

The Separation of Interface and Implementation in C++

Bruce Martin[†]

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, California 94304
martin@hplabs.hp.com

abstract

A C++ *class* declaration combines the external interface of an object with the implementation of that interface. It is desirable to be able to write *client* code that depends only on the external interface of a C++ object and not on its implementation. Although C++ encapsulation can hide the implementation details of a class from client code, the client must refer to the class name and thus depends on the implied implementation as well as its interface.

In this paper, we review why separating an object's interface from its implementation is desirable and present a C++ programming style supporting a separate interface lattice and multiple implementation lattices. We describe minor language extensions that make the distinction between the interface lattice and implementation lattice apparent to the C++ programmer. Implementations are combined using standard C++ multiple inheritance. The operations of an interface are given by the union of operations of its contained interfaces. Variables and parameters are typed by interfaces. We describe how a separate interface lattice and multiple implementation lattices are realized in standard C++ code.

[†]Current address: Sun Microsystems, 2550 Garcia Ave,
Mountain View, CA 94043.
Bruce.E.Martin@eng.sun.com

1.0 Introduction

The *class* construct of the C++ language[10] mixes the notion of an interface to an object with the implementation of it. This paper demonstrates how to separate interface from implementation in C++.

An *interface* declares a set of operations. An object typed by that interface guarantees to support those operations. An *implementation*, on the other hand, defines *how* the object supports those operations, that is an implementation defines the representation of the object and a set of algorithms implementing the operations declared in the interface.

An interface may contain other interfaces and add new operations. An object whose type is given by the expanded interface supports the *union* of the operations of the contained interfaces and the new operations. An object supporting the expanded interface may be accessed by code expecting an object supporting one of the contained interfaces. If an interface is defined by combining multiple interfaces, an interface lattice results.

Similarly, an implementation may be provided in terms of other existing implementations.

That is, rather than providing implementations for all of the operations of an interface, an implementation may *inherit* some code and representation from other implementations. In the presence of multiple inheritance of implementations, an implementation lattice results.

If interface is separated from implementation, the interface lattice of an object need not be the same as the implementation lattice of the object. That is, the structure of the interface lattice need not be equivalent to the implementation lattice. Furthermore, there may be several different implementation lattices supporting the same interface.

1.1 Why separate interface from implementation?

Separating interface from implementation is desirable for achieving flexible, extensible, portable and modular software. If client code¹ depends only on the interface to an object and not on the object's implementation, a different implementation can be substituted and the client code continues to work, without change or recompilation. Furthermore, the client code continues to work on objects supporting an expanded interface.

Snyder describes in [9] how combining interface and implementation in a single class construct violates encapsulation. Snyder demonstrates that changing an object's implementation affects clients of that object when inheritance is used both for reusing code and for subtyping.

Our primary motivation for separating interface and implementation in C++ is to cleanly map C++ on to a system of distributed objects. In a distributed environment, allowing multiple implementation lattices for an interface is essential. Interfaces are *global* to the distributed environment, while implementations are

local. For a distributed program that crosses process, machine and administrative boundaries, maintaining a single implementation of an interface is difficult, if not impossible. However, as described in [6], it is feasible in an RPC based system to maintain a global space of interfaces.

1.2 C++ classes

A C++ *class* combines the interface of an object with the implementation of that interface. Although C++ encapsulation can hide the implementation details of a class from client code, the client code must refer to the class name and thus depends on the implied implementation.

Consider the C++ class, `stack`, in figure Figure 1.. The `stack` abstraction is implemented by an array, `elements`, and an integer, `top_of_stack`. Both the abstraction and the implementation are named, `stack`. Client code that declares variables and parameters of class `stack` identifies both the abstraction and the implementation.

```
class stack {
    int elements[MAX];
    int top_of_stack;
public:
    void push(int);
    int pop();
};
```

Figure 1. A C++ class, `stack`

A C++ *derived class* may add both to the interface of an object and to its implementation. That is, the derived class may add new public member functions and private members. The single class construct implies a single combined interface and implementation lattice.

