25

Roles of Classes

Some things better change ...
but fundamental themes
should revel in persistence.
– Stephen J. Gould

Kinds of classes — concrete types — abstract types — nodes — changing interfaces — object I/O — actions — interface classes — handles — use counts — application frameworks — advice — exercises.

25.1 Kinds of Classes [role.intro]

The C++ class is a programming language construct that serves a variety of design needs. In fact, I find that the solution to most knotty design problems involves the introduction of a new class to represent some notion that had been left implicit in the previous draft design (and maybe the elimination of other classes). The great variety of roles that a class can play leads to a variety of kinds of classes that are specialized to serve a particular need well. In this chapter, a few archetypical kinds of classes are described, together with their inherent strengths and weaknesses:

§25.2 Concrete types
§25.3 Abstract types
§25.4 Nodes
§25.5 Operations
§25.6 Interfaces
§25.7 Handles
§25.8 Application frameworks

These ‘‘kinds of classes’’ are design notions and not language constructs. The unattained, and probably unattainable, ideal is to have a minimal set of simple and orthogonal kinds of classes from which all well-behaved and useful classes could be constructed. It is important to note that each of

these kinds of classes has a place in design and none is inherently better than the others for all uses. Much confusion in discussions of design and programming comes from people trying to use only one or two kinds of classes exclusively. This is usually done in the name of simplicity, yet it leads to contorted and unnatural uses of the favored kinds of classes.

The description here emphasizes the pure forms of these kinds of classes. Naturally, hybrid forms can also be used. However, a hybrid ought to appear as the result of a design decision based on an evaluation of the engineering tradeoffs and not a result of some misguided attempt to avoid making decisions. ''Delaying decisions'' is too often a euphemism for ''avoiding thinking.'' Novice designers will usually do best by staying away from hybrids and also by following the style of an existing component with properties that resemble the desired properties for the new component. Only experienced programmers should attempt to write a general-purpose component or library, and every library designer should be ''condemned'' to use, document, and support his or her creation for some years. Also, please note §23.5.1.

25.2 Concrete Types [role.concrete]

Classes such as ***vector*** (§16.3), ***list*** (§17.2.2), ***Date*** (§10.3), and ***complex*** (§11.3, §22.5) are *concrete* in the sense that each is the representation of a relatively simple concept with all the operations essential for the support of that concept. Also, each has a one-to-one correspondence between its interface and an implementation and none are intended as a base for derivation. Typically, concrete types are not fitted into a hierarchy of related classes. Each concrete type can be understood in isolation with minimal reference to other classes. If a concrete type is implemented well, programs using it are comparable in size and speed to programs a user would write using a hand-crafted and specialized version of the concept. Similarly, if the implementation changes significantly the interface is usually modified to reflect the change. In all of this, a concrete type resembles a built-in type. Naturally, the built-in types are all concrete. User-defined concrete types, such as complex numbers, matrices, error messages, and symbolic references, often provide fundamental types for some application domain.

The exact nature of a class' interface determines what implementation changes are significant in this context; more abstract interfaces leave more scope for implementation changes but can compromise run-time efficiency. Furthermore, a good implementation does not depend on other classes more than absolutely necessary so that the class can be used without compile-time or run-time overheads caused by the accommodation of other ''similar'' classes in a program.

To sum up, a class providing a concrete type aims:

[1] to be a close match to a particular concept and implementation strategy;

[2] to provide run-time and space efficiency comparable to ''hand-crafted'' code through the use of inlining and of operations taking full advantage of the properties of the concept and its implementation;

[3] to have only minimal dependency on other classes; and

[4] to be comprehensible and usable in isolation.

The result is a tight binding between user code and implementation code. If the implementation changes in any way, user code will have to be recompiled because user code almost always contains calls of inline functions or local variables of a concrete type.

The name ''concrete type'' was chosen to contrast with the common term ''abstract type.'' The relationship between concrete and abstract types is discussed in §25.3.

Concrete types cannot directly express commonality. For example, ***list*** and ***vector*** provide similar sets of operations and can be used interchangeably by some template functions. However, there is no relationship between the types ***list<int>*** and ***vector<int>*** or between ***list<Shape*>*** and ***list<Circle*>*** (§13.6.3), even though ***we*** can discern their similarities.

For naively designed concrete types, this implies that code using them in similar ways will look dissimilar. For example, iterating through a ***List*** using a ***next()*** operation differs dramatically from iterating through a ***Vector*** using subscripting:

```
void my(List& sl)
{
    for (T* p = sl.first(); p; p = sl.next()) { // ''natural'' list iteration
        // my stuff
    }
    // ...
}

void your(Vector& v)
{
    for (int i = 0; i<v.size(); i++) {      // ''natural'' vector iteration
        // your stuff
    }
    // ...
}
```

The difference in iteration style is natural in the sense that a get-next-element operation is essential to the notion of a list (but not that common for a vector) and subscripting is essential to the notion of a vector (but not for a list). The availability of operations that are ''natural'' relative to a chosen implementation strategy is often crucial for efficiency and important for ease of writing the code.

The obvious snag is that the code for fundamentally similar operations, such as the previous two loops, can look dissimilar, and code that uses different concrete types for similar operations cannot be used interchangeably. In realistic examples, it takes significant thought to find similarities and significant redesign to provide ways of exploiting such similarities once found. The standard containers and algorithms are an example of a thorough rethinking that makes it possible to exploit similarities between concrete types without losing their efficiency and elegance benefits (§16.2).

To take a concrete type as an argument, a function must specify that exact concrete type as an argument type. There will be no inheritance relationships that can be used to make the argument declaration less specific. Consequently, an attempt to exploit similarities between concrete types will involve templates and generic programming as described in §3.8. When the standard library is used, iteration becomes:

```
template<class C> void ours(const C& c)
{
    for (C::const_iterator p = c.begin(); p!=c.end(); ++p) { // standard library iteration
        // ...
    }
}
```

The fundamental similarity between containers is exploited, and this in turn opens the possibility for further exploitation as done by the standard algorithms (Chapter 18).

