in procedure signatures*, and for *checking code correctness* at compile time.

3.2 Unsatisfactory Attempts

Two solutions to ensure at compile time that a formal parameter meets some requirements are given in [31] (see the sections A.1 and A.2); another one, quite similar, is given in [28]. Unfortunately, *the presence of concept does not appear in procedure signatures*. This is a major drawback because it prevents procedure overloading, which is of prime importance for us (see section 3.4.3).

Another solution is to map a concept into a generic version of the PROXY PATTERN [10, 11] (see the section A.3). However, this design introduces an extra cost at runtime compared to "classical" generic programming: construction of concept instances can be a huge penalty when procedure calls are involved in intensive loops.

3.3 Proposal of Concept Specification and Use

The solution that we present now is based on the *Curiously Recurring Template* [8].

3.3.1 Parameter-types Constrained by Inheritance

We propose to translate a concept into a class parameterized by a model class and to have each model class (for instance `A1`) derive from the proper concept class (`ConceptA<A1>` in this case):

```
template< class ModelA >
struct ConceptA
{
  ModelA& self()
   { return static_cast<ModelA&>( *this );}
};

struct A1 : public ConceptA<A1>
{
  void meth() {}
};
```

At a generic procedure call, the conversion of a model object into its corresponding concept type is implicit [1] (`foo` is instantiated at compile time with `ModelA` set to `A1`) and, in the procedure body, the object can be downcast from its concept type back to its model type thanks to the method `self()`:

```
template< class ModelA >
void foo( ConceptA<ModelA>& data )
{
  ModelA& _data = data.self();
  _data.meth();
}

int main()
{
  A1 a;
  foo( a );
}
```

Generic procedures have thus precise signatures, type checking of parameter-types is enforced, and efficiency is preserved. But at this point, we do not express the set of concept requirements.

3.3.2 Deferred Methods

Since the model class is known at the concept class level, implementations of concept methods can be looked up in the model class. Concept methods are then explicit and downcasting is avoided in generic procedures:

```
template< class ModelA >
struct ConceptA
{
  void meth() { self().meth_impl(); }
protected:
  ModelA& self()
   { return static_cast<ModelA&>( *this ); }
};

template< class ModelA >
void foo( ConceptA<ModelA>& data )
{
  data.meth();
}

struct A1 : public ConceptA<A1>
{
  void meth_impl() {}
};

int main()
{
  A1 a;
  foo( a );
}
```

Please note that this implementation distinguishes the declaration of a concept method (e.g. `meth()`) from its "deferred" implementation in a model class (`meth_impl()` in this case) – the different naming prevents `meth()` to call itself when the user forgets to give a method

implementation in a subclass, and also enhances error messages (see section 3.4.2).

3.3.3 Deferred Typedefs

Concept requirements also concern "nested type names", that is the ability to look up a type as member of a class (for instance `X::iterator` where `X` is an *STL* container).

We propose to first declare a traits class [25] and to define nested type names in the concept class as referring to the traits class contents (this implementation avoids naming recursion). Then, the generic procedures can use nested type names:

```
template< class ModelA >
struct ConceptA_traits {};

template< class ModelA >
struct ConceptA
{
  typedef typename ConceptA_traits<ModelA>::TypeB B;
  //...
};

template<class ModelA>
void foo( ConceptA<ModelA>& data )
{
  typename ConceptA<ModelA>::B b;
}
```

Lastly, a model declares "un-nested (!) type names" by specializing the traits class; here `TypeB` for `A1` is `B1`:

```
struct B1 {};


struct A1; // fwd dcl

template< >
struct ConceptA_traits<A1>
{
  typedef B1 TypeB;
};

struct A1 : public ConceptA<A1>
{
  //...
};

int main()
{
  A1 a;
  foo( a );
}
```

A full code sample, combining both deferred methods and typedefs, is available in section A.4; a bigger code sample with several related concepts is also available (section A.10).

3.4 Advantages of our proposal

3.4.1 Good Readability

From both the algorithm programmer and the algorithm user points of view, the use of concepts make the requirements explicit in the prototypes of procedures. Moreover the requirements of a concept are easily available since they are mapped directly as a class.

In our experience of developing and using a generic library, we notice that code is far easier to read and manage when concepts are explicit.

3.4.2 Pertinent Error Messages

In generic libraries, errors messages often make references to the deep internal of their implementation when a type does not fulfill the requirements of the concepts expected by a procedure.

Using the proposed idiom, error messages are greatly enhanced in all cases: when a procedure is called with a wrong model object, when the programmer forgets to define an implementation in a model class for a method declared in its concept, or when an algorithm calls a method that is not part of a concept. The section A.5 contains various errors and shows the corresponding output messages.

From experience, having pertinent error messages turns out to be of great help when maintaining and using a generic library.

3.4.3 Massive Overloading

We are developing an image processing library, Olena [12], based on generic programming. In this library, most of the operations have several options and should be implemented differently depending on the argument types. To take a simple example, the operation of addition does not lead to the same procedure whether you add two images (summation pixel to pixel) or add a constant value to the pixels of an image. It makes sense to want procedure overloading, which is far easier with explicit concepts. It is now possible to declare the procedures:

```
template< class I1, class I2 >
void add( ConceptImage<I1>& ima1, const ConceptImage<I2>& ima2 );
```

and

```
template< class I, class T >
void add( ConceptImage<I>& ima, const ConceptValue<T>& val );
```

whereas, in the "classic" generic programming style, both procedures would share the same prototype.

Having massive overloading is of prime importance for the design of user-friendly libraries since they need to offer a single procedure name for various declinations of an operation, that is, when operations should manage different sets of input (see the textbook case of section A.6).

3.5 Going Further With Concepts

3.5.1 Handling Multiple Concept Compliance

A class can be a model of several concepts at the same time; this is what we call *multiple concept compliance*.

In Olena [12], *Image* is a concept that generalizes structures such as 2D and 3D images with various topology, and *Predicate* is a concept that designates procedure which returns a Boolean value for each pixel of an image (put differently, a predicate can be seen as a binary mask). A 2D image of Booleans (say `Image2D<bool>`) should therefore be both a model of *Image* and a model of *Predicate*.

In C++, the compliance of a model to several concepts is handled with multiple inheritance (see the section A.7). Considering again the latter example, the class `Image2D<bool>`, specialization of `Image2D<T>`, inherits from both concepts *Image* and *Predicate* (see the files of the section A.10).

3.5.2 Handling (Multiple) Concept Refinement

A *refinement* is a relationship between two concepts when one concept is defined by adding requirements to the other one.

Refinement can be easily handled through inheritance with the proposed idiom, and multiple refinement, through multiple inheritance. If it happens that a concept refines several concepts that declare the same methods or the same nested type names, to fix ambiguities, the refined concept has to redeclare these methods or these nested type names. This is illustrated by the section A.8.

When concept refinements lead to a diamond shaped graph, the idiom also works (see the section A.9).

3.5.3 Method Name Binding

Olena [12] supports various concepts of iterators over image structures. Two of them are *forward iterator* and *exhaustive iterator*. The former is used to browse the pixels in an ordered way and declares the methods `first()`, `next()`, `is_done()` and `get_value()`; the latter is just required to browse all the pixels whatever the order and declares `init()`, `other()`, `is_done()` and `get_value()`. This distinction has been made to clarify the assumptions made about the iteration schemes in algorithms, and to allow traversal optimizations in complex image structures. However, in the case of `Image2D<T>`, a single iterator class (`Image2D_Forward_Iterator`) provides the implementations of both concepts.

The problem of having different method names required for the same operation (for instance, `first()` and `init()`) can be solved in two ways. You can add to the iterator class the method `init_impl()` that simply calls `first_impl()`. The drawback of this solution is that it is intrusive: defining a new concept which can rely on existing classes leads to modifying the contents of these classes. A much better solution is to specialize the method `init()` of the exhaustive iterator concept to call `first_impl()` *in lieu* of `init_impl()`:

```
class Image2D_Forward_Iterator :
  public Concept_Forward_Iterator< Image2D_Forward_Iterator >,
  public Concept_Exhaustive_Iterator< Image2D_Forward_Iterator >
{
  public:
    void first_impl() { /*...*/ }
  //...
};

template< >
void Concept_Exhaustive_Iterator< Image2D_Forward_Iterator >::init()
{
  self().first_impl();
}
```

3.5.4 More About Concept Methods

Constness. The proposed idiom relies on inheritance between concepts and models and makes use of method deferring. It thus respects constness issues of variables and methods.

Class Methods. The way instance methods (member functions) are deferred from concepts to models, section 3.3.2, can be extended to class methods (static member functions):

```
template< class Model >
struct Concept
{
  static void smeth() { Model::smeth_impl(); }
  //...
};
```

Parameterized Methods. The idiom also permits that a concept declares parameterized methods and defers their definitions in model classes.

Generic Template Method Pattern. Inheritance in generic programming can also be used to factor methods. In particular, a concept operation can be described in terms of other operations of the same concept. For instance, in a forward iterator concept, a method `get_next_value()` could be directly implemented as a call to `next()` followed by a call to `get_value()`. This is the generic version [11] of the TEMPLATE METHOD pattern of Gamma *et al.* [10]; see section 4.5.

4 Generic Design Patterns

Several generic programming idioms have already been discovered and many are listed in [32]. 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 [17]).

4.1 Pattern Translation

In this section, we show that some design patterns from Gamma *et al.* [10] can be translated into this paradigm. These patterns have a strong dynamic behavior and their design structure seems antagonist with generic programming. Nevertheless, since the inferior type of each object in generic programming is known at compile-time, method calls in these patterns can be solved statically, i.e. at compile-time. To this aim, the structures of the patterns are redesigned [11].

We thus preserve the modularity and reusability properties of common design patterns but we avoid the performance penalty due to dynamic binding, which is a critical issue in numerical computing. This results in better design capabilities for object-oriented generic libraries.