Consider the derived class, `counted_stack` of figure Figure 2.. It expands the public interface by adding a member function, `size()` and it inherits the private members, `elements` and `top_of_stack`. It is impossible

1. We refer to code invoking an operation on some object as *client code*. The term *client* does not necessarily denote distributed computing.

to be a `counted_stack` without also containing the array, `elements`.

```
class counted_stack:
public stack {
    int no_items;
public:
    int size();
    void push(int);
    int pop();
};
```

Figure 2. A derived class, `counted_stack`

C++ 2.0 has added *pure virtual functions*, *multiple inheritance* and *virtual base classes* to the language. This paper shows how those constructs can be used to support a separate interface lattice and multiple implementation lattices.

1.3 Related work

Languages such as Ada and Modula 2 explicitly separate the interface to a program module from the implementation of it. These languages do not have mechanisms for inheriting implementations.

The Abel project[1] at Hewlett-Packard Laboratories has explored the role of interfaces in statically typed object-oriented programming languages. Abel interfaces are more flexible than those described here for C++. In particular, the return type of an operation may be specialized and parameters generalized in an extended interface. C++ requires them to be the same.

Several object-based distributed systems ([7], [2], [3], [8]) use or extend C++ as the programmer's interface to the distributed objects. In such systems, interfaces are not explicit but rather considered to be the public member function declarations of a C++ class. As such, interface and implementation lattices must have the exact structure at all nodes in the distributed system. This paper demonstrates how such C++ distributed systems can have an

explicit, global interface lattice and multiple, local implementation lattices.

2.0 Separation Model

This paper presents a model for C++ programs in which an interface lattice can be supported by different multiple implementation lattices. We describe the model in terms of some minor language extensions that have been implemented in a preprocessor producing standard C++ 2.0 code. The language extensions make the separation between interface lattice and implementation lattice apparent to the C++ programmer; they also make our description clearer. However, the model could be viewed as a C++ programming style and programmers could write the C++ we describe in section 3.0 directly. The language extensions enforce the style.

Throughout the paper, we use an example of a *bus stop*. The interface *BusStop* is given by combining the interface *PeopleQueue* with the interface *Port*.

2.1 Interfaces

Figure 3. gives the interface to a queue of people. The interface declares three operations: `enq` adds a person to the queue, `deq` removes and returns a person from the queue and `size` returns the number of people in the queue.

```
interface PeopleQueue {
    enq(person *);
    person *deq();
    int size();
};
```

Figure 3. The interface to a queue of people

The interface represents a contract between client code invoking an operation on an object meeting the `PeopleQueue` interface and code implementing the interface. The contract states that the object will support the three operations specified in the contract. It says

nothing, however, about the implementation of those three operations.

Similarly, figure Figure 4. gives the interface to a `Port`. A `Port` represents a place where a vehicle departs at some time for some destination. The `dest` operation returns a reference to an object meeting the `city` interface and the `departs` operation returns a reference to an object meeting the `time` interface.

```
interface Port{
    city *dest();
    time *departs();
};
```

Figure 4. The interface to a port

An operation declaration gives a name and the types of parameters and return values. Parameters and return values are either C++ primitive and aggregation types or they are references to objects supporting an interface. They are *not* typed by a C++ implementation class. For example, in the `PeopleQueue` interface of figure Figure 3., the `enq` operation takes as a parameter a reference to an object supporting the `person` interface, the `deq` operation returns an object supporting the `person` interface and the `size` operation returns an integer value.

An interface declares the *public* interface to an object; there are no `public`, `protected` or `private` labels, as there are in C++ classes.

Interfaces can be combined and expanded. Figure Figure 5. defines a `BusStop` interface in terms of a `PeopleQueue` and a `Port`. The declaration states that a `BusStop` is a `PeopleQueue` and it is a `Port`. An object meeting the `BusStop` interface supports all of the operations defined by the `PeopleQueue` and by the `Port`; it can be used in any context expecting either an object meeting the `PeopleQueue` interface or one meeting the `Port` interface. In addition, a `BusStop` supports the `covered` operation. (The `covered` operation is true if the bus stop is covered.)

```
interface BusStop :
    PeopleQueue, Port {
    boolean covered();
};
```

Figure 5. The interface to a bus stop

Figure Figure 6. presents a graphical representation of the interface lattice for a `BusStop`.