To use a concrete type well, the user must understand its particular details. There are (typically) no general properties that hold for all concrete types in a library that can be relied on to save the user the bother of knowing the individual classes. This is the price of run-time compactness and efficiency. Sometimes that is a price well worth paying; sometimes it is not. It can also be the case that an individual concrete class is easier to understand and use than is a more general (abstract) class. This is often the case for classes that represent well-known data types such as arrays and lists.

Note, however, that the ideal is still to hide as much of the implementation as is feasible without seriously hurting performance. Inline functions can be a great win in this context. Exposing member variables by making them public or by providing set and get functions that allow the user to manipulate them directly is almost never a good idea (§24.4.2). Concrete types should still be types and not just bags of bits with a few functions added for convenience.

25.2.1 Reuse of Concrete Types [role.reuse]

Concrete types are rarely useful as bases for further derivation. Each concrete type aims at providing a clean and efficient representation of a single concept. A class that does that well is rarely a good candidate for the creation of different but related classes through public derivation. Such classes are more often useful as members or private base classes. There, they can be used effectively without having their interfaces and implementations mixed up with and compromised by those of the new classes. Consider deriving a new class from ***Date***:

```
class My_date : public Date {
    // ...
};
```

Is it ever valid for ***My_date*** to be used as a plain ***Date***? Well, that depends on what ***My_date*** is, but in my experience it is rare to find a concrete type that makes a good base class without modification.

A concrete type is ''reused'' unmodified in the same way as built-in types such as ***int*** are (§10.3.4). For example:

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
    // ...
};
```

This form of use (reuse?) is usually simple, effective, and efficient.

Maybe it was a mistake not to design ***Date*** to be easy to modify through derivation? It is sometimes asserted that *every* class should be open to modification by overriding and by access from derived class member functions. This view leads to a variant of ***Date*** along these lines:

```
class Date2 {
public:
        // public interface, consisting primarily of virtual functions
protected:
        // other implementation details (possibly including some representation)
private:
        // representation and other implementation details
};
```

To make writing overriding functions easy and efficient, the representation is declared ***protected***. This achieves the objective of making ***Date2*** arbitrarily malleable by derivation, yet keeping its user interface unchanged. However, there are costs:

[1] *Less efficient basic operations*. A C++ virtual function call is a fraction slower than an ordinary function call, virtual functions cannot be inlined as often as non-virtual functions, and a class with virtual functions typically incurs a one-word space overhead.

[2] *The need to use free store*. The aim of ***Date2*** is to allow objects of different classes derived from ***Date2*** to be used interchangeably. Because the sizes of these derived classes differ, the obvious thing to do is to allocate them on the free store and access them through pointers or references. Thus, the use of genuine local variables dramatically decreases.

[3] *Inconvenience to users*. To benefit from the polymorphism provided by the virtual functions, accesses to ***Date2***s must be through pointers or references.

[4] *Weaker encapsulation*. The virtual operations can be overridden and protected data can be manipulated from derived classes (§12.4.1.1).

Naturally, these costs are not always significant, and the behavior of a class defined in this way is often exactly what we want (§25.3, §25.4). However, for a simple concrete type, such as ***Date2***, the costs are unnecessary and can be significant.

Finally, a well-designed concrete type is often the ideal representation for a more malleable type. For example:

```
class Date3 {
public:
        // public interface, consisting primarily of virtual functions
private:
        Date d;
};
```

This is the way to fit concrete types (including built-in types) into a class hierarchy when that is needed. See also §25.10[1].

25.3 Abstract Types [role.abstract]

The simplest way of loosening the coupling between users of a class and its implementers and also between code that creates objects and code that uses such objects is to introduce an abstract class that represents the interface to a set of implementations of a common concept. Consider a naive ***Set***:

```
template<class T> class Set {
public:
        virtual void insert(T*) = 0;
        virtual void remove(T*) = 0;

        virtual int is_member(T*) = 0;

        virtual T* first() = 0;
        virtual T* next() = 0;

        virtual ~Set() { }
};
```

This defines an interface to a set with a built-in notion of iteration over its elements. The absence of a constructor and the presence of a virtual destructor is typical (§12.4.2). Several implementations are possible (§16.2.1). For example:

```
template<class T> class List_set : public Set<T>, private list<T> {
        // ...
};

template<class T> class Vector_set : public Set<T>, private vector<T> {
        // ...
};
```

The abstract class provides the common interface to the implementations. This means we can use a ***Set*** without knowing which kind of implementation is used. For example:

```
void f(Set<Plane*>& s)
{
        for (Plane** p = s.first(); p; p = s.next()) {
                // my stuff
        }
        // ...
}

List_set<Plane*> sl;
Vector_set<Plane*> v(100);