Nota bene: having read the book of Gamma *et al.* [10] is a requirement to understand this section. In particular, we do not repeat the elements that can be found in this book.

4.2 Generic Iterator

Intent

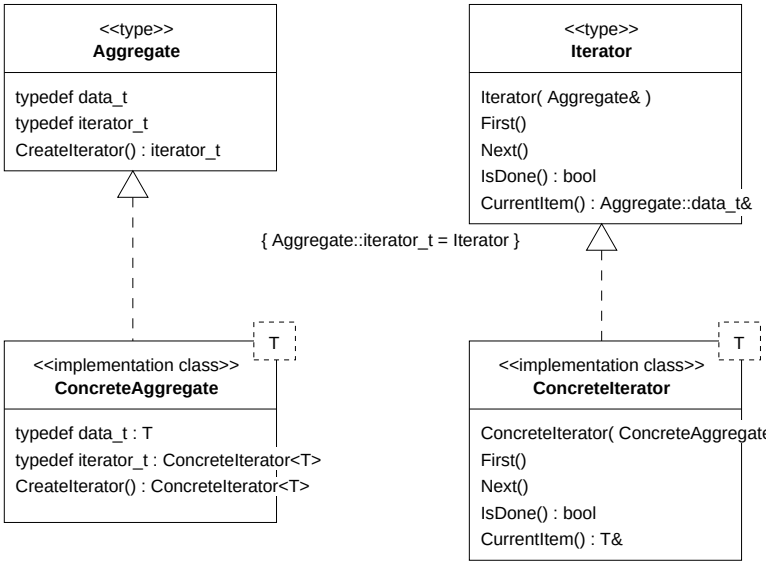
Exactly the same as the original pattern: *"Provide a way to access the elements of an aggregate without exposing its underlying representation."*

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

Structure

In the diagram below, we used a non-standard extension of UML to represent type aliases in classes.



Participants

For this pattern, two concepts are defined: *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* [29]. In fact, generic programming came out 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 [26].

4.3 Generic Visitor

Intent

Exactly the same as the original pattern: *"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."*

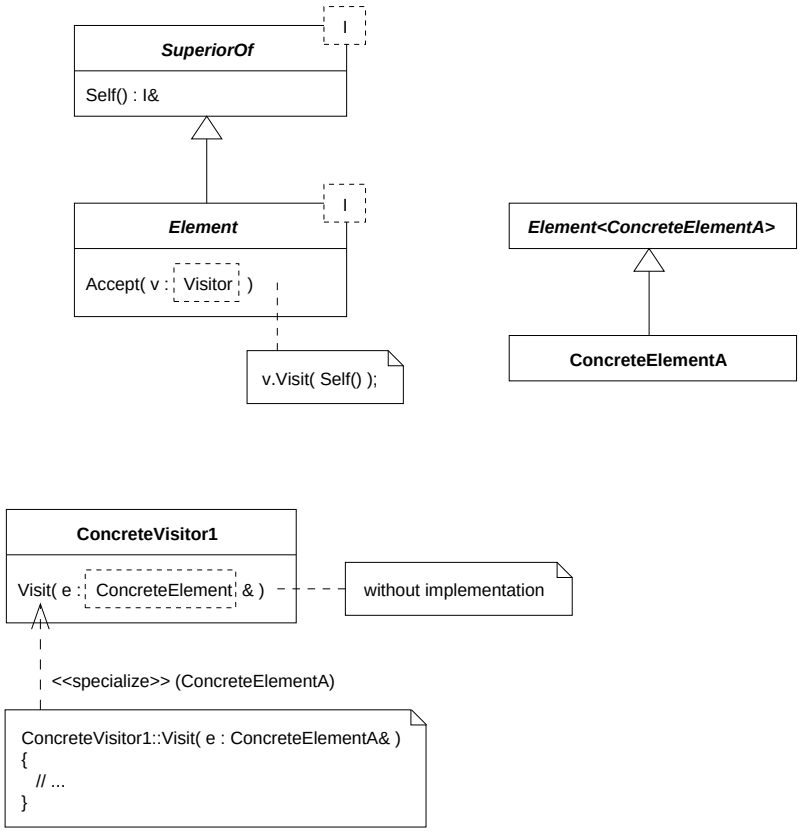
Antecedents

In generic programming, an algorithm implementation does not have to be given when its writing cannot be general (or when it cannot be efficient whatever the data type). In the case of the VISITOR pattern, the operation varies with the type of the operation target. A trivial design of such an operation is to declare (without implementation) a parametric procedure which argument plays the role of the operation target. For a given target type, to define a method leads to define a particular implementation for the procedure. The VISITOR pattern is then a simple specialization of a parametric procedure. The C++ code is given below (please note that the specialization has no parameter).

```
class ConcreteElementA
{
  //...
};

template< class ConcreteElement >
void ConcreteVisit1( ConcreteElement& e );

template<>
void ConcreteVisit1( ConcreteElementA& e )
{
  //...
}

int main()
{
  ConcreteElementA e;
  ConcreteVisit1( e );
}
```

Such a design, very used in generic programming, does not have the avantages of the translation of the VISITOR pattern proposed in the following.

Structure



In this diagram, we represent a parametric method by boxing its parameter. For instance, `Visitor` is a type, parameter of the parametric method `Accept`. As UML does not support parametric methods, we are out of the norm!

Participants

The class `Element` defines the parametric method `Accept` which has a visitor as argument; the type of this visitor is the parameter of the method. Whereas this method has to be defined in all subclasses of `Element` in Gamma's pattern, here, the method is defined only once.

Each visitor, for instance `ConcreteVisitor1`, declares a parametric method `Visit` which argument is the element to be visited; the type of this element is the parameter of the method. Contrary to Gamma's pattern, no hierarchy is required to manage visitors. Defining a visitor for a particular element is done by giving an implementation for a specialization of the method `Visit`.

Please note that the VISITOR pattern design is more concise in generic programming than in object-oriented programming. This is due to two factors: the methods `Accept` and `Visit` have a polymorphic behavior with respect to their argument, and the idiom described in section 4.5 draws a link between `Accept` and `Visit`.

Implementation

```
template< class I >
class Element : SuperiorOf< I >
{
public:
  template< class Visitor >
  void Accept( Visitor v )  { v.Visit( Self() ); }
protected:
  Element()  {}
};

class ConcreteElementA : Element< ConcreteElementA >
{
  //...
};

class ConcreteVisitor1
{
public:
  template< class ConcreteElement >
  void Visit( ConcreteElement& e );
};

template<>
void ConcreteVisitor1::Visit( ConcreteElementA& e )
{
  // ...
}

int main}()
{
  ConcreteElementA e;
  e.Accept( ConcreteVisitor1() );
}
```

The VISITOR pattern that we propose can be implemented in C++ as just above. The code is much closer to the one of Gamma than to the one given at the beginning of this section. The advantage of this new design is twofold: a call to an extrusive method is a real call to the object by the means of the method `Accept`, and the life of a visitor denotes the duration of the method call.

4.4 Generic Abstract Factory

Intent

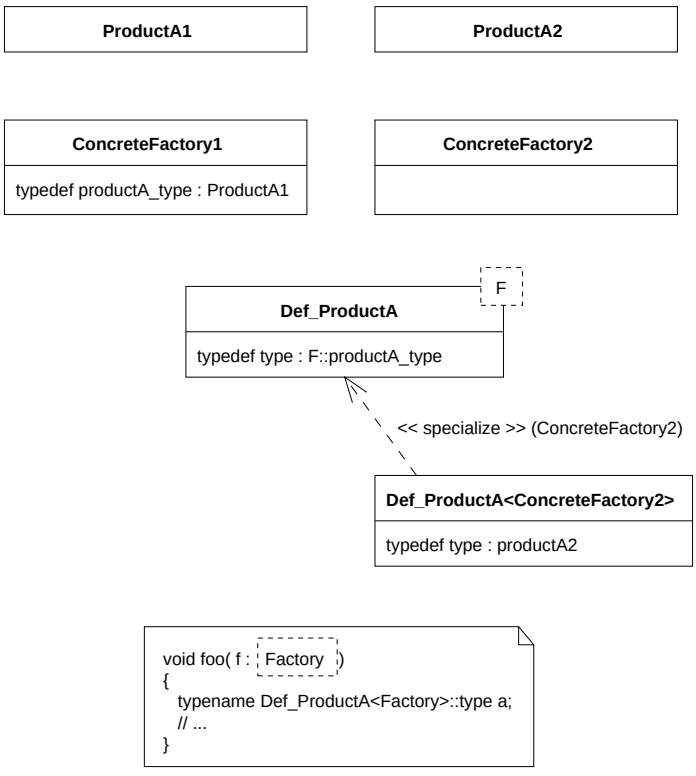
Exactly the same as the original pattern: *"Provide an interface for creating families of related or independent objects without specifying their concrete classes."*

Antecedents

Let us consider again the ITERATOR example. A container plays the role of a factory; the product type is deduced from the factory type (Cf. the structure section). This pattern is used in STL for containers whose contents can be browsed both forwards and backwards (the concept is named "reversible containers"); they are vectors, doubly linked lists and dequeues. They all define two products: forward iterators and backward iterators.

The parametric class `std::__list_iterator`, defined in STL, never explicitely appears in client code, as for any class name of concrete products. On the other side, one manipulates `A::iterator` where `A` is a concrete factory, passed as parameter or given explicitely.

Structure



Participants

We have two kinds of products, `ProductA1` and `ProductA2`, and two kinds of factories, `ConcreteFactory1` and `ConcreteFactory2`. The former factory declares a type definition : the type alias `ConcreteFactory1::productA_type` represents `ProductA1`. On the other

hand, the latter factory declares no alias. In order to make the class `ConcreteFactory2` become a factory, we should have to intrusively add the type definition `productA_type`. We propose an original solution to avoid modifying the existing class: an extrusive type definition which remains compatible with intrusive ones.