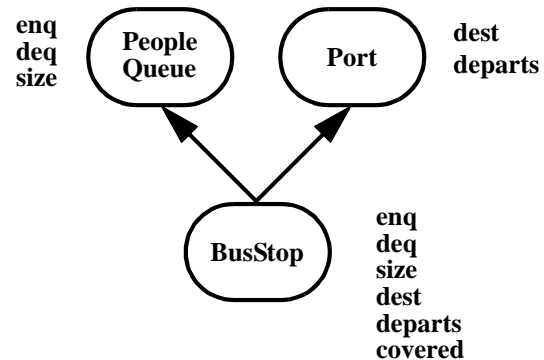


Figure 6. Interface lattice for a bus stop

The containment of interfaces is not as flexible as discussed in [1]. Operations are simply the declarations of member functions in the C++ sense. Refinement of parameter or result types is not supported. As in C++, an operation whose name matches another operation but whose parameters differ is considered to be overloaded.

Ambiguities cannot result when combining multiple interfaces; the operations of an interface are declarations, not definitions.

2.2 Implementations

Interfaces alone do not result in executing programs. Programmers must also provide *implementations* of an interface. Figure Figure 7. gives an implementation, named `linked_people`, of `PeopleQueue`. It uses a linked list to represent the queue. Figure

```

class linked_people
  implements PeopleQueue {
    recptr *head, *tail;
  public:
    linked_people() {
      head=tail=NULL;
    }
    enq(person *p);
    person *deq();
    int size();
};

```

Figure 7. A linked list implementation of a queue of people

Figure 8. gives an implementation, named `people_buffer`, of `PeopleQueue`. It uses an array to represent the queue.

Notice in figures Figure 7. and Figure 8. that a class is declared to be an implementation of an interface using the `implements` keyword. Multiple implementations of the same interface can exist in a single C++ program; `people_buffer` and `linked_people` both implement the `PeopleQueue` interface.

Classes provide algorithms implementing the operations declared in the interface and may declare state and local member functions. Local member functions are called in implementation code. Implementations may also define constructors and destructors for the implementation. A constructor is inherently

```

class people_buffer
  implements PeopleQueue {
    person **buf;
    int last;
  public:
    people_buffer(int sz) {
      last=0;
      buf = new person *[sz];
    }
    enq(person *p);
    person *deq();
    int size() {return last;}
};

```

Figure 8. An array implementation of a queue of people.

implementation dependent. For example, the constructor for the `people_buffer` in figure Figure 8. takes an integer argument indicating the upper bound on the size of the buffer, while the constructor for the `linked_people` takes no arguments.

Notice that the parameter types of operations are given as other interfaces, not implementations. Thus, it would be impossible to define an operation that took a `linked_people` as a parameter. Similarly, local and member variables are typed by interface, not implementation. By doing so, client code is independent of a particular implementation.

Implementations may *reuse* other implementations. For example, figure Figure 9. gives an implementation of `BusStop`, named

```

class muni_stop
  implements BusStop
  reuses public: linked_people {
    boolean shelter;
  public:
    muni_stop(boolean cov){
      shelter=cov;
    }
    city *dest();
    time *departs();
    boolean covered() {
      return shelter;
    }
};

```

Figure 9. Municipal bus stop implementation of the bus stop

`muni_stop`, suitable to represent municipal bus stops. It reuses the implementation of `linked_people`.

Figure Figure 10. gives a graphical representation of the `muni_stop` implementation.²

Figure Figure 11. gives a different implementation of `BusStop`, named `inter_city`. It

2. We use ovals to represent interfaces and rectangles to represent implementations in all graphics.

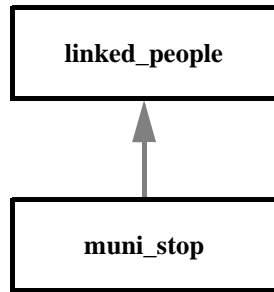


Figure 10. Graphical representation of the `muni_stop` implementation

reuses both the `people_buffer` and the `pair`³ implementations.