void g()
{
        f(sl);
        f(v);
}
```

For concrete types, we required a redesign of the implementation classes to express commonality and used a template to exploit it. Here, we must design a common interface (in this case ***Set***), but no commonality beyond the ability to implement the interface is required of the classes used for implementation.

Furthermore, users of ***Set*** need not know the declarations of ***List_set*** and ***Vector_set***, so users need not depend on these declarations and need not be recompiled or in any way changed if ***List_set*** or ***Vector_set*** changes or even if a new implementation of ***Set*** – say ***Tree_set*** – is

introduced. All dependencies are contained in functions that explicitly use a class derived from ***Set***. In particular, assuming the conventional use of header files the programmer writing ***f(Set&)*** needs only include ***Set.h*** and not ***List_set.h*** or ***Vector_set.h***. An ''implementation header'' is needed only where a ***List_set*** or a ***Vector_set***, respectively, is created. An implementation can be further insulated from the actual classes by introducing an abstract class that handles requests to create objects (''a factory;'' §12.4.4).

This separation of the interface from the implementations implies the absence of access to operations that are ''natural'' to a particular implementation but not general enough to be part of the interface. For example, because a ***Set*** doesn't have a notion of ordering we cannot support a subscripting operator in the ***Set*** interface even if we happen to be implementing a particular ***Set*** using an array. This implies a run-time cost due to missed hand optimizations. Furthermore, inlining typically becomes infeasible (except in a local context, when the compiler knows the real type), and all interesting operations on the interface become virtual function calls. As with concrete types, sometimes the cost of an abstract type is worth it; sometimes it is not. To sum up, an abstract type aims to:

[1] define a single concept in a way that allows several implementations of it to coexist in a program;
[2] provide reasonable run-time and space efficiency through the use of virtual functions;
[3] let each implementation have only minimal dependency on other classes; and
[4] be comprehensible in isolation.

Abstract types are not better than concrete types, just different. There are difficult and important tradeoffs for the user to make. The library provider can dodge the issue by providing both, thus leaving the choice to the user. The important thing is to be clear about to which world a class belongs. Limiting the generality of an abstract type in an attempt to compete in speed with a concrete type usually fails. It compromises the ability to use interchangeable implementations without significant recompilation after changes. Similarly, attempting to provide ''generality'' in concrete types to compete with the abstract type notion also usually fails. It compromises the efficiency and appropriateness of a simple class. The two notions can coexist – indeed, they *must* coexist because concrete classes provide the implementations for the abstract types – but they must not be muddled together.

Abstract types are often not intended to be bases for further derivation beyond their immediate implementation. Derivation is most often used just to supply implementation. However, a new interface can be constructed from an abstract class by deriving a more extensive abstract class from it. This new abstract class must then in turn be implemented through further derivation by a non-abstract class (§15.2.5).

Why didn't we derive ***List*** and ***Vector*** classes from ***Set*** in the first place to save the introduction of ***List_set*** and ***Vector_set*** classes? In other words, why have concrete types when we can have abstract types?

[1] *Efficiency*. We want to have concrete types such as ***vector*** and ***list*** without the overheads implied by decoupling the implementations from the interfaces (as implied by the abstract type style).
[2] *Reuse*. We need a mechanism to fit types designed ''elsewhere'' (such as ***vector*** and ***list***) into a new library or application by giving them a new interface (rather than rewriting them).

[3] *Multiple interfaces*. Using a single common base for a variety of classes leads to fat interfaces (§24.4.3). Often, it is better to provide a new interface to a class used for new purposes (such as a ***Set*** interface for a ***vector***) rather than modify its interface to serve multiple purposes.

Naturally, these points are related. They are discussed in some detail for the ***Ival_box*** example (§12.4.2, §15.2.5) and in the context of container design (§16.2). Using the ***Set*** base class would have resulted in a based-container solution relying on node classes (§25.4).

Section §25.7 describes a more flexible iterator in that the binding of the iterator to the implementation yielding the objects can be specified at the point of initialization and changed at run time.

25.4 Node Classes [role.node]

A class hierarchy is built with a view of derivation different from the interface/implementer view used for abstract types. Here, a class is viewed as a foundation on which to build. Even if it is an abstract class, it usually has some representation and provides some services for its derived classes. Examples of node classes are ***Polygon*** (§12.3), the initial ***Ival_slider*** (§12.4.1), and ***Satellite*** (§15.2).

Typically, a class in a hierarchy represents a general concept of which its derived classes can be seen as specializations. The typical class designed as an integral part of a hierarchy, a *node class*, relies on services from base classes to provide its own services. That is, it calls base class member functions. A typical node class provides not just an implementation of the interface specified by its base class (the way an implementation class does for an abstract type). It also adds new functions itself, thus providing a wider interface. Consider ***Car*** from the traffic-simulation example in §24.3.2:

```
class Car : public Vehicle {
public:
    Car(int passengers, Size_category size, int weight, int fc)
        : Vehicle(passengers,size,weight), fuel_capacity(fc) { /* ... */ }

    // override relevant virtual functions from Vehicle:

    void turn(Direction);
    // ...

    // add Car-specific functions:

    virtual add_fuel(int amount);   // a car needs fuel to run
    // ...
};
```

The important functions are the constructor through which the programmer specifies the basic properties that are relevant to the simulation and the (virtual) functions that allow the simulation routines to manipulate a ***Car*** without knowing its exact type. A ***Car*** might be created and used like this:

```
void user()
{
    // ...
    Car* p = new Car(3,economy,1500,60);
    drive(p,bs_home,MH);    // enter into simulated traffic pattern
    // ...
}
```

A node class usually needs constructors and often a nontrivial constructor. In this, node classes differ from abstract types, which rarely have constructors.

The operations on ***Car*** will typically use operations from the base class ***Vehicle*** in their implementations. In addition, the user of a ***Car*** relies on services from its base classes. For example, ***Vehicle*** provides the basic functions dealing with weight and size so that ***Car*** doesn't have to:

```
bool Bridge::can_cross(const Vehicle& r)
{
    if (max_weight<r.weight()) return false;
    // ...
}
```

This allows programmers to create new classes such as ***Car*** and ***Truck*** from a node class ***Vehicle*** by specifying and implementing only what needs to differ from ***Vehicle***. This is often referred to as ''programming by difference'' or ''programming by extension.''

Like many node classes, a ***Car*** is itself a good candidate for further derivation. For example, an ***Ambulance*** needs additional data and operations to deal with emergencies:

```
class Ambulance : public Car, public Emergency {
public:
    Ambulance();

    // override relevant Car virtual functions:

    void turn(Direction);
    // ...

    // override relevant Emergency virtual functions:

    virtual dispatch_to(const Location&);
    // ...