To achieve this, a structure `Def_ProductA` parametrized by `F` defines by default the alias `type` as the alias `productA_type` of the class `F`. In this way, `Def_ProductA< ConcreteFactory1>::type` is automatically an alias of `ProductA1` and it is now possible to specialize the structure `Def_ProductA` for the parameter `ConcreteFactory2` in order to make `Def_ProductA<ConcreteFactory2>::type` be an alias of `ProductA2`. Finally, the client uniformly uses intrusive and extrusive aliases (Cf. the procedure `foo`).

Contrary to the pattern of Gamma, inheritance is no longer needed, neither for factories, nor for products. Introducing a new product merely requires to add a new parametrized structure to handle the types aliases (e.g., `Def_ProductB`), and to specialize this structure when the alias `productB_type` is not provided by the factory. When a factory class is created, the types aliases are to be inserted in this new class for each product.

4.5 Generic Template Method

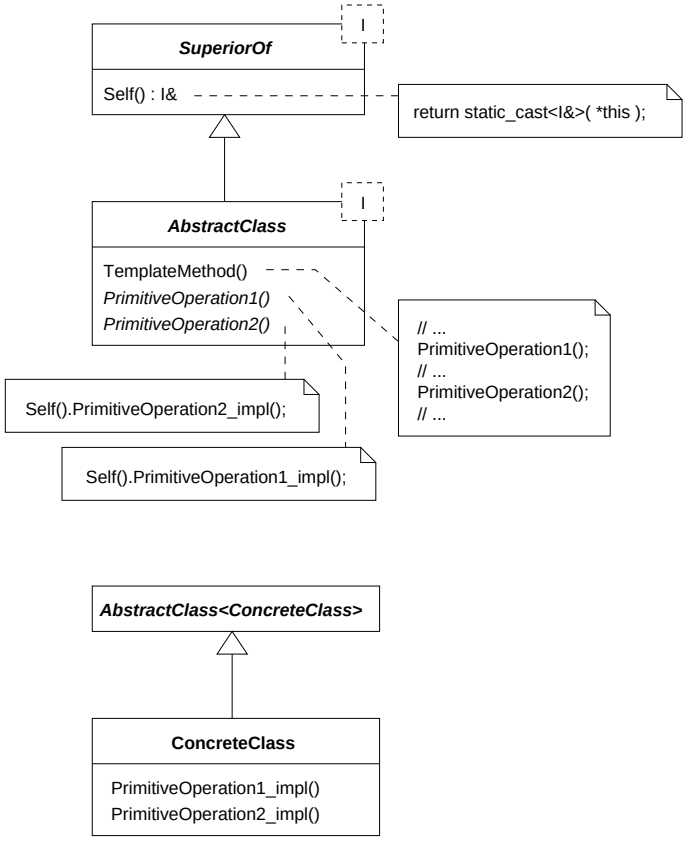
Intent

Exactly the same as the original pattern: *"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure."*

Motivation

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



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.

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

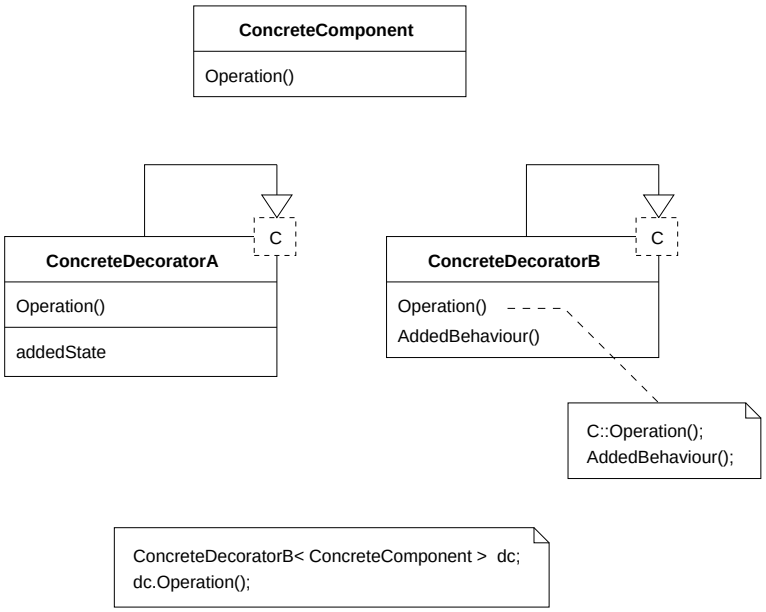
This pattern relies on an idiom (the *Curiously Recurring Template*) given in [8] and based on [5]. In this idiom, an binary operator, for instance `+`, is defined in a superior class from the corresponding unary operator, here `+=`, defined in an inferior class.

4.6 Generic Decorator

Intent

About the same as the original pattern: *"Attach additional responsibilities to an object."*

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 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 [4] 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.

5 Conclusion and Perspectives

Object-oriented generic programming is a 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 [16]. In addition, the generic components are compatibles with traditional components via generative programming [33]. This contrasts with the previous image of generic programming, seen as incompatible with dynamic approaches because of its static orientation.

In this report, 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:

- strong typing can be effective thanks to concept-types ;
- design patterns can be used to better modularize and reuse parts or all of generic programs.

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 report, and Andreas Rüping for his relevant critics on a paper about generic patterns.

A Code for Concept-Type

A.1 `sutter1.cc`

```
#include <assert.h>
#include <iostream>


template< class ModelA >
class ConceptA
{
private:
  bool ValidateRequirements()
  {
    void (ModelA::*meth_ptr)() = &ModelA::meth; // checks exact signature :)
    meth_ptr = 0; // just to avoid a warning
    return true;
  }
public:
  ~ConceptA()
  {
    assert( ValidateRequirements() ); // forces requirement validation :)
  }
};


template< class ModelA >
void foo( ModelA& data ) // no concept presence in signature :(
{
  ConceptA< ModelA >(); // validates requirements at compile time :)
  data.meth();
}


struct A1
{
  void meth()
  {
    std::cout << "A1::meth()" << std::endl;
  }
};


int main()
{
  A1 a;
  foo( a );
}
```

A.2 sutter2.cc

```
#include <assert.h>
#include <iostream>


template< class ModelA >
struct ConceptA
{
  static void meth( ModelA& self ) { return self.meth(); }
};


template< class ModelA >
void foo( ModelA& data ) // no concept presence in signature :(
{
  ConceptA<ModelA>::meth( data ); // unfriendly syntax :(
}


struct A1
{
  void meth()
  {
    std::cout << "A1::meth()" << std::endl;
  }
};


int main()
{
  A1 a;
  foo( a );
}
```

A.3 `proxy.cc`

```
/*
 *  file proxy.cc
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>



// concept definition


template< class ModelA >
struct ConceptA
{
  ConceptA( ModelA& model ) : _model( model )
  {
    std::cout << "proxy ctor" << std::endl;
  }
  void meth() { _model.meth(); }
  ModelA& _model;
};



// generic procedure and its facade


template< class ModelA >
void foo_impl( ConceptA< ModelA > data )
{
  data.meth();
}


template< class ModelA >
void foo( ModelA& data )
{
  foo_impl( ConceptA< ModelA >( data ) );
}



// model


struct A1
{
  void meth()
  {
    std::cout << "A1::meth()" << std::endl;
  }
};



// main


int main()
```

```
{
  A1 a;
  foo( a );
}
```

A.4 `proposal.cc`

```
/*
 *  file proposal.cc
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>



// concept definition


template< class ModelA >
struct ConceptA_traits {};


template< class ModelA >
struct ConceptA
{
  typedef typename ConceptA_traits< ModelA >::TypeB  B;
  void meth() { self().meth_impl(); }
protected:
  ModelA& self()
   { return static_cast< ModelA& >( *this ); }
};



// generic procedure


template< class ModelA >
void foo( ConceptA< ModelA >& data )
{
  typename ConceptA< ModelA >::B*  b;
  b = 0;
  data.meth();
}



// model


struct B1 {};


struct A1;


template<>
struct ConceptA_traits< A1 >
{
  typedef B1 TypeB;
};


struct A1 : public ConceptA< A1 >
{
  void meth_impl()
```

```
  {
    std::cout << "A1::meth_impl()" << std::endl;
  }
};


// main


int main()
{
  A1 a;
  foo( a );
}
```

A.5 Error messages

A.5.1 `proposal-err1.cc`

```
/*                                                      //  1
 *  file proposal-err1.cc                               //  2
 *                                                      //  3
 *  Copyright (C) 2000 EPITA-LRDE                       //  4
 *  EPITA Research and Development Laboratory           //  5
 */                                                     //  6
                                                        //  7
#include <iostream>                                     //  8
                                                        //  9
                                                        // 10
                                                        // 11
// concept definition                                   // 12
                                                        // 13
                                                        // 14
template< class ModelA >                                // 15
struct ConceptA_traits {};                              // 16
                                                        // 17
                                                        // 18
template< class ModelA >                                // 19
struct ConceptA                                         // 20
{                                                       // 21
  typedef typename ConceptA_traits< ModelA >::TypeB  B; // 22
  void meth() { self().meth_impl(); }                   // 23
protected:                                              // 24
  ModelA& self()                                        // 25
   { return static_cast< ModelA& >( *this ); }          // 26
};                                                      // 27
                                                        // 28
                                                        // 29
                                                        // 30
// generic procedure                                    // 31
                                                        // 32
                                                        // 33
template< class ModelA >                                // 34
void foo( ConceptA< ModelA >& data )                    // 35
{                                                       // 36
  typename ConceptA< ModelA >::B*  b;                   // 37
  b = 0;                                                // 38
  data.meth();                                          // 39
}                                                       // 40
                                                        // 41
                                                        // 42
                                                        // 43
// model                                                // 44
                                                        // 45
                                                        // 46
struct B1 {};                                           // 47
                                                        // 48
                                                        // 49
struct A1;                                              // 50
                                                        // 51
                                                        // 52
template<>                                              // 53
struct ConceptA_traits< A1 >                            // 54
{                                                       // 55
  typedef B1 TypeB;                                     // 56
};                                                      // 57
                                                        // 58
                                                        // 59
struct A1 : public ConceptA< A1 >                       // 60
```

```
{                                                                   // 61
                                                                    // 62
                                                                    // 63
                                                                    // 64
                                                                    // 65
};                                                                  // 66
                                                                    // 67
                                                                    // 68
                                                                    // 69
// main                                                             // 70
                                                                    // 71
                                                                    // 72
int main()                                                          // 73
{                                                                   // 74
  A1 a;                                                             // 75
  foo( a );                                                         // 76
}                                                                   // 77


/*

bug: a model does not define an implementation for a method declared
as part of its concept.

error message:

proposal-err1.cc: In method ‘void ConceptA<A1>::meth()’:
proposal-err1.cc:39:   instantiated from ‘foo<A1>(ConceptA<A1> &)’
proposal-err1.cc:76:   instantiated from here
proposal-err1.cc:23: no matching function for call to ‘A1::meth_impl ()’

*/
```