Figure Figure 12. gives a graphical representation of the `inter_city` implementation.

Notice that there are multiple implementation lattices in figures Figure 10. and Figure 12., that they have different structures and that the *implementation* lattice of figure Figure 10. is not structurally the same as the *interface* lattice of figure Figure 12.. Without the separation of interface and implementation this would not be possible.

Implementations reuse other implementations according to C++ inheritance semantics. A class declares the operations it will implement

```

class inter_city
  implements BusStop
  reuses public: pair,
    public: people_buffer {
  public:
    inter_city();
    boolean covered() {
      return true;
    }
};
  
```

Figure 11. Intercity implementation of the bus stop

3. The `pair` implementation of the `Port` interface is left to the reader as an exercise.

and inherits the ones it does not. Thus, for example, the `muni_stop` implementation of the `BusStop` interface given in figure Figure 9. declares and implements the `dest`, `departs` and `covered` operations but reuses the `enq`, `deq` and `size` functions from the `linked_people` class. On the other hand, the `inter_city` implementation declares and implements only the `covered` operation and reuses the others from `people_buffer` and `pair`.

Ambiguities are resolved in the C++ way, by the programmer. If a class inherits ambiguous member functions from multiple classes, the class must also declare the member function and provide an implementation of it. There is, of course, no ambiguity caused by interfaces -- the interface lattice is separate.

Programmers control the visibility of the declarations in the implementation lattice using `public`, `private` and `protected`. However, member functions that implement operations declared in the interface lattice are, by definition, `public`.

2.3 Object instantiation

Implementations are named when an object is instantiated by the `new` operator or allocated on the run time stack. However, to promote modularity and flexibility, this code should be isolated. Client code that refers to an object via variables typed by the object's implementation, rather than by one of its supported interfaces,

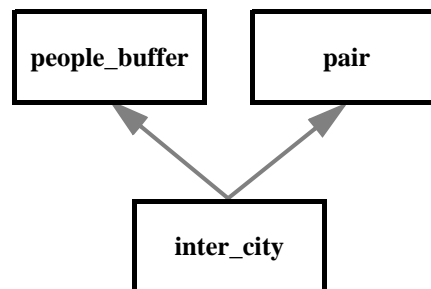


Figure 12. Graphical representation of the `inter_city` implementation

creates unnecessary dependencies on implementation; the client code can only be used with objects of that implementation.

Most code should refer to an object using variables typed by interface. For example, if an instance of the `muni_stop` implementation is created like this:

```
BusStop *bs = new muni_stop(false)
```

all further references to it will be typed by its interface, `BusStop`, not by its implementation.

Similarly, when an object is allocated on the run time stack, it should be referred to by an interface it supports. A `muni_stop` imple-

```
muni_stop ms(false);
inter_city ic;
simulate(BusStop*, BusStop *);
:
simulate(&ms, &ic);
:
```

Figure 13. Naming implementations when allocated on run time stack

mentation and a `inter_city` implementation are created on the run time stack in figure Figure 13.. The identification of the implementation should be isolated to these declarations. The `simulate` function is defined to operate on two objects supporting the `BusStop` interface, independent of implementation. The `simulate` function is a client of the objects. It cannot see anything about the implementations; the types of the parameters do not name implementations.

2.4 Design Goals

The design of the separation model was constrained by three practical requirements.

First, the separation model needed to be easily realized in C++. This meant that at some level, the separation model had to be a programming style. This constrained the model. For exam-

ple, the model allows an object of an extended interface to be accessed by code expecting an object of a contained interface but as shown in [1], the notion of a contained interface is stronger than it need be.

Next, the design of the separation model was also constrained by a desire to preserve the C++ inheritance model for reusing implementations. The rules for when to declare a member function in a class, the C++ approach towards ambiguities and the definitions of virtual and non-virtual base classes were preserved for implementations.

