Item 38: Model "has-a" or "is-implemented-in-terms-of" through composition

Composition is the relationship between types that arises when objects of one type contain objects of another type. For example:

```
class Address { ... };              // where someone lives


class PhoneNumber { ... };


class Person {

public:

  ...


private:

  std::string name;                 // composed object

  Address address;                  // ditto

  PhoneNumber voiceNumber;          // ditto

  PhoneNumber faxNumber;            // ditto

};
```

In this example, `Person` objects are composed of `string`, `Address`, and `PhoneNumber` objects. Among programmers, the term *composition* has lots of synonyms. It's also known as *layering, containment, aggregation*, and *embedding*.

Item 32 explains that public inheritance means "is-a." Composition has a meaning, too. Actually, it has two meanings. Composition means either "has-a" or "is-implemented-in-terms-of." That's because you are dealing with two different domains in your software. Some objects in your programs correspond to things in the world you are modeling, e.g., people, vehicles, video frames, etc. Such objects are part of the *application domain*. Other objects are purely implementation artifacts, e.g., buffers, mutexes, search trees, etc. These kinds of objects correspond to your software's *implementation domain*. When composition occurs between objects in the application domain, it expresses a has-a relationship. When it occurs in the implementation domain, it expresses an is-implemented-in-terms-of relationship.

The `Person` class above demonstrates the has-a relationship. A `Person` object has a name, an address, and voice and fax telephone numbers. You wouldn't say that a person *is* a name or that a person *is* an address. You would say that a person *has* a name and *has* an address. Most people have little difficulty with this distinction, so confusion between the roles of is-a and has-a is relatively rare.

Somewhat more troublesome is the difference between is-a and is-implemented-in-terms-of. For example, suppose you need a template for classes representing fairly small sets of objects, i.e., collections without duplicates. Because reuse is a wonderful thing, your first instinct is to employ the standard library's `set` template. Why write a new template when you can use one that's already been written?

Unfortunately, `set` implementations typically incur an overhead of three pointers per element. This is because `set`s are usually implemented as balanced search trees, something that allows them to guarantee logarithmic-time lookups, insertions, and erasures. When speed is more important than space, this is a reasonable design, but it turns out that for

your application, space is more important than speed. The standard library's `set` thus offers the wrong trade-off for you. It seems you'll need to write your own template after all.

Still, reuse *is* a wonderful thing. Being the data structure maven you are, you know that of the many choices for implementing sets, one is to use linked lists. You also know that the standard C++ library has a `list` template, so you decide to (re)use it.

In particular, you decide to have your nascent `Set` template inherit from `list`. That is, `Set<T>` will inherit from `list<T>`. After all, in your implementation, a `Set` object will in fact *be* a `list` object. You thus declare your `Set` template like this:

```
template<typename T>                          // the wrong way to use list for Set

class Set: public std::list<T> { ... };
```

Everything may seem fine at this point, but in fact there is something quite wrong. As Item 32 explains, if `D` is-a `B`, everything true of `B` is also true of `D`. However, a `list` object may contain duplicates, so if the value 3051 is inserted into a `list<int>` twice, that list will contain two copies of 3051. In contrast, a `Set` may not contain duplicates, so if the value 3051 is inserted into a `Set<int>` twice, the set contains only one copy of the value. It is thus untrue that a `Set` is-a `list`, because some of the things that are true for `list` objects are not true for `Set` objects.

Because the relationship between these two classes isn't is-a, public inheritance is the wrong way to model that relationship. The right way is to realize that a `Set` object can be *implemented in terms of* a `list` object:

```
template<class T>                     // the right way to use list for Set

class Set {

public:

  bool member(const T& item) const;


  void insert(const T& item);

  void remove(const T& item);


  std::size_t size() const;


private:

  std::list<T> rep;                   // representation for Set data

};
```

`Set`'s member functions can lean heavily on functionality already offered by `list` and other parts of the standard library, so the implementation is straightforward, as long as you're familiar with the basics of programming with the STL:

```
template<typename T>

bool Set<T>::member(const T& item) const
```

```
{
  return std::find(rep.begin(), rep.end(), item) != rep.end();
}
template<typename T>
void Set<T>::insert(const T& item)
{
  if (!member(item)) rep.push_back(item);
}
template<typename T>
void Set<T>::remove(const T& item)
{
  typename std::list<T>::iterator it =                // see Item 42 for info on
    std::find(rep.begin(), rep.end(), item);          // "typename" here
  if (it != rep.end()) rep.erase(it);
}
template<typename T>
std::size_t Set<T>::size() const
{
  return rep.size();
}
```

These functions are simple enough that they make reasonable candidates for inlining, though I know you'd want to review the discussion in Item 30 before making any firm inlining decisions.

One can argue that `Set`'s interface would be more in accord with Item 18's admonition to design interfaces that are easy to use correctly and hard to use incorrectly if it followed the STL container conventions, but following those conventions here would require adding a lot of stuff to `Set` that would obscure the relationship between it and `list`. Since that relationship is the point of this Item, we'll trade STL compatibility for pedagogical clarity. Besides, nits about `Set`'s interface shouldn't overshadow what's indisputably right about `Set`: the relationship between it and `list`. That relationship is not is-a (though it initially looked like it might be), it's is-implemented-in-terms-of.

Things to Remember

- Composition has meanings completely different from that of public inheritance.
- In the application domain, composition means has-a. In the implementation domain, it means is-implemented-in-terms-of.