A.5.2 `proposal-err2.cc`

```
/*                                                                  //  1
 *  file proposal-err2.cc                                          //  2
 *                                                                 //  3
 *  Copyright (C) 2000 EPITA-LRDE                                  //  4
 *  EPITA Research and Development Laboratory                      //  5
 */                                                                //  6
                                                                   //  7
#include <iostream>                                                //  8
                                                                   //  9
                                                                   // 10
                                                                   // 11
// concept definition                                              // 12
                                                                   // 13
                                                                   // 14
template< class ModelA >                                           // 15
struct ConceptA_traits {};                                         // 16
                                                                   // 17
                                                                   // 18
template< class ModelA >                                           // 19
struct ConceptA                                                    // 20
{                                                                  // 21
  typedef typename ConceptA_traits< ModelA >::TypeB  B;            // 22
  void meth() { self().meth_impl(); }                              // 23
protected:                                                         // 24
  ModelA& self()                                                   // 25
   { return static_cast< ModelA& >( *this ); }                     // 26
};                                                                 // 27
```

```
                                                        // 28
                                                        // 29
                                                        // 30
// generic procedure                                    // 31
                                                        // 32
                                                        // 33
template< class ModelA >                                // 34
void foo( ConceptA< ModelA >& data )                    // 35
{                                                       // 36
  typename ConceptA< ModelA >::B*  b;                   // 37
  b = 0;                                                // 38
  data.meth();                                          // 39
}                                                       // 40
                                                        // 41
                                                        // 42
                                                        // 43
// model                                                // 44
                                                        // 45
                                                        // 46
struct B1 {};                                           // 47
                                                        // 48
                                                        // 49
struct A1;                                              // 50
                                                        // 51
                                                        // 52
template<>                                              // 53
struct ConceptA_traits< A1 >                            // 54
{                                                       // 55
  typedef B1 TypeB;                                     // 56
};                                                      // 57
                                                        // 58
                                                        // 59
struct A1 : public ConceptA< A1 >                       // 60
{                                                       // 61
  void meth_impl()                                      // 62
  {                                                     // 63
    std::cout << "A1::meth_impl()" << std::endl;        // 64
  }                                                     // 65
};                                                      // 66
                                                        // 67
                                                        // 68
                                                        // 69
// main                                                 // 70
                                                        // 71
                                                        // 72
int main()                                              // 73
{                                                       // 74
  struct A2 {} a;                                       // 75
  foo( a );                                             // 76
}                                                       // 77


/*

bug: an object passed to a generic procedure is not a model of the
proper concept.

error message:

proposal-err2.cc: In function ‘int main()’:
proposal-err2.cc:76: no matching function for call to ‘foo (main()::A2 &)’

*/
```

A.5.3 `proposal-err3.cc`

```
/*                                                      //  1
 *  file proposal-err3.cc                               //  2
 *                                                      //  3
 *  Copyright (C) 2000 EPITA-LRDE                       //  4
 *  EPITA Research and Development Laboratory           //  5
 */                                                     //  6
                                                        //  7
#include <iostream>                                     //  8
                                                        //  9
                                                        // 10
                                                        // 11
// concept definition                                   // 12
                                                        // 13
                                                        // 14
template< class ModelA >                                // 15
struct ConceptA_traits {};                              // 16
                                                        // 17
                                                        // 18
template< class ModelA >                                // 19
struct ConceptA                                         // 20
{                                                       // 21
  typedef typename ConceptA_traits< ModelA >::TypeB  B; // 22
  void meth() { self().meth_impl(); }                   // 23
protected:                                              // 24
  ModelA& self()                                        // 25
   { return static_cast< ModelA& >( *this ); }          // 26
};                                                      // 27
                                                        // 28
                                                        // 29
                                                        // 30
// generic procedure                                    // 31
                                                        // 32
                                                        // 33
template< class ModelA >                                // 34
void foo( ConceptA< ModelA >& data )                    // 35
{                                                       // 36
  typename ConceptA< ModelA >::B*  b;                   // 37
  b = 0;                                                // 38
  data.other_meth();                                    // 39
}                                                       // 40
                                                        // 41
                                                        // 42
                                                        // 43
// model                                                // 44
                                                        // 45
                                                        // 46
struct B1 {};                                           // 47
                                                        // 48
                                                        // 49
struct A1;                                              // 50
                                                        // 51
                                                        // 52
template<>                                              // 53
struct ConceptA_traits< A1 >                            // 54
{                                                       // 55
  typedef B1 TypeB;                                     // 56
};                                                      // 57
                                                        // 58
                                                        // 59
struct A1 : public ConceptA< A1 >                       // 60
{                                                       // 61
  void meth_impl()                                      // 62
```

```
  {                                                                  // 63
    std::cout << "A1::meth_impl()" << std::endl;                     // 64
  }                                                                  // 65
};                                                                   // 66
                                                                     // 67
                                                                     // 68
                                                                     // 69
// main                                                              // 70
                                                                     // 71
                                                                     // 72
int main()                                                           // 73
{                                                                    // 74
  A1 a;                                                              // 75
  foo( a );                                                          // 76
}                                                                    // 77


/*

bug: a generic procedure calls a method that is not declared as part of
a concept.

error message:

proposal-err3.cc: In function ‘void foo<A1>(ConceptA<A1> &)’:
proposal-err3.cc:76:   instantiated from here
proposal-err3.cc:39: no matching function for call to ‘ConceptA<A1>::other_meth ()’

*/
```

A.5.4 `proposal-err4.cc`

```
/*                                                                   //  1
 *  file proposal-err4.cc                                           //  2
 *                                                                  //  3
 *  Copyright (C) 2000 EPITA-LRDE                                   //  4
 *  EPITA Research and Development Laboratory                       //  5
 */                                                                 //  6
                                                                     //  7
#include <iostream>                                                  //  8
                                                                     //  9
                                                                     // 10
                                                                     // 11
// concept definition                                               // 12
                                                                     // 13
                                                                     // 14
template< class ModelA >                                             // 15
struct ConceptA_traits {};                                           // 16
                                                                     // 17
                                                                     // 18
template< class ModelA >                                             // 19
struct ConceptA                                                      // 20
{                                                                    // 21
  typedef typename ConceptA_traits< ModelA >::TypeB  B;              // 22
  void meth() { self().meth_impl(); }                                // 23
protected:                                                           // 24
  ModelA& self()                                                     // 25
   { return static_cast< ModelA& >( *this ); }                       // 26
};                                                                   // 27
                                                                     // 28
                                                                     // 29
                                                                     // 30
```

```
// generic procedure                                          // 31
                                                              // 32
                                                              // 33
template< class ModelA >                                      // 34
void foo( ConceptA< ModelA >& data )                          // 35
{                                                             // 36
  typename ConceptA< ModelA >::B*  b;                         // 37
  b = 0;                                                      // 38
  data.meth();                                                // 39
}                                                             // 40
                                                              // 41
                                                              // 42
                                                              // 43
// model                                                      // 44
                                                              // 45
                                                              // 46
struct B1 {};                                                 // 47
                                                              // 48
                                                              // 49
                                                              // 50
                                                              // 51
                                                              // 52
                                                              // 53
                                                              // 54
                                                              // 55
                                                              // 56
                                                              // 57
                                                              // 58
                                                              // 59
struct A1 : public ConceptA< A1 >                             // 60
{                                                             // 61
  void meth_impl()                                            // 62
  {                                                           // 63
    std::cout << "A1::meth_impl()" << std::endl;            // 64
  }                                                           // 65
};                                                            // 66
                                                              // 67
                                                              // 68
                                                              // 69
// main                                                       // 70
                                                              // 71
                                                              // 72
int main()                                                    // 73
{                                                             // 74
  A1 a;                                                       // 75
  foo( a );                                                   // 76
}                                                             // 77


/*

bug: a model does not define a ‘‘nested type name’’ whereas its concept
requires it

error message:

proposal-err2.cc: In instantiation of ‘ConceptA<A1>’:
proposal-err2.cc:61:   instantiated from here
proposal-err2.cc:22: no type named ‘TypeB’ in ‘struct ConceptA_traits<A1>’

proposal-err2.cc: In function ‘void foo<A1>(ConceptA<A1> &)’:
proposal-err2.cc:76:   instantiated from here
proposal-err2.cc:37: no type named ‘B’ in ‘struct ConceptA<A1>’

proposal-err2.cc:38: ‘b’ undeclared (first use this function)
proposal-err2.cc:38: (Each undeclared identifier is reported only once
```

```
proposal-err2.cc:38: for each function it appears in.)
*/
```