Finally, what the separation model adds should be simple. To use the separation model, the C++ programmer must learn

- that an extended interface supports the union of the operations of its contained interfaces,
- that variables and parameters should be typed by interfaces not by implementations,
- that code instantiating an object names an implementation and should be isolated
- and that all class member functions implementing operations declared in interfaces are public.

3.0 Realization in C++

We now discuss how to realize the separation model in C++ and some of the problems we encountered in realizing it. Although we describe this in terms of the behavior of our preprocessor for the language extensions given in section 2.0, it could be viewed as a programming style and the C++ programmer could write this code directly.

The presentation assumes some understanding of C++ implementation strategy, particularly for multiple inheritance. It is too detailed to repeat here; we refer the reader to [11].

3.1 Interfaces

An interface is translated to a C++ class of the same name containing only pure virtual functions. Pure virtual functions, those defined with the odd `=0`, indicate the C++ class will *not* provide an implementation of the function. A C++ class with a pure virtual function cannot be instantiated via the `new` operator or on the stack. These are the desired semantics for interfaces. Figure Figure 14. gives the `Port` interface of figure Figure 4. translated to standard C++.

```
class Port {
public:
    virtual city *dest()=0;
    virtual time *departs()=0;
};
```

Figure 14. C++ representation of the `Port` interface

Interfaces contain *only* pure virtual functions; C++ allows pure virtual functions to be declared in a class that also contains implementation. Doing so breaks the strict separation of interface and implementation.

3.2 Combining Interfaces

The separation model defined the operations of an extended interface to be the union of the operations in its contained interfaces and the added operations. Furthermore, the separation model allows an object supporting an interface to be accessed in a context expecting an object of one of the contained interfaces.

Combining interfaces by inheriting the C++ abstract classes representing the interfaces almost works. The *isa* relationship is provided by inheriting C++ abstract classes. However, C++ 2.0 does not allow pure virtual functions to be inherited and requires them to be redeclared in derived classes. When translating an interface declaration, the preprocessor generates the union of the pure virtual functions in the derived abstract classes. Figure Figure 15. gives the `BusStop` interface of figure Figure

```
class BusStop :
    public virtual PeopleQueue,
    public virtual Port {
public:
    virtual boolean covered()=0;
    virtual int enq(person *)=0;
    virtual person *deq()=0;
    virtual int size()=0;
    virtual city *dest()=0;
    virtual time *departs()=0;
};
```

Figure 15. C++ representation of the `BusStop` interface

5. translated to standard C++. Notice that the `BusStop` is declared to be a derived class of both `PeopleQueue` and `Port` but redeclares the pure virtual functions from both.

The C++ abstract classes representing interfaces are inherited as virtual base classes.

As discussed in [11], using non-virtual base classes may make sense for some implementations. It does not for interfaces. Virtual base classes are semantically closer to the separation model's notion of interface combination.

Using virtual base classes to represent interfaces results in an "inconvenience" for the programmer. Without virtual base classes, type casting is implemented as pointer arithmetic. Programmers are allowed to do unsafe type casting from a reference to a base class to a reference to a derived class because it requires a simple address calculation. However, C++ represents virtual base classes as a pointer in the derived class to the virtual base class. Casting from a derived class to a virtual base class is not a simple address calculation but instead follows the pointer to the virtual base class. Casting from a virtual base class to a derived class is not supported.

Some might consider the restriction on type-unsafe casting to be a benefit of the representing interfaces as virtual base classes! However, C++ programs often do need to cast from a base class to a derived class. Achieving this in

a type safe fashion requires associating type information at run time with objects such as proposed in [4]. Basically, implementations can return the addresses of the “interface part” being requested. One possible result, of course, is that the implementation does not support the requested interface and the cast fails.

3.3 Implementations

An implementation is represented as a C++ derived class of the interface it implements. An implementation is also a C++ derived class of the C++ classes representing the reused implementations.