    // add Ambulance-specific functions:

    virtual int patient_capacity();  // number of stretchers
    // ...
};
```

To sum up, a node class

[1] relies on its base classes both for its implementation and for supplying services to its users;
[2] provides a wider interface (that is, an interface with more public member functions) to its users than do its base classes;
[3] relies primarily (but not necessarily exclusively) on virtual functions in its public interface;
[4] depends on all of its (direct and indirect) base classes;

[5] can be understood only in the context of its base classes;
[6] can be used as a base for further derivation; and
[7] can be used to create objects.

Not every node class will conform to all of points 1, 2, 6, and 7, but most do. A class that does not conform to point 6 resembles a concrete type and could be called a *concrete node class*. For example, a concrete node class can be used to implement an abstract class (§12.4.2) and variables of such a class can be allocated statically and on the stack. Such a class is sometimes called a *leaf class*. However, any code operating on a pointer or reference to a class with virtual functions must take into account the possibility of a further derived class (or assume without language support that further derivation hasn't happened). A class that does not conform to point 7 resembles an abstract type and could be called an *abstract node class*. Because of unfortunate traditions, many node classes have at least some ***protected*** members to provide a less restricted interface for derived classes (§12.4.1.1).

Point 4 implies that to compile a node class, a programmer must include the declarations of all of its direct and indirect base classes and all of the declarations that they, in turn, depend on. In this, a node class again provides a contrast to an abstract type. A user of an abstract type does not depend on the classes used to implement it and need not include them to compile.

25.4.1 Changing Interfaces [role.io]

By definition, a node class is part of a class hierarchy. Not every class in a hierarchy needs to offer the same interface. In particular, a derived class can provide more member functions, and a sibling class can provide a completely different set of functions. From a design perspective, ***dynamic_cast*** (§15.4) can be seen as a mechanism for asking an object if it provides a given interface.

As an example, consider a simple object I/O system. Users want to read objects from a stream, determine that they are of the expected types, and then use them. For example:

```
void user()
{
        // ... open file assumed to hold shapes, and attach ss as an istream for that file ...

        Io_obj* p = get_obj(ss); // read object from stream

        if (Shape* sp = dynamic_cast<Shape*>(p)) {
                sp->draw();     // use the Shape
                // ...
        }
        else {
                // oops: non-shape in Shape file
        }
}
```

The function ***user()*** deals with shapes exclusively through the abstract class ***Shape*** and can therefore use every kind of shape. The use of ***dynamic_cast*** is essential because the object I/O system can deal with many other kinds of objects and the user may accidentally have opened a file containing perfectly good objects of classes that the user has never heard of.

This object I/O system assumes that every object read or written is of a class derived from ***Io_obj***. Class ***Io_obj*** must be a polymorphic type to allow us to use ***dynamic_cast***. For example:

```
class Io_obj {
public:
    virtual Io_obj* clone() const =0;      // polymorphic
    virtual ~Io_obj() {}
};
```

The critical function in the object I/O system is ***get_obj()***, which reads data from an ***istream*** and creates class objects based on that data. Assume that the data representing an object on an input stream is prefixed by a string identifying the object's class. The job of ***get_obj()*** is to read that string prefix and call a function capable of creating and initializing an object of the right class. For example:

```
typedef Io_obj* (*PF)(istream&);          // pointer to function returning an Io_obj*

map<string,PF> io_map;                    // maps strings to creation functions

bool get_word(istream& is, string& s);    // read a word from is into s

Io_obj* get_obj(istream& s)
{
    string str;
    bool b = get_word(s,str);             // read initial word into str
    if (b == false) throw No_class();     // io format problem

    PF f = io_map[str];                   // lookup 'str' to get function
    if (f == 0) throw Unknown_class();    // no match for 'str'

    return f(s);                          // construct object from stream
}
```

The ***map*** called ***io_map*** holds pairs of name strings and functions that can construct objects of the class with that name.

We could define class ***Shape*** in the usual way, except for deriving it from ***Io_obj*** as required by ***user()***:

```
class Shape : public Io_obj {
    // ...
};
```

However, it would be more interesting (and in many cases more realistic) to use a defined ***Shape*** (§2.6.2) unchanged:

```
class Io_circle : public Circle, public Io_obj {
public:
    Io_circle* clone() const { return new Io_circle(*this); }  // using copy constructor
    Io_circle(istream&); // initialize from input stream
    static Io_obj* new_circle(istream& s) { return new Io_circle(s); }
    // ...
};
```

This is an example of how a class can be fitted into a hierarchy using an abstract class with less foresight than would have been required to build it as a node class in the first place (§12.4.2, §25.3).

The *Io_circle(istream&)* constructor initializes an object with data from its *istream* argument. The *new_circle()* function is the one put into the *io_map* to make the class known to the object I/O system. For example:

```
io_map["Io_circle"]=&Io_circle::new_circle;
```

Other shapes are constructed in the same way:

```
class Io_triangle : public Triangle, public Io_obj {
    // ...
};
```

If the provision of the object I/O scaffolding becomes tedious, a template might help:

```
template<class T> class Io : public T, public Io_obj {
public:
    Io* clone() const { return new Io(*this); }     // override Io_obj::clone()

    Io(istream&);                                   // initialize from input stream

    static Io* new_io(istream& s) { return new Io(s); }
    // ...
};
```

Given this, we can define *Io_circle*:

```
typedef Io<Circle> Io_circle;
```

We still need to define *Io<Circle>::Io(istream&)* explicitly, though, because it needs to know about the details of *Circle*.

The *Io* template is an example of a way to fit concrete types into a class hierarchy by providing a handle that is a node in that hierarchy. It derives from its template parameter to allow casting from *Io_obj*. Unfortunately, this precludes using *Io* for a built-in type:

```
typedef Io<Date> Io_date;       // wrap concrete type
typedef Io<int> Io_int;         // error: cannot derive from built-in type
```

This problem can be handled by providing a separate template for built-in types or by using a class representing a built-in type (§25.10[1]).

This simple object I/O system may not do everything anyone ever wanted, but it almost fits on a single page and the key mechanisms have many uses. In general, these techniques can be used to invoke a function based on a string supplied by a user and to manipulate objects of unknown type through interfaces discovered through run-time type identification.

25.5 Actions [role.action]

The simplest and most obvious way to specify an action in C++ is to write a function. However, if an action has to be delayed, has to be transmitted ''elsewhere'' before being performed, requires its own data, has to be combined with other actions (§25.10[18,19]), etc., then it often becomes attractive to provide the action in the form of a class that can execute the desired action and provide other services as well. A function object used with the standard algorithms is an obvious example

(§18.4), and so are the manipulators used with ***iostream***s (§21.4.6). In the former case, the actual action is performed by the application operator, and in the latter case, by the << or >> operators. In the case of ***Form*** (§21.4.6.3) and ***Matrix*** (§22.4.7), compositor classes were used to delay execution until sufficient information had been gathered for efficient execution.

A common form of action class is a simple class containing just one virtual function (typically called something like ''do_it''):