A.6 `overloading.cc`

```
/*
 *  file overloading.cc
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>



// concepts' definition


template< class Model1 >
struct Concept1_traits {};

template< class Model1 >
struct Concept1
{
  void meth1() { self().meth1_impl(); }
protected:
  Model1& self()
   { return static_cast< Model1& >( *this ); }
};


template< class Model2 >
struct Concept2_traits {};

template< class Model2 >
struct Concept2
{
  void meth2() { self().meth2_impl(); }
protected:
  Model2& self()
   { return static_cast< Model2& >( *this ); }
};


template< class Model3 >
struct Concept3_traits {};

template< class Model3 >
struct Concept3
{
  void meth3() { self().meth3_impl(); }
protected:
  Model3& self()
   { return static_cast< Model3& >( *this ); }
};



// generic procedures


template< class Model1, class Model2 >
void foo( Concept1< Model1 >& arg1, Concept2< Model2 >& arg2 )
{
  std::cout << "foo( Concept1, Concept2 ):" << std::endl;
  arg1.meth1();
```

```
  arg2.meth2();
}


template< class Model2, class Model1 >
void foo( Concept2< Model2 >& arg1, Concept1< Model1 >& arg2 )
{
  std::cout << "foo( Concept2, Concept1 ):" << std::endl;
  arg1.meth2();
  arg2.meth1();
}


template< class Model1, class Model3 >
void foo( Concept1< Model1 >& arg1, Concept3< Model3 >& arg2 )
{
  std::cout << "foo( Concept1, Concept3 ):" << std::endl;
  arg1.meth1();
  arg2.meth3();
}



// models


struct M1 : public Concept1< M1 >
{
  void meth1_impl()
  {
    std::cout << "M1::meth1_impl()" << std::endl;
  }
};


struct M2 : public Concept2< M2 >
{
  void meth2_impl()
  {
    std::cout << "M2::meth2_impl()" << std::endl;
  }
};


struct M3 : public Concept3< M3 >
{
  void meth3_impl()
  {
    std::cout << "M3::meth3_impl()" << std::endl;
  }
};



// main


int main()
{
  M1 m1;
  M2 m2;
  M3 m3;

  foo( m1, m2 );
  std::cout << std::endl;
```

```
  foo( m2, m1 );
  std::cout << std::endl;

  foo( m1, m3 );
}
```

A.7 `multicompliance.cc`

```
/*
 *  file multicompliance.cc
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>



// Concept1


template< class Model1 >
struct Concept1_traits {};


template< class Model1 >
struct Concept1
{
  typedef typename Concept1_traits< Model1 >::TypeB  B;
  void meth() { self().meth_impl(); }
protected:
  Model1& self()
   { return static_cast< Model1& >( *this ); }
};



// Concept2


template< class Model2 >
struct Concept2_traits {};


template< class Model2 >
struct Concept2
{
  typedef typename Concept2_traits< Model2 >::TypeB  B;
  typedef typename Concept2_traits< Model2 >::TypeD  D;
  void meth() { self().meth_impl(); }
  void meth2() { self().meth2_impl(); }
protected:
  Model2& self()
   { return static_cast< Model2& >( *this ); }
};



// procedures


template< class Model1 >
void foo( Concept1< Model1 >& data )
{
  std::cout << "foo( Concept1 ):" << std::endl;
  typename Concept1< Model1 >::B*  b;
  b = 0;
  data.meth();
}
```

```
template< class Model2 >
void bar( Concept2< Model2 >& data )
{
  std::cout << "bar( Concept2 ):" << std::endl;
  typename Concept2< Model2 >::B*  b;
  b = 0;
  typename Concept2< Model2 >::D*  d;
  d = 0;
  data.meth();
  data.meth2();
}


// some classes


struct ClassB {};
struct ClassD {};


// M is a model of both Concept1 and Concept2


struct M;


template<>
struct Concept1_traits< M >
{
  typedef ClassB TypeB;
};


template<>
struct Concept2_traits< M >
{
  typedef ClassB TypeB;
  typedef ClassD TypeD;
};


struct M : public Concept1< M >, public Concept2< M >
{
  void meth_impl()
  {
    std::cout << "M::meth_impl()" << std::endl;
  }
  void meth2_impl()
  {
    std::cout << "M::meth2_impl()" << std::endl;
  }
};


// main


int main()
{
  M m;
```

```
  foo( m );
  std::cout << std::endl;

  bar( m );
}
```

A.8 `refinement.cc`

```
/*
 *  file refinement.cc
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>



// ConceptBase1


template< class ModelBase1 >
struct ConceptBase1_traits {};


template< class ModelBase1 >
struct ConceptBase1
{
  typedef typename ConceptBase1_traits< ModelBase1 >::TypeB  B;
  void meth() { this->self().meth_impl(); }
  void meth1() { this->self().meth1_impl(); }
protected:
  ModelBase1& self()
   { return static_cast< ModelBase1& >( *this ); }
};



// ConceptBase2


template< class ModelBase2 >
struct ConceptBase2_traits {};


template< class ModelBase2 >
struct ConceptBase2
{
  typedef typename ConceptBase2_traits< ModelBase2 >::TypeB  B;
  typedef typename ConceptBase2_traits< ModelBase2 >::TypeD  D;
  void meth() { this->self().meth_impl(); }
  void meth2() { this->self().meth2_impl(); }
protected:
  ModelBase2& self()
   { return static_cast< ModelBase2& >( *this ); }
};



// ConceptR refines both ConceptBase1 and ConceptBase2


template< class ModelR >
struct ConceptR_traits {};


template< class ModelR >
struct ConceptR : public ConceptBase1< ModelR >,
                  public ConceptBase2< ModelR >
```

```
{
  typedef typename ConceptR_traits< ModelR >::TypeB  B; // fix ambiguity
  typedef typename ConceptR_traits< ModelR >::TypeE  E;
  void meth() { this->self().meth_impl(); } // fix ambiguity
  void methR() { this->self().methR_impl(); }
protected:
  ModelR& self()
   { return static_cast< ModelR& >( *this ); }
};



// generic procedures


template< class ModelBase1 >
void foo( ConceptBase1< ModelBase1 >& data )
{
  std::cout << "foo( ConceptBase1 ):" << std::endl;
  typename ConceptBase1< ModelBase1 >::B*  b;
  b = 0;
  data.meth();
  data.meth1();
}


template< class ModelR >
void bar( ConceptR< ModelR >& data )
{
  std::cout << "bar( ConceptR ):" << std::endl;
  typename ConceptR< ModelR >::B*  b;
  b = 0;
  typename ConceptR< ModelR >::D*  d;
  d = 0;
  typename ConceptR< ModelR >::E*  e;
  e = 0;
  data.meth();
  data.meth1();
  data.meth2();
  data.methR();
}


template< class ModelR >
void base( ConceptR< ModelR >& data )
{
  std::cout << "base( ConceptR ):" << std::endl;
  foo( data );
}



// some classes


struct ClassB {};
struct ClassD {};
struct ClassE {};



// M is a model of ConceptR


struct M;
```

```
template<>
struct ConceptBase1_traits< M >
{
  typedef ClassB TypeB;
};

template<>
struct ConceptBase2_traits< M >
{
  typedef ClassB TypeB;
  typedef ClassD TypeD;
};

template<>
struct ConceptR_traits< M > : public ConceptBase1_traits< M >,
                              public ConceptBase2_traits< M >
{
  typedef ClassB TypeB; // fix ambiguity
  typedef ClassE TypeE;
};


struct M : public ConceptR< M >
{
  void meth_impl()
  {
    std::cout << "M::meth_impl()" << std::endl;
  }
  void meth1_impl()
  {
    std::cout << "M::meth1_impl()" << std::endl;
  }

  void meth2_impl()
  {
    std::cout << "M::meth2_impl()" << std::endl;
  }

  void methR_impl()
  {
    std::cout << "M::methR_impl()" << std::endl;
  }
};



// main


int main()
{
  M m;

  foo( m );
  std::cout << std::endl;

  bar( m );
  std::cout << std::endl;

  base( m );
}
```

A.9 `diamond.cc`

```
/*
 *  file diamond.cc
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>



// ConceptTop


template< class ModelTop >
struct ConceptTop_traits {};


template< class ModelTop >
struct ConceptTop
{
  typedef typename ConceptTop_traits< ModelTop >::TypeB  B; // declared only once => no ambiguity
  void meth() { this->self().meth_impl(); }
protected:
  ModelTop& self()
   { return static_cast< ModelTop& >( *this ); }
};



// ConceptMiddle1


template< class ModelMiddle1 >
struct ConceptMiddle1_traits {};


template< class ModelMiddle1 >
struct ConceptMiddle1 : public ConceptTop< ModelMiddle1 >
{
};



// ConceptMiddle2


template< class ModelMiddle2 >
struct ConceptMiddle2_traits {};


template< class ModelMiddle2 >
struct ConceptMiddle2 : public ConceptTop< ModelMiddle2 >
{
};



// ConceptBottom


template< class ModelBottom >
```

```
struct ConceptBottom_traits {};


template< class ModelBottom >
struct ConceptBottom : public ConceptMiddle1< ModelBottom >,
                       public ConceptMiddle2< ModelBottom >
{
  void meth() { this->self().meth_impl(); } // fix ambiguity
protected:
  ModelBottom& self()
   { return static_cast< ModelBottom& >( *this ); }
};



// generic procedures


template< class ModelBottom >
void foo( ConceptBottom< ModelBottom >& data )
{
  std::cout << "foo( ConceptBottom ):" << std::endl;
  typename ConceptBottom< ModelBottom >::B*  b;
  b = 0;
  data.meth();
}



// a class


struct ClassB {};



// M is a model of every concepts


struct M;


template<>
struct ConceptTop_traits< M >
{
  typedef ClassB TypeB;
};


template<>
struct ConceptMiddle1_traits< M > : public ConceptTop_traits< M >
{
};

template<>
struct ConceptMiddle2_traits< M > : public ConceptTop_traits< M >
{
};

template<>
struct ConceptBottom_traits< M > : public ConceptMiddle1_traits< M >,
                                   public ConceptMiddle2_traits< M >
{
};
```

```
struct M : public ConceptBottom< M >
{
  void meth_impl()
  {
    std::cout << "M::meth_impl()" << std::endl;
  }
};


// main


int main()
{
  M m;
  foo( m );
}
```