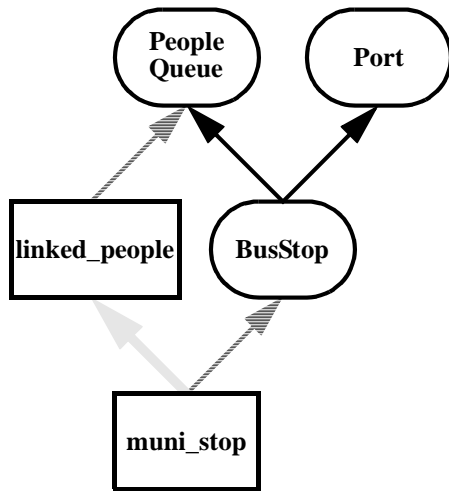


Figure 16. Resulting C++ class lattice for muni_stop

Figure Figure 16. graphically represents the resulting C++ class lattice for the muni_stop implementation. Similarly, figure Figure 17. graphically represents the resulting class lattice for the inter_city implementation. The ovals denote interfaces, the rectangles denote implementations, the solid arrows represent the *isa* relation, the dashed arrows represent the *implements* relation and the grey arrows represent the *reuses* relation. From a C++ point of view, these distinctions are irrelevant; the ovals and rectangles are all classes and the structure represents a multiple inheritance class lattice.

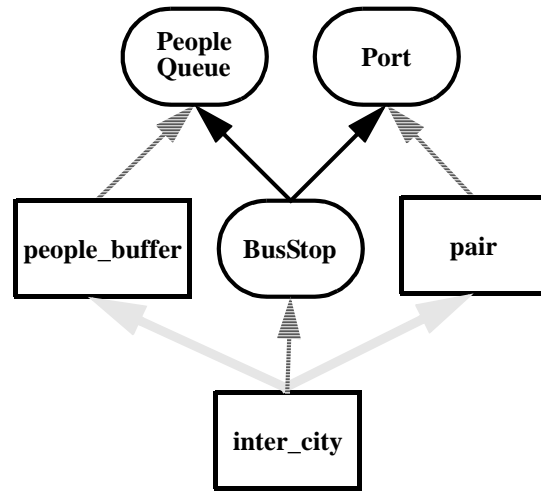


Figure 17. Resulting C++ class lattice for inter_city

3.4 Binding implementations to interfaces

If a class provides an implementation of an operation, the programmer declares and defines the member function for that class. The translation described above works.

On the other hand, if the implementation reuses other implementations, the obvious translation does not quite work. Consider the C++ representation of the inter_city implementation of figure Figure 11. to be the C++ code of figure Figure 18..

```
class inter_city :
    public virtual BusStop,
    public: pair,
    public: people_buffer {
public:
    inter_city();
    virtual boolean covered() {
        return true;
    }
};
```

Figure 18. Incorrect C++ for the inter_city implementation

C++ inheritance does not work directly; the class in figure Figure 18. inherits pure virtual functions and implemented functions. In C++ 2.0 inheriting a pure virtual function is not allowed. In C++ 2.1 it is allowed but the resulting class is still viewed as ambiguous.[5]

To overcome these problems, the preprocessor for the separation model generates explicit calls to the inherited functions. For example, the `inter_city` implementation given in figure Figure 11. reuses the `enq` member function from `people_buffer`. In the generated C++ code of figure Figure 19., an explicit call is generated to `people_buffer::enq`.

```
class inter_city :
    public virtual BusStop,
    public: pair,
    public: people_buffer {
public:
    inter_city();
    virtual boolean covered() {
        return true;
    }
    virtual int enq(person *p0 ){
        return
            people_buffer::enq(p0);
    }
    virtual person *deq() {
        return
            people_buffer::deq();
    }
    virtual int size() {
        return
            people_buffer::size();
    }
    virtual city *dest (); {
        return pair::dest();
    }
    virtual time *departs () {
        return pair::departs();
    }
};
```

Figure 19. Correct C++ code for the `inter_city` implementation

Of course, in order to preserve C++ inheritance for implementations, calls are not generated when the multiple inheritance of implementations is ambiguous. In that case, the program-

mer must declare and define a member function to resolve the ambiguity.