```
class Action {
public:
    virtual int do_it(int) = 0;
    virtual ~Action() { }
};
```

Given this, we can write code – say a menu – that can store actions for later execution without using pointers to functions, without knowing anything about the objects invoked, and without even knowing the name of the operation it invokes. For example:

```
class Write_file : public Action {
    File& f;
public:
    int do_it(int) { return f.write().succeed(); }
};

class Error_response : public Action {
    string message;
public:
    int do_it(int);
};

int Error_response::do_it(int)
{
    Response_box db(message.c_str(), "continue","cancel","retry");

    switch (db.get_response()) {
    case 0:
        return 0;
    case 1:
        abort();
    case 2:
        current_operation.redo();
        return 1;
    }
}

Action* actions[] = {
    new Write_file(f),
    new Error_response("you blew it again"),
    // ...
};
```

A user of ***Action*** can be completely insulated from any knowledge of derived classes such as ***Write_file*** and ***Error_response***.

This is a powerful technique that should be treated with some care by people with a background in functional decomposition. If too many classes start looking like *Action*, the overall design of the system may have deteriorated into something unduly functional.

Finally, a class can encode an operation for execution on a remote machine or for storage for future use (§25.10[18]).

25.6 Interface Classes [role.interface]

One of the most important kinds of classes is the humble and mostly overlooked interface class. An interface class doesn't do much – if it did, it wouldn't be an interface class. It simply adjusts the appearance of some service to local needs. Because it is impossible in principle to serve all needs equally well all the time, interface classes are essential to allow sharing without forcing all users into a common straitjacket.

The purest form of an interface doesn't even cause any code to be generated. Consider the *Vector* specialization from §13.5:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
    typedef Vector<void*> Base;

    Vector() : Base() {}
    Vector(int i) : Base(i) {}

    T*& operator[](int i) { return static_cast<T*&>(Base::operator[](i)); }

    // ...
};
```

This (partial) specialization turns the unsafe *Vector<void*>* into a much more useful family of type-safe vector classes. Inline functions are often essential for making interface classes affordable. In cases such as this, when an inline forwarding function does only type adjustment, there is no added overhead in time or space.

Naturally, an abstract base class representing an abstract type implemented by concrete types (§25.2) is a form of interface class, as are the handles from §25.7. However, here we will focus on classes that have no more specific function than adjusting an interface.

Consider the problem of merging two hierarchies using multiple inheritance. What can be done if there is a name clash, that is, two classes have used the same name for virtual functions performing completely different operations? For example, consider a Wild-West videogame in which user interactions are handled by a general window class:

```
class Window {
    // ...
    virtual void draw(); // display image
};

class Cowboy {
    // ...
    virtual void draw(); // pull gun from holster
};
```

```
class Cowboy_window : public Cowboy, public Window {
    // ...
};
```

A ***Cowboy_window*** represents the animation of a cowboy in the game and handles the user/player's interactions with the cowboy character. We would prefer to use multiple inheritance, rather than declaring either the ***Window*** or the ***Cowboy*** as members, because there will be many service functions defined for both ***Window***s and ***Cowboy***s. We would like to pass a ***Cowboy_window*** to such functions without special actions required by the programmer. However, this leads to a problem defining ***Cowboy_window*** versions of ***Cowboy*****::*****draw*****()** and ***Window*****::*****draw*****()**.

There can be only one function defined in ***Cowboy_window*** called ***draw*****()**. Yet because service functions manipulate ***Window***s and ***Cowboy***s without knowledge of ***Cowboy_window***s, ***Cowboy_window*** must override both ***Cowboy***'s ***draw*****()** and ***Window***'s ***draw*****()**. Overriding both functions by a single ***draw*****()** function would be wrong because, despite the common name, the ***draw*****()** functions are unrelated and cannot be redefined by a common function.

Finally, we would also like ***Cowboy_window*** to have distinct, unambiguous names for the inherited functions ***Cowboy*****::*****draw*****()** and ***Window*****::*****draw*****()**.

To solve this problem, we need to introduce an extra class for ***Cowboy*** and an extra class for ***Window***. These classes introduce the two new names for the ***draw*****()** functions and ensure that a call of the ***draw*****()** functions in ***Cowboy*** and ***Window*** calls the functions with the new names:

```
class CCowboy : public Cowboy {          // interface to Cowboy renaming draw()
public:
    virtual int cow_draw() = 0;
    void draw() { cow_draw(); }          // override Cowboy::draw()
};

class WWindow : public Window {          // interface to Window renaming draw()
public:
    virtual int win_draw() = 0;
    void draw() { win_draw(); }          // override Window::draw()
};
```

We can now compose a ***Cowboy_window*** from the interface classes ***CCowboy*** and ***WWindow*** and override ***cow_draw*****()** and ***win_draw*****()** with the desired effect:

```
class Cowboy_window : public CCowboy, public WWindow {
    // ...
    void cow_draw();
    void win_draw();
};
```

Note that this problem was serious only because the two ***draw*****()** functions have the same argument type. If they have different argument types, the usual overloading resolution rules will ensure that no problem manifests itself despite the unrelated functions having the same name.

For each use of an interface class, one could imagine a special-purpose language extension that could perform the desired adjustment a little bit more efficiently or a little more elegantly. However, each use of an interface class is infrequent and supporting them all with specialized language constructs would impose a prohibitive burden of complexity. In particular, name clashes arising from the merging of class hierarchies are not common (compared with how often a programmer will write a class) and tend to arise from the merging of hierarchies generated from dissimilar cultures – such as games and window systems. Merging such dissimilar hierarchies is not easy, and resolving name clashes will more often than not be the least of the programmer's problems. Other problems include dissimilar error handling, dissimilar initialization, and dissimilar memory-management strategies. The resolution of name clashes is discussed here because the technique of introducing an interface class with a forwarding function has many other applications. It can be used not only to change names, but also to change argument and return types, to introduce run-time checking, etc.