A.10 Big Sample of Explicit Concept Use

A.10.1 `c_aggregate.hh`

```
/*
 *  file c_aggregate.hh
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#ifndef C_AGGREGATE_HH
#define C_AGGREGATE_HH


template< typename M >
struct c_aggregate_traits {};


template< typename M >
struct c_aggregate
{
  typedef  typename c_aggregate_traits<M>::value_type     value_type;
  typedef  typename c_aggregate_traits<M>::site_type      site_type;
  typedef  typename c_aggregate_traits<M>::iterator_type  iterator_type;

  value_type& operator[]( const site_type& s )
    {
      return self().operator[]( s );
    }

  const value_type& operator[]( const site_type& s ) const
    {
      return self().operator[]( s );
    }

  iterator_type create_iterator() const
    {
      return self().create_iterator();
    }

protected:
  M& self() { return static_cast<M&>( *this ); }
  const M& self() const { return static_cast<const M&>( *this ); }
};


#endif // C_AGGREGATE_HH
```

A.10.2 `c_iterator.hh`

```
/*
 *  file c_iterator.hh
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */
```

```
#ifndef C_ITERATOR_HH
#define C_ITERATOR_HH


template< typename M >
struct c_iterator_traits {};


template< typename M >
struct c_iterator
{
  typedef  typename c_iterator_traits<M>::site_type  site_type;

  void first()
    {
      return self().first();
    }
  void is_ok() const
    {
      return self().is_ok();
    }
  void next()
    {
      return self().next();
    }
  operator site_type()
    {
      return site_type( self() );
    }

protected:
  M& self() { return static_cast<M&>( *this ); }
  const M& self() const { return static_cast<const M&>( *this ); }
};


#endif // C_ITERATOR_HH
```

A.10.3 `c_predicate.hh`

```
/*
 *  file c_predicate.hh
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#ifndef C_PREDICATE_HH
#define C_PREDICATE_HH


template< typename M >
struct c_predicate_traits {};


template< typename M >
struct c_predicate
{
  typedef  typename c_predicate_traits<M>::site_type  site_type;

  bool operator[]( const site_type& s ) const
    {
```

```
      return self().operator[]( s );
    }

protected:
  M& self() { return static_cast<M&>( *this ); }
  const M& self() const { return static_cast<const M&>( *this ); }
};


#endif // C_PREDICATE_HH
```

A.10.4 eq.hh

```
/*
 *  file eq.hh
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#ifndef EQ
#define EQ

#include "c_predicate.hh"
#include "c_aggregate.hh"


template< typename A > class eq;


template< typename A >
struct c_predicate_traits< eq<A> >
{
  typedef  typename c_aggregate_traits<A>::site_type  site_type;
};


template< typename A >
class eq : public c_predicate< eq<A> >
{
public:
  eq( c_aggregate<A>& agg,
      typename c_aggregate_traits<A>::value_type value ) :
    _agg( agg ),
    _value( value )
    {
    }
  eq( const eq<A>& other ) :
    _agg( other._agg ),
    _value( other._value )
    {
    }
  bool operator[]( const typename c_predicate_traits< eq<A> >::site_type& s ) const
    {
      return _agg[s] == _value;
    }
private:
  c_aggregate<A>& _agg;
  typename c_aggregate_traits<A>::value_type _value;
};
```

```
template< typename A >
eq<A> make_eq( c_aggregate<A>& agg,
               typename c_aggregate_traits<A>::value_type value )
{
  return eq<A>( agg, value );
}


#endif // EQ
```

A.10.5 `image2d.hh`

```
/*
 *  file image2d.hh
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#ifndef IMAGE2D_HH
#define IMAGE2D_HH


// concepts

#include "c_aggregate.hh"
#include "c_predicate.hh"
#include "c_iterator.hh"



// site


struct point2d
{
  point2d()
    {
      irow = 0;
      icol = 0;
    }
  point2d( int irow_, int icol_ )
    {
      irow = irow_;
      icol = icol_;
    }
  point2d( const point2d& p )
    {
      irow = p.irow;
      icol = p.icol;
    }
  int irow;
  int icol;
};



// iterator


class iterator2d;
```

```
template<>
struct c_iterator_traits< iterator2d >
{
  typedef  point2d  site_type;
};


class iterator2d : public c_iterator< iterator2d >
{
public:
  iterator2d( int nrows, int ncols )
    {
      _nrows = nrows;
      _ncols = ncols;
    }
  void first()
    {
      _p.irow = 0;
      _p.icol = 0;
    }
  bool is_ok() const
    {
      return _p.irow < _nrows;
    }
  void next()
    {
      ++_p.icol;
      if ( _p.icol == _ncols )
        {
          ++_p.irow;
          _p.icol = 0;
        }
    }
  operator point2d()
    {
      return _p;
    }
private:
  point2d _p;
  int _nrows;
  int _ncols;
};



// image2d<T>


template< typename T > class image2d;


template< typename T >
struct c_aggregate_traits< image2d<T> >
{
  typedef  T           value_type;
  typedef  point2d     site_type;
  typedef  iterator2d  iterator_type;
};


template< typename T >
class image2d : public c_aggregate< image2d<T> >
{
public:
```

```
  image2d( int nrows, int ncols )
    {
      _nrows = nrows;
      _ncols = ncols;
      _data = new T*[ nrows ];
      for ( int irow = 0; irow < _nrows; ++irow )
        _data[irow] = new T[ ncols ];
    }
  ~image2d()
    {
      for ( int irow = 0; irow < _nrows; ++irow )
        delete[] _data[irow];
      delete[] _data;
    }

  T& operator[]( const point2d& p )
    {
      return _data[p.irow][p.icol];
    }

  const T& operator[]( const point2d& p ) const
    {
      return _data[p.irow][p.icol];
    }

  iterator2d create_iterator() const
    {
      return iterator2d( _nrows, _ncols );
    }

  void println()
    {
      for ( int irow = 0; irow < _nrows; ++irow )
        {
          for ( int icol = 0; icol < _ncols; ++icol )
            cout << _data[irow][icol] << ' ';
          cout << endl;
        }
      cout << endl;
    }

protected:
  int _nrows;
  int _ncols;
  T** _data;
};


// specialization


template<> class image2d<bool>;


template<>
struct c_predicate_traits< image2d<bool> >
{
  typedef  point2d  site_type;
};



template<>
class image2d<bool> : public c_aggregate< image2d<bool> >,
```

```
                      public c_predicate< image2d<bool> >
{
public:

  image2d( int nrows, int ncols )
    {
      _nrows = nrows;
      _ncols = ncols;
      _data = new bool*[ nrows ];
      for ( int irow = 0; irow < _nrows; ++irow )
        _data[irow] = new bool[ ncols ];
    }
  ~image2d()
    {
      for ( int irow = 0; irow < _nrows; ++irow )
        delete[] _data[irow];
      delete[] _data;
    }

  bool& operator[]( const point2d& p )
    {
      return _data[p.irow][p.icol];
    }

  const bool& operator[]( const point2d& p ) const
    {
      return _data[p.irow][p.icol];
    }

  iterator2d create_iterator() const
    {
      return iterator2d( _nrows, _ncols );
    }

  void println() const
    {
      for ( int irow = 0; irow < _nrows; ++irow )
        {
          for ( int icol = 0; icol < _ncols; ++icol )
            cout << _data[irow][icol] << ' ';
          cout << endl;
        }
      cout << endl;
    }

protected:
  int _nrows;
  int _ncols;
  bool** _data;
};


#endif // IMAGE2D_HH
```

A.10.6 `main.cc`

```
/*
 *  file main.cc
 *
 *  Copyright (C) 2000 EPITA-LRDE
 *  EPITA Research and Development Laboratory
```

```
 */

#include<iostream>
using namespace std;

#include "image2d.hh"
#include "eq.hh"



template< typename A >
void foo( c_aggregate<A>&                              input,
          typename c_aggregate_traits<A>::site_type    site,
          typename c_aggregate_traits<A>::value_type   value )
{
  input[site] = value;
}



template< typename A >
void bar( c_aggregate<A>&                              input,
          typename c_aggregate_traits<A>::value_type   value )
{
  typename c_aggregate_traits<A>::iterator_type i = input.create_iterator();
  for ( i.first(); i.is_ok(); i.next() )
    input[i] += value;
}



template< typename A, typename P >
void ful( c_aggregate<A>&                              input,
          typename c_aggregate_traits<A>::value_type   value,
          const c_predicate<P>&                        pred )
{
  typename c_aggregate_traits<A>::iterator_type i = input.create_iterator();
  for ( i.first(); i.is_ok(); i.next() )
    if ( pred[i] == true )
      input[i] += value;
}


int main()
{
  image2d<int> i(3,3);
  i.println();

  foo( i, point2d(1,1), 3 );
  i.println();

  bar( i, 5 );
  i.println();


  image2d<bool> pred(3,3);
  pred[ point2d(0,0) ] = true;
  pred[ point2d(1,1) ] = true;
  pred[ point2d(2,2) ] = true;
  pred.println();

  ful( i, 7, pred );
  i.println();

  ful( i, 7, make_eq( i, 5 ) );
  i.println();
```

```
}
```

A.10.7 `log`

```
0 0 0
0 0 0
0 0 0

0 0 0
0 3 0
0 0 0

5 5 5
5 8 5
5 5 5

1 0 0
0 1 0
0 0 1

12 5 5
5 15 5
5 5 12

12 12 12
12 15 12
12 12 12
```

