Chapter 1

Protected Access to Internal Representations

1.1 Introduction

High level data structures typically maintain integrity of an internal data representation, which they protect from the user. A minimal while complete interface of the data structure allows manipulations in the domain of valid representations. Additional operations might benefit from being allowed to access the internal data representation directly. An example are intermediate steps within an algorithm where the internal representation would be invalid. We present a general method to accomplish access in a safe manner, such that the high level data structures can guarantee validity after the possibly compromising algorithm has finished its work. An example are polyhedral surfaces in the Basic Library, where a construction process like for a file scanner could be performed more efficiently on the internal halfedge data structure than by using the high-level Euler operators of the polyhedron.

The solution provided here is inspired by the strategy pattern [GHJV95], though it serves a different intent, see Figure 1.1. The abstract base class *Modifier_ base*<*R*> declares a pure virtual member function *operator()* that accepts a single reference parameter of the internal representation type. The

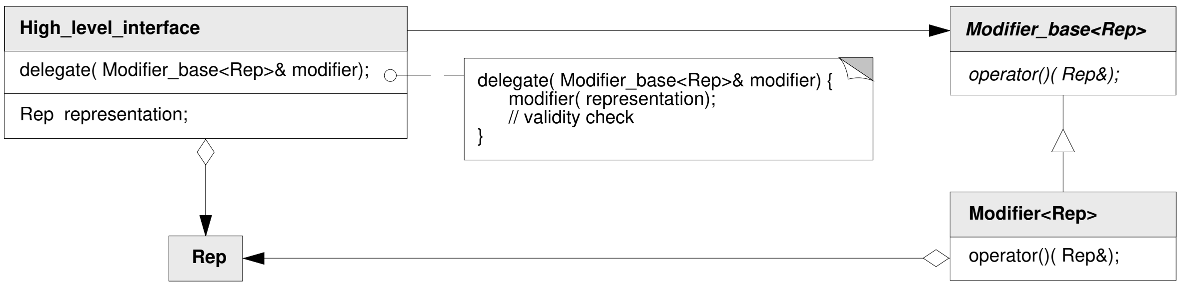

Figure 1.1: Class diagram for the modifier. It illustrates the safe access to an internal representation through an high-level interface.

member function *delegate()* of the high-level interface calls this *operator()* with its internal representation. An actual modifier implements this virtual function, thus gaining access to the internal representation. Once, the modifier has finished its work, the member function *delegate()* is back in control and can check the validity of the internal representation. Summarizing, a user can implement and apply arbitrary functions based on the internal representation and keeps the benefit if a protected high-level interface. User provided modifiers must in any case return a valid internal representation or the checker in the high-level interface is allowed (and supposed) to abort the program. The indirection via the virtual function invocation is negligible for operations that consists of more than a pointer update or integer addition.

1.2 Abstract Basis for Modifiers (*Modifier_ base<R>*)

Definition

Modifier_ base<R> is an abstract base class providing the interface for any modifier. A modifier is a function object derived from *Modifier_ base<R>* that implements the pure virtual member function *operator()*, which accepts a single reference parameter *R&* on which the modifier is allowed to work. *R* is the type of the internal representation that is to be modified.

#include <CGAL/Modifier_ base.h>

Types

typedef R	*Representation;*	the internal representation type.

Operations

virtual void	*modifier.operator()(R& rep)*	*Postcondition*: *rep* is a valid representation.

Example

The following fragment defines a class `A` with an internal representation `i` of type `int`. It provides a member function `delegate()`, which gives a modifier access to the internal variable and checks validity thereafter. The example modifier sets the internal variable to 42. The example function applies the modifier to an instance of class `A`.

```
class A {
    int i;  // protected internal representation
public:
    void delegate( CGAL::Modifier_base<int>& modifier) {
        modifier(i);
        CGAL_postcondition( i > 0);  // check validity
    }
};

struct Modifier : public CGAL::Modifier_base<int> {
    void operator()( int& rep) { rep = 42;}
```

```
};

void use_it() {
    A a;
    Modifier m;
    a.delegate(m);  // a.i == 42 and A has checked that A::i > 0.
}
```

Bibliography

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.