Because the forwarding functions `CCowboy::draw()` and `WWindow::draw()` are virtual functions, they cannot be optimized away by simple inlining. It is, however, possible for a compiler to recognize them as simple forwarding functions and then optimize them out of the call chains that go through them.

25.6.1 Adjusting Interfaces [role.range]

A major use of interface functions is to adjust an interface to match users' expectations better, thus moving code that would have been scattered throughout a user's code into an interface. For example, the standard `vector` is zero-based. Users who want ranges other than `0` to `size-1` must adjust their usage. For example:

```
void f()
{
    vector v<int>(10);          // range [0:9]

    // pretend v is in the range [1:10]:

    for (int i = 1; i<=10; i++) {
        v[i-1] = 7;      // remember to adjust index
        // ...
    }
}
```

A better way is to provide a `vector` with arbitrary bounds:

```
class Vector : public vector<int> {
    int lb;
public:
    Vector(int low, int high) : vector<int>(high-low+1) { lb=low; }

    int& operator[](int i) { return vector<int>::operator[](i-lb); }

    int low() { return lb; }
    int high() { return lb+size()-1; }
};
```

A ***Vector*** can be used like this:

```
void g()
{
    Vector v(1,10);          // range [1:10]

    for (int i = 1; i<=10; i++) {
        v[i] = 7;
        // ...
    }
}
```

This imposes no overhead compared to the previous example. Clearly, the ***Vector*** version is easier to read and write and is less error-prone.

Interface classes are usually rather small and (by definition) do rather little. However, they crop up wherever software written according to different traditions needs to cooperate because then there is a need to mediate between different conventions. For example, interface classes are often used to provide C++ interfaces to non-C++ code and to insulate application code from the details of libraries (to leave open the possibility of replacing the library with another).

Another important use of interface classes is to provide checked or restricted interfaces. For example, it is not uncommon to have integer variables that are supposed to have values in a given range only. This can be enforced (at run time) by a simple template:

```
template<int low, int high> class Range {
    int val;
public:
    class Error { }; // exception class

    Range(int i) { Assert<Error>(low<=i&&i<high); val = i; } // see §24.3.7.2
    Range operator=(int i) { return *this=Range(i); }

    operator int() { return val; }
    // ...
};

void f(Range<2,17>);
void g(Range<-10,10>);

void h(int x)
{
    Range<0,2001> i = x;     // might throw Range::Error
    int i1 = i;

    f(3);
    f(17);                   // throws Range::Error
    g(-7);
    g(100);                  // throws Range::Error
}
```

The ***Range*** template is easily extended to handle ranges of arbitrary scalar types (§25.10[7]).

An interface class that controls access to another class or adjusts its interface is sometimes called a ***wrapper***.

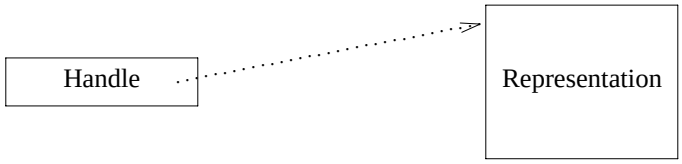
25.7 Handle Classes [role.handle]

An abstract type provides an effective separation between an interface and its implementations. However, as used in §25.3 the connection between an interface provided by an abstract type and its implementation provided by a concrete type is permanent. For example, it is not possible to rebind an abstract iterator from one source – say, a set – to another – say, a stream – once the original source becomes exhausted.

Furthermore, unless one manipulates an object implementing an abstract class through pointers or references, the benefits of virtual functions are lost. User code may become dependent on details of the implementation classes because an abstract type cannot be allocated statically or on the stack (including being accepted as a by-value argument) without its size being known. Using pointers and references implies that the burden of memory management falls on the user.

Another limitation of the abstract class approach is that a class object is of fixed size. Classes, however, are used to represent concepts that require varying amounts of storage to implement them.

A popular technique for dealing with these issues is to separate what is used as a single object into two parts: a handle providing the user interface and a representation holding all or most of the object's state. The connection between the handle and the representation is typically a pointer in the handle. Often, handles have a bit more data than the simple representation pointer, but not much more. This implies that the layout of a handle is typically stable even when the representation changes and also that handles are small enough to move around relatively freely so that pointers and references need not be used by the user.



The ***String*** from §11.12 is a simple example of a handle. The handle provides an interface to, access control for, and memory management for the representation. In this case, both the handle and the representation are concrete types, but the representation class is often an abstract class.

Consider the abstract type ***Set*** from §25.3. How could one provide a handle for it, and what benefits and cost would that involve? Given a set class, one might simply define a handle by overloading the `->` operator:

```
template<class T> class Set_handle {
    Set<T>* rep;
public:
    Set<T>* operator->() { return rep; }

    Set_handle(Set<T>* pp) : rep(pp) { }
};
```

This doesn't significantly affect the way ***Set***s are used; one simply passes ***Set_handle***s around instead of ***Set***&s or ***Set****s. For example:

```
void f(Set_handle<int> s)
{
    for (int* p = s->first(); p; p = s->next())
    {
        // ...
    }
}

void user()
{
    Set_handle<int> sl(new List_set<int>);
    Set_handle<int> v(new Vector_set<int>(100));

    f(sl);
    f(v);
}
```

Often, we want a handle to do more than just provide access. For example, if the ***Set*** class and the ***Set_handle*** class are designed together it is easy to do reference counting by including a use count in each ***Set***. In general, we do not want to design a handle together with what it is a handle to, so we will have to store any information that needs to be shared by a handle in a separate object. In other words, we would like to have non-intrusive handles in addition to the intrusive ones. For example, here is a handle that removes an object when its last handle goes away:

```
template<class X> class Handle {
    X* rep;
    int* pcount;
public:
    X* operator->() { return rep; }

    Handle(X* pp) : rep(pp), pcount(new int(1)) { }
    Handle(const Handle& r) : rep(r.rep), pcount(r.pcount) { (*pcount)++; }

    Handle& operator=(const Handle& r)
    {
        if (rep == r.rep) return *this;
        if (--(*pcount) == 0) {
            delete rep;
            delete pcount;
        }
        rep = r.rep;
        pcount = r.pcount;
        (*pcount)++;
        return *this;
    }

    ~Handle() { if (--(*pcount) == 0) { delete rep; delete pcount; } }

    // ...
};
```

Such a handle can be passed around freely. For example:

```
void f1(Handle<Set>);

Handle<Set> f2()
{
    Handle<Set> h(new List_set<int>);
    // ...
    return h;
}

void g()
{
    Handle<Set> hh = f2();
    f1(hh);
    // ...
}
```

Here, the set created in *f2()* will be deleted upon exit from *g()* – unless *f1()* held on to a copy; the programmer does not need to know.