B Code of Patterns

B.1 `gamma_iterator.hh`

```
/*
 *  file gamma_iterator.hh
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */



// fwd decl
template< typename T > class Iterator;



// --------------------------  Aggregate<T>


template< typename T >
class Aggregate
{
public:
  virtual Iterator<T>& CreateIterator() = 0;
};



// --------------------------  Iterator<T>



template< typename T >
class Iterator
{
public:
  virtual void First() = 0;
  virtual void Next() = 0;
  virtual bool IsDone() const = 0;
  virtual T& CurrentItem() = 0;
};



// --------------------------  ConcreteAggregate<T>



template< typename T >
class ConcreteAggregate : public Aggregate<T>
{
public:
  ConcreteAggregate();
  virtual Iterator<T>& CreateIterator();
  const T& value_at( unsigned index ) const;
  T& value_at( unsigned index );
  unsigned size() const;
private:
  T _value[10];
};
```

```
// -------------------------  ConcreteIterator<T>


template< typename T >
class ConcreteIterator : public Iterator<T>
{
public:
  ConcreteIterator( ConcreteAggregate<T>& aggregate );
  virtual void First();
  virtual void Next();
  virtual bool IsDone() const;
  virtual T& CurrentItem();
private:
  ConcreteAggregate<T>& _aggregate;
  unsigned _index;
};


// -------------------------  proc


template< typename T >
void add( Aggregate<T>& input, T value );
```

B.2 gamma_iterator.cc

```
/*
 *  file gamma_iterator.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <assert.h>
#include <iostream>
#include "gamma_iterator.hh"



// -------------------------  ConcreteAggregate<T>



template< typename T > inline
ConcreteAggregate<T>::ConcreteAggregate()
{
  for ( unsigned index = 0; index < size(); ++index )
    {
      _value[ index ] = T( index );
    }
}



template< typename T >
Iterator<T>& ConcreteAggregate<T>::CreateIterator()
{
  return *new ConcreteIterator<T>( *this );
}



template< typename T > inline
const T& ConcreteAggregate<T>::value_at( unsigned index ) const
{
  return _value[ index ];
}



template< typename T > inline
T& ConcreteAggregate<T>::value_at( unsigned index )
{
  assert( index < size() );
  return _value[ index ];
}



template< typename T > inline
unsigned ConcreteAggregate<T>::size() const
{
  return 10;
}
```

```
template< typename T > inline
std::ostream& operator<<( std::ostream& ostr, const ConcreteAggregate<T>& obj )
{
  for ( unsigned index = 0; index < obj.size(); ++index )
    {
      ostr << obj.value_at( index ) << ' ';
    }
  return ostr;
}



// -------------------------  ConcreteIterator<T>



template< typename T > inline
ConcreteIterator<T>::ConcreteIterator( ConcreteAggregate<T>& aggregate ) :
  _aggregate( aggregate )
{
  _index = _aggregate.size();
}



template< typename T >
void ConcreteIterator<T>::First()
{
  _index = 0;
}



template< typename T >
void ConcreteIterator<T>::Next()
{
  ++_index;
}



template< typename T >
bool ConcreteIterator<T>::IsDone() const
{
  return _index >= _aggregate.size();
}



template< typename T >
T& ConcreteIterator<T>::CurrentItem()
{
  return _aggregate.value_at( _index );
}



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

```
// -------------------------  main


int main()
{
  ConcreteAggregate<int> aggregate;
  std::cout << aggregate << std::endl;
  add( aggregate, 1 );
  std::cout << aggregate << std::endl;
}
```

B.3 generic_iterator.hh

```
/*
 *  file generic_iterator.hh
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */



// fwd decl
template< typename T > class ConcreteIterator;



// --------------------------  ConcreteAggregate<T>



template< typename T >
class ConcreteAggregate
{
public:
  typedef T value_type;
  typedef ConcreteIterator<T> iterator_type;
  ConcreteAggregate();
  ConcreteIterator<T>& CreateIterator();
  const T& value_at( unsigned index ) const;
  T& value_at( unsigned index );
  unsigned size() const;
private:
  T _value[10];
};



// --------------------------  ConcreteIterator<T>



template< typename T >
class ConcreteIterator
{
public:
  ConcreteIterator( ConcreteAggregate<T>& aggregate );
  void First();
  void Next();
  bool IsDone() const;
  T& CurrentItem();
private:
  ConcreteAggregate<T>& _aggregate;
  unsigned _index;
};



// --------------------------  proc



template< typename A >
void add( A& input, typename A::value_type value );
```

B.4 generic_iterator.cc

```
/*
 *  file generic_iterator.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <assert.h>
#include <iostream>
#include "generic_iterator.hh"



// -------------------------  ConcreteAggregate<T>



template< typename T > inline
ConcreteAggregate<T>::ConcreteAggregate()
{
  for ( unsigned index = 0; index < size(); ++index )
    {
      _value[ index ] = T( index );
    }
}



template< typename T > inline
ConcreteIterator<T>& ConcreteAggregate<T>::CreateIterator()
{
  return *new ConcreteIterator<T>( *this );
}



template< typename T > inline
const T& ConcreteAggregate<T>::value_at( unsigned index ) const
{
  return _value[ index ];
}



template< typename T > inline
T& ConcreteAggregate<T>::value_at( unsigned index )
{
  assert( index < size() );
  return _value[ index ];
}



template< typename T > inline
unsigned ConcreteAggregate<T>::size() const
{
  return 10;
}
```

```
template< typename T > inline
std::ostream& operator<<( std::ostream& ostr, const ConcreteAggregate<T>& obj )
{
  for ( unsigned index = 0; index < obj.size(); ++index )
    {
      ostr << obj.value_at( index ) << ' ';
    }
  return ostr;
}



// -------------------------  ConcreteIterator<T>



template< typename T > inline
ConcreteIterator<T>::ConcreteIterator( ConcreteAggregate<T>& aggregate ) :
  _aggregate( aggregate )
{
  _index = _aggregate.size();
}



template< typename T > inline
void ConcreteIterator<T>::First()
{
  _index = 0;
}



template< typename T > inline
void ConcreteIterator<T>::Next()
{
  ++_index;
}



template< typename T > inline
bool ConcreteIterator<T>::IsDone() const
{
  return _index >= _aggregate.size();
}



template< typename T > inline
T& ConcreteIterator<T>::CurrentItem()
{
  return _aggregate.value_at( _index );
}



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

```
// -------------------------  main


int main()
{
  ConcreteAggregate<int> aggregate;
  std::cout << aggregate << std::endl;
  add( aggregate, 1 );
  std::cout << aggregate << std::endl;
}
```

B.5 gamma_visitor.cc

```
/*
 *  file gamma_visitor.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>



// fwd decl
class ConcreteElementA;
class ConcreteElementB;



// -------------------------  Visitor



class Visitor
{
public:
  virtual void VisitConcreteElementA( ConcreteElementA& e ) = 0;
  virtual void VisitConcreteElementB( ConcreteElementB& e ) = 0;
};



// -------------------------  Element



class Element
{
public:
  virtual void Accept( Visitor& v ) = 0;
};



// -------------------------  ConcreteElementA



class ConcreteElementA : public Element
{
public:
  virtual void Accept( Visitor& v )
    {
      v.VisitConcreteElementA( *this );
    }
};



// -------------------------  ConcreteElementB



class ConcreteElementB : public Element
```

```
{
public:
  virtual void Accept( Visitor& v )
    {
      v.VisitConcreteElementB( *this );
    }
};


// -------------------------  ConcreteVisitor1


class ConcreteVisitor1 : public Visitor
{
public:
  virtual void VisitConcreteElementA( ConcreteElementA& e )
    {
      std::cout << "ConcreteVisitor1::VisitConcreteElementA( ConcreteElementA& )" << std::endl;
    }
  virtual void VisitConcreteElementB( ConcreteElementB& e )
    {
      std::cout << "ConcreteVisitor1::VisitConcreteElementB( ConcreteElementB& )" << std::endl;
    }
};


// -------------------------  main


int main()
{
  ConcreteElementA e;
  e.Accept( *new ConcreteVisitor1() );
}
```

B.6 generic_visitor.cc

```
/*
 *  file generic_visitor.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>


// -------------------------  SuperiorOf<I>


template< class I >
class SuperiorOf
{
protected:
  SuperiorOf()
    {
    }
  I& Self()
    {
      return static_cast<I&>(*this);
    }
};


// -------------------------  Element<I>


template< class I >
class Element : SuperiorOf<I>
{
public:
  template< class Visitor >
  void Accept( Visitor v )
    {
      v.Visit( Self() );
    }
};


// -------------------------  ConcreteElementA


class ConcreteElementA : public Element< ConcreteElementA >
{ // ...
};


// -------------------------  ConcreteElementB


class ConcreteElementB : public Element< ConcreteElementB >
```

```
{ // ...
};



// --------------------------  ConcreteVisitor1



// decl

class ConcreteVisitor1
{
public:
  template< class ConcreteElement >
  void Visit( ConcreteElement& e );
};



// impl

template<>
void ConcreteVisitor1::Visit( ConcreteElementA& e )
{
  std::cout << "ConcreteVisitor1::Visit( ConcreteElementA& )" << std::endl;
}

template<>
void ConcreteVisitor1::Visit( ConcreteElementB& e )
{
  std::cout << "ConcreteVisitor1::Visit( ConcreteElementB& )" << std::endl;
}



// --------------------------  main



int main()
{
  ConcreteElementA e;
  e.Accept( ConcreteVisitor1() );
}
```