Naturally, this convenience comes at a cost, but for many applications the cost of storing and maintaining the use count is acceptable.

Sometimes, it is useful to extract the representation pointer from a handle and use it directly. For example, this would be needed to pass an object to a function that does not know about handles. This works nicely provided the called function does not destroy the object passed to it or store a pointer to it for use after returning to its caller. An operation for rebinding a handle to a new representation can also be useful:

```
template<class X> class Handle {
    // ...

    X* get_rep() { return rep; }

    void bind(X* pp)
    {
        if (pp != rep) {
            if (--*pcount == 0) {
                delete rep;
                *pcount = 1;                // recycle pcount
            }
            else
                pcount = new int(1);        // new pcount
            rep = pp;
        }
    }
};
```

Note that derivation of new classes from ***Handle*** isn't particularly useful. It is a concrete type without virtual functions. The idea is to have one handle class for a family of classes defined by a base class. Derivation from this base class can be a powerful technique. It applies to node classes as well as to abstract types.

As written, ***Handle*** doesn't deal with inheritance. To get a class that acts like a genuine use-counted pointer, ***Handle*** needs to be combined with ***Ptr*** from §13.6.3.1 (see §25.10[2]).

A handle that provides an interface that is close to identical to the class for which it is a handle is often called a *proxy*. This is particularly common for handles that refer to an object on a remote machine.

25.7.1 Operations in Handles [role.handle.op]

Overloading `->` enables a handle to gain control and do some work on each access to an object. For example, one could collect statistics about the number of uses of the object accessed through a handle:

```
template <class T> class Xhandle {
    T* rep;
    int no_of_accesses;
public:
    T* operator->() { no_of_accesses++; return rep; }

    // ...
};
```

Handles for which work needs to be done both before *and* after access require more elaborate programming. For example, one might want a set with locking while an insertion or a removal is being done. Essentially, the representation class' interface needs to be replicated in the handle class:

```
template<class T> class Set_controller {
    Set<T>* rep;
    Lock lock;
    // ...
public:
    void insert(T* p) { Lock_ptr x(lock); rep->insert(p); }  // see §14.4.1
    void remove(T* p) { Lock_ptr x(lock); rep->remove(p); }

    int is_member(T* p) { return rep->is_member(p); }

    T get_first() { T* p = rep->first(); return p ? *p : T(); }
    T get_next() { T* p = rep->next(); return p ? *p : T(); }

    T first() { Lock_ptr x(lock); T tmp = *rep->first(); return tmp; }
    T next() { Lock_ptr x(lock); T tmp = *rep->next(); return tmp; }

    // ...
};
```

Providing these forwarding functions is tedious (and therefore somewhat error-prone), although it is neither difficult nor costly in run time.

Note that only some of the ***set*** functions required locking. In my experience, it is typical that a class needing pre- and post-actions requires them for only some member functions. In the case of locking, locking on all operations – as is done for monitors in some systems – leads to excess locking and sometimes causes a noticeable decrease in concurrency.

An advantage of the elaborate definition of all operations on the handle over the overloading of `->` style of handles is that it is possible to derive from class ***Set_controller***. Unfortunately, some of the benefits of being a handle are compromised if data members are added in the derived class. In particular, the amount of code shared (in the handled class) decreases compared to the amount of code written in each handle.

25.8 Application Frameworks [role.framework]

Components built out of the kinds of classes described in §25.2–§25.7 support design and reuse of code by supplying building blocks and ways of combining them; the application builder designs a framework into which these common building blocks are fitted. An alternative, and sometimes more ambitious, approach to the support of design and reuse is to provide code that establishes a common framework into which the application builder fits application-specific code as building blocks. Such an approach is often called an *application framework*. The classes establishing such a framework often have such fat interfaces that they are hardly types in the traditional sense. They approximate the ideal of being complete applications, except that they don't do anything. The specific actions are supplied by the application programmer.

As an example, consider a filter, that is, a program that reads an input stream, (maybe) performs some actions based on that input, (maybe) produces an output stream, and (maybe) produces a final result. A naive framework for such programs would provide a set of operations that an application programmer might supply:

```
class Filter {
public:
    class Retry {
    public:
        virtual const char* message() { return 0; }
    };

    virtual void start() { }
    virtual int read() = 0;
    virtual void write() { }
    virtual void compute() { }
    virtual int result() = 0;

    virtual int retry(Retry& m) { cerr << m.message() << '\n'; return 2; }

    virtual ~Filter() { }
};
```

Functions that a derived class must supply are declared pure virtual; other functions are simply defined to do nothing.

The framework also provides a main loop and a rudimentary error-handling mechanism:

```
int main_loop(Filter* p)
{
    for(;;) {
        try {
            p->start();
            while (p->read()) {
                p->compute();
                p->write();
            }
            return p->result();
        }
        catch (Filter::Retry& m) {
            if (int i = p->retry(m)) return i;
        }
        catch (...) {
            cerr << "Fatal filter error\n";
            return 1;
        }
    }
}
```

Finally, I could write my program like this:

```
class My_filter : public Filter {
    istream& is;
    ostream& os;
    int nchar;
public:
    int read() { char c; is.get(c); return is.good(); }
    void compute() { nchar++; }
    int result() { os << nchar << " characters read\n"; return 0; }

    My_filter(istream& ii, ostream& oo) : is(ii), os(oo), nchar(0) { }
};
```

and activate it like this:

```
int main()
{
    My_filter f(cin,cout);
    return main_loop(&f);
}
```

Naturally, for a framework to be of significant use, it must provide more structure and many more services than this simple example does. In particular, a framework is typically a hierarchy of node classes. Having the application programmer supply leaf classes in a deeply nested hierarchy allows commonality between applications and reuse of services provided by such a hierarchy. A framework will also be supported by a library that provides classes that are useful for the application programmer when specifying the action classes.