B.7 gamma_factory.cc

```
/*
 *  file gamma_factory.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>


// -------------------------  ProductA?


class ProductA
{
public:
  virtual void echo() const = 0;
};


class ProductA1 : public ProductA
{
public:
  virtual void echo() const
    {
      std::cout << "ProductA1" << std::endl;
    }
};


class ProductA2 : public ProductA
{
public:
  virtual void echo() const
    {
      std::cout << "ProductA2" << std::endl;
    }
};


// -------------------------  ProductB?


class ProductB
{
public:
  virtual void echo() const = 0;
};


class ProductB1 : public ProductB
{
public:
  virtual void echo() const
    {
```

```
      std::cout << "ProductB1" << std::endl;
    }
};


class ProductB2 : public ProductB
{
public:
  virtual void echo() const
    {
      std::cout << "ProductB2" << std::endl;
    }
};


// -------------------------  AbstractFactory


class AbstractFactory
{
public:
  virtual ProductA& CreateProductA() = 0;
  virtual ProductB& CreateProductB() = 0;
};


// -------------------------  ConcreteFactory1


class ConcreteFactory1 : public AbstractFactory
{
public:
  virtual ProductA& CreateProductA()
    {
      return *new ProductA1();
    }
  virtual ProductB& CreateProductB()
    {
      return *new ProductB1();
    }
};


// -------------------------  ConcreteFactory2


class ConcreteFactory2 : public AbstractFactory
{
public:
  virtual ProductA& CreateProductA()
    {
      return *new ProductA2();
    }
  virtual ProductB& CreateProductB()
    {
      return *new ProductB2();
    }
};
```

```
// -------------------------  foo


void foo( AbstractFactory& factory )
{
  ProductA& a = factory.CreateProductA();
  a.echo();
}


// -------------------------  main


int main()
{
  ConcreteFactory2 factory;
  foo( factory );
}
```

B.8 generic_factory.cc

```
/*
 *  file generic_factory.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>


// -------------------------  ProductA?


class ProductA1
{
public:
  void echo() const
    {
      std::cout << "ProductA1" << std::endl;
    }
};


class ProductA2
{
public:
  void echo() const
    {
      std::cout << "ProductA2" << std::endl;
    }
};


// -------------------------  ProductB?


class ProductB1
{
public:
  void echo() const
    {
      std::cout << "ProductB1" << std::endl;
    }
};


class ProductB2
{
public:
  void echo() const
    {
      std::cout << "ProductB2" << std::endl;
    }
};
```

```
// -------------------------  Def_Product?<F>


template< class F >
struct Def_ProductA
{
  typedef  typename F::productA_type  type;
};


template< class F >
struct Def_ProductB
{
  typedef  typename F::productB_type  type;
};


// -------------------------  ConcreteFactory1


class ConcreteFactory1
{
public:
  typedef  ProductA1  productA_type;
  typedef  ProductB1  productB_type;
};


// -------------------------  ConcreteFactory2


class ConcreteFactory2
{
};


template<>
struct Def_ProductA< ConcreteFactory2 >
{
  typedef  ProductA2  type;
};


template<>
struct Def_ProductB< ConcreteFactory2 >
{
  typedef  ProductB2  type;
};


// -------------------------  foo<Factory>


template< class Factory >
void foo( Factory& factory )
```

```
{
  typename Def_ProductA< Factory >::type  a;
  a.echo();
}


// -------------------------  main


int main()
{
  ConcreteFactory2 factory;
  foo( factory );
}
```

B.9 gamma_template_method.cc

```
/*
 *  file gamma_template_method.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>


// -------------------------  AbstractClass


class AbstractClass
{
public:
  void TemplateMethod()
    {
      // ...
      PrimitiveOperation1();
      // ...
      PrimitiveOperation2();
      // ...
    }
  virtual void PrimitiveOperation1() = 0;
  virtual void PrimitiveOperation2() = 0;
};


// -------------------------  ConcreteClass


class ConcreteClass : public AbstractClass
{
public:
  virtual void PrimitiveOperation1()
    {
      std::cout << "ConcreteClass::PrimitiveOperation1" << std::endl;
    }
  virtual void PrimitiveOperation2()
    {
      std::cout << "ConcreteClass::PrimitiveOperation2" << std::endl;
    }
};


// -------------------------  main


int main()
{
  ConcreteClass obj;
  obj.TemplateMethod();
}
```

B.10 `generic_template_method.cc`

```
/*
 *  file generic_template_method.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>


// -------------------------  SuperiorOf<I>


template< class I >
class SuperiorOf
{
protected:
  SuperiorOf()
    {
    }
  I& Self()
    {
      return static_cast<I&>(*this);
    }
};


// -------------------------  AbstractClass<I>


template< class I >
class AbstractClass : public SuperiorOf<I>
{
public:
  void TemplateMethod()
    {
      // ...
      PrimitiveOperation1();
      // ...
      PrimitiveOperation2();
      // ...
    }
  void PrimitiveOperation1()
    {
      Self().PrimitiveOperation1_impl();
    }
  void PrimitiveOperation2()
    {
      Self().PrimitiveOperation2_impl();
    }
};


// -------------------------  ConcreteClass
```

```
class ConcreteClass : public AbstractClass< ConcreteClass >
{
public:
  void PrimitiveOperation1_impl()
    {
      std::cout << "ConcreteClass::PrimitiveOperation1" << std::endl;
    }
  void PrimitiveOperation2_impl()
    {
      std::cout << "ConcreteClass::PrimitiveOperation2" << std::endl;
    }
};


// -------------------------  main


int main()
{
  ConcreteClass obj;
  obj.TemplateMethod();
}
```

B.11 gamma_decorator.cc

```
/*
 *  file gamma_decorator.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>


// -------------------------  Component


class Component
{
public:
  virtual void Operation() = 0;
};


// -------------------------  ConcreteComponent


class ConcreteComponent : public Component
{
public:
  virtual void Operation()
    {
      std::cout << "ConcreteComponent::Operation" << std::endl;
    }
};


// -------------------------  Decorator


class Decorator : public Component
{
public:
  Decorator( Component& component ) :
    _component( component )
    {
    }
  virtual void Operation()
    {
      _component.Operation();
    }
protected:
  Component& _component;
};


// -------------------------  ConcreteDecoratorA
```

```
class ConcreteDecoratorA : public Decorator
{
public:
  ConcreteDecoratorA( Component& component ) :
    Decorator( component )
    {
    }
  virtual void Operation()
    {
      Decorator::Operation();
      addedState = 1;
    }
protected:
  int addedState;
};


// -------------------------  ConcreteDecoratorB


class ConcreteDecoratorB : public Decorator
{
public:
  ConcreteDecoratorB( Component& component ) :
    Decorator( component )
    {
    }
  virtual void Operation()
    {
      Decorator::Operation();
      AddedBehaviour();
    }
protected:
  void AddedBehaviour()
    {
      std::cout << "ConcreteDecoratorB::AddedBehaviour" << std::endl;
    }
};


// -------------------------  main


int main()
{
  ConcreteDecoratorB  cb( *new ConcreteComponent() );
  cb.Operation();
}
```

B.12 generic_decorator.cc

```
/*
 *  file generic_decorator.cc
 *
 *  Copyright (C) 1999 EPITA-LRDE
 *  EPITA Research and Development Laboratory
 */

#include <iostream>


// -------------------------  ConcreteComponent


class ConcreteComponent
{
public:
  void Operation()
    {
      std::cout << "ConcreteComponent::Operation" << std::endl;
    }
};


// -------------------------  ConcreteDecoratorA<C>


template< class C >
class ConcreteDecoratorA : public C
{
public:
  void Operation()
    {
      C::Operation();
      addedState = 1;
    }
protected:
  int addedState;
};


// -------------------------  ConcreteDecoratorB<C>


template< class C >
class ConcreteDecoratorB : public C
{
public:
  void Operation()
    {
      C::Operation();
      AddedBehaviour();
    }
protected:
  void AddedBehaviour()
    {
      std::cout << "ConcreteDecoratorB::AddedBehaviour" << std::endl;
```

```
    }
};


// -------------------------  main


int main()
{
  ConcreteDecoratorB< ConcreteComponent >  cb;
  cb.Operation();
}
```

References

[1] *International Standard of C++*, September 1998. ISO/IEC 14882:1998(E), section 14.8.2.1, item 3.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science Series. Springer-Verlag, 1996.

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

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

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

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[7] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[8] 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`

[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.

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

[11] Thierry Géraud and Alexandre Duret-Lutz. Generic programming redesign pattern. In *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000)*, Irsee, Germany, July 2000. Submitted.

[12] Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-François Mangin. Obtaining genericity for image processing and pattern recognition algorithms. In *Proceedings of the 15th International Conference on Pattern Recognition (ICPR'2000)*, Barcelona, Spain, September 2000. To appear.

[13] Thierry Géraud, Yoann Fabre, Dimitri Papadopoulos-Orfanos, and Jean-François Mangin. Vers une réutilisabilité totale des algorithmes de traitement d'images. In *17th Symposium on Signal and Image Processing (GRETSI'99)*, volume 2, pages 331–334, Vannes, France, September 1999. Available in English as a technical report at:
`http://www.lrde.epita.fr/publications`

[14] 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`

[15] Intermetrics, Inc. *Ada Reference Manual*, November 1994. Draft of ISO/IEC 8652:1995, Version 5.95.

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

[17] 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`

[18] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

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

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

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

[22] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software: Practice and Experience*, 24(7):632–642, July 1994.

[23] David R. Musser and Alexandre A. Stepanov. Generic programming projects and open problems. 1998.
`http://www.cs.rpi.edu/~musser/gp/pop/index.html`

[24] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for java. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, Paris, France, January 1997.

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

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

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

[28] Jeremy Siek. Template argument checking. OON mailing list, August 1999.

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

[30] Herb Sutter. *Exceptional C++*. C++ In-Depth Series. Addison Wesley, 2000.

[31] Herb Sutter. Guru of the Week #71: Inheritance traits?, August 2000. To appear (available from `http://dejanews.com/`).

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

[33] 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`