25.9 Advice [role.advice]

[1] Make conscious decisions about how a class is to be used (both as a designer and as a user); §25.1.
[2] Be aware of the tradeoffs involved among the different kinds of classes; §25.1.
[3] Use concrete types to represent simple independent concepts; §25.2.
[4] Use concrete types to represent concepts where close-to-optimal efficiency is essential; §25.2.
[5] Don't derive from a concrete class; §25.2.
[6] Use abstract classes to represent interfaces where the representation of objects might change; §25.3.
[7] Use abstract classes to represent interfaces where different representations of objects need to coexist; §25.3.
[8] Use abstract classes to represent new interfaces to existing types; §25.3.
[9] Use node classes where similar concepts share significant implementation details; §25.4.
[10] Use node classes to incrementally augment an implementation; §25.4.
[11] Use Run-time Type Identification to obtain interfaces from an object; §25.4.1.
[12] Use classes to represent actions with associated state; §25.5.
[13] Use classes to represent actions that need to be stored, transmitted, or delayed; §25.5.
[14] Use interface classes to adapt a class for a new kind of use (without modifying the class); §25.6.
[15] Use interface classes to add checking; §25.6.1.
[16] Use handles to avoid direct use of pointers and references; §25.7.
[17] Use handles to manage shared representations; §25.7.
[18] Use an application framework where an application domain allows for the control structure to be predefined; §25.8.

25.10 Exercises [role.exercises]

1. (∗1) The ***Io*** template from §25.4.1 does not work for built-in types. Modify it so that it does.
2. (∗1.5) The ***Handle*** template from §25.7 does not reflect inheritance relationships of the classes for which it is a handle. Modify it so that it does. That is, you should be able to assign a ***Handle<Circle*>*** to a ***Handle<Shape*>*** but not the other way around.
3. (∗2.5) Given a ***String*** class, define another string class using it as the representation and providing its operations as virtual functions. Compare the performance of the two classes. Try to find a meaningful class that is best implemented by publicly deriving from the string with virtual functions.
4. (∗4) Study two widely used libraries. Classify the library classes in terms of concrete types, abstract types, node classes, handle classes, and interface classes. Are abstract node classes and concrete node classes used? Is there a more appropriate classification for the classes in these libraries? Are fat interfaces used? What facilities – if any – are provided for run-time type information? What is the memory-management strategy?
5. (∗2) Use the ***Filter*** framework (§25.8) to implement a program that removes adjacent duplicate words from an input stream but otherwise copies the input to output.
6. (∗2) Use the ***Filter*** framework to implement a program that counts the frequency of words on

an input stream and produces a list of (word,count) pairs in frequency order as output.

7. (∗1.5) Write a ***Range*** template that takes both the range and the element type as template parameters.
8. (∗1) Write a ***Range*** template that takes the range as constructor arguments.
9. (∗2) Write a simple string class that performs no error checking. Write another class that checks access to the first. Discuss the pros and cons of separating basic function and checking for errors.
10. (∗2.5) Implement the object I/O system from §25.4.1 for a few types, including at least integers, strings, and a class hierarchy of your choice.
11. (∗2.5) Define a class ***Storable*** as an abstract base class with virtual functions ***write_out***`()` and ***read_in***`()`. For simplicity, assume that a character string is sufficient to specify a permanent storage location. Use class ***Storable*** to provide a facility for writing objects of classes derived from ***Storable*** to disk, and for reading such objects from disk. Test it with a couple of classes of your own choice.
12. (∗4) Define a base class ***Persistent*** with operations ***save***`()` and ***no_save***`()` that control whether an object is written to permanent storage by a destructor. In addition to ***save***`()` and ***no_save***`()`, what operations could ***Persistent*** usefully provide? Test class ***Persistent*** with a couple of classes of your own choice. Is ***Persistent*** a node class, a concrete type, or an abstract type? Why?
13. (∗3) Write a class ***Stack*** for which it is possible to change implementation at run time. Hint: ''Every problem is solved by yet another indirection.''
14. (∗3.5) Define a class ***Oper*** that holds an identifier of type ***Id*** (maybe a ***string*** or a C-style string) and an operation (a pointer to function or some function object). Define a class ***Cat_object*** that holds a list of ***Oper***s and a ***void****. Provide ***Cat_object*** with operations ***add_oper***`(`***Oper***`)`, which adds an ***Oper*** to the list; ***remove_oper***`(`***Id***`)`, which removes an ***Oper*** identified by ***Id*** from the list; and an ***operator***`()(`***Id***`,` ***arg***`)`, which invokes the ***Oper*** identified by ***Id***. Implement a stack of ***Cat***s by a ***Cat_object***. Write a small program to exercise these classes.
15. (∗3) Define a template ***Object*** based on class ***Cat_object***. Use ***Object*** to implement a stack of ***String***s. Write a small program to exercise this template.
16. (∗2.5) Define a variant of class ***Object*** called ***Class*** that ensures that objects with identical operations share a list of operations. Write a small program to exercise this template.
17. (∗2) Define a ***Stack*** template that provides a conventional and type-safe interface to a stack implemented by the ***Object*** template. Compare this stack to the stack classes found in the previous exercises. Write a small program to exercise this template.
18. (∗3) Write a class for representing operations to be shipped to another computer to execute there. Test it either by actually sending commands to another machine or by writing commands to a file and then executing the commands read from the file.
19. (∗2) Write a class for composing operations represented as function objects. Given two function objects f and g, ***Compose***`(`f`,`g`)` should make an object that can be invoked with an argument x suitable for g and return $f(g(x))$, provided the return value of $g()$ is an acceptable argument type for $f()$.