We note that the above binding discussion does not pertain to state or local member functions since they are not part of the abstract classes representing interfaces. C++ inheritance applies directly.

Finally, to tie all of this together, we present the generated C++ class lattice in appendix A for the `muni_stop` implementation. Without the language extensions, appendix A represents what the programmer would write instead of the interfaces in figures Figure 3. through Figure 5. and the implementations in figures Figure 7. and Figure 9..

3.5 Object Layout

When an object is defined by separate interface and implementation lattices, the C++ layout of the object in memory is analogous to the logical separation model described in this paper. The layout of an object contains an implementation part and an interface part. The implementation part of an object is *physically separate* from the interface part. Furthermore, for all objects meeting the same interface, the interface part is structurally identical; the differences in representation are found only in the implementation part of the object. This is as it must be for a compiler to generate client code without knowing anything about the implementation of an object.

Figure Figure 20. graphically depicts the layout of an object implemented by `muni_stop`. The implementation part is above the thick line and the interface part is below it. As a result of using virtual base classes to represent interfaces, the addresses of the various interfaces supported by the object are also stored as pointers in the implementation part. Type casting from the implementation to one of the supported interfaces by the object simply follows the pointer. The addresses of the contained interfaces are stored as pointers in the interface part. Type casting from an interface to one of

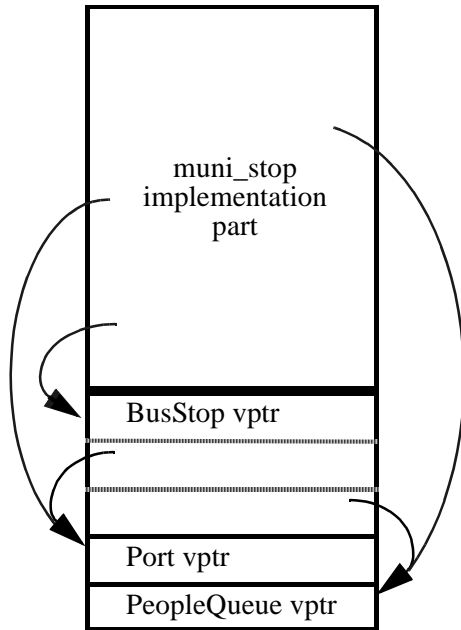


Figure 20. Physical layout of an object implemented by `muni_stop`

its contained interfaces simply follows the pointer. An `BusStop` object implemented by `inter_city` differs from figure Figure 20. in the implementation part only. The structure of the interface part is identical.

3.6 Performance

Characterizing the execution and storage cost of using a *programming style* is tricky. What do we compare a program written according to the style to? The use of the programming style may have influenced the programmer to define and implement a completely different set of abstractions to achieve the same task.

We can, however, characterize the performance of a specific transformation. Assume we mechanically transform an existing C++ class lattice into a separate interface lattice and a single implementation lattice such that the transformed code has the same behavior as the original code. The original class lattice, the interface lattice and the implementation lattice all have the same structure.

The interface lattice contains operations defined by the public member functions of the original class lattice. For non-function public members, we introduce operations of the same name.

The implementation lattice adds the non-public members of the original C++ class lattice. The non-function public members are implemented as simple inline functions that return their values.

We compare the differences in execution time and storage use.

3.6.1 Execution cost

The original and transformed code have the same execution cost. All virtual functions in the original class lattice remain virtual in the transformed code. Since the separation model requires that all operations are virtual functions, non-virtual member functions become virtual functions in the transformed code. However, to maintain the same behavior, all calls to non-virtual functions in the original code must be transformed to class qualified calls. Because of these transformations of function calls, the original calls to non-virtual functions remain direct function calls. Similarly, inline functions are still expanded at the point of the call.

3.6.2 Storage cost

Instances in the transformed code require more space than instances in the original C++ code.

The implementation part of an instance of a transformed class is increased by a pointer to each virtual base class representing the supported interfaces. (See the arrows from the implementation part of figure Figure 20.)

The instance also includes storage for the interface part. Each interface in the interface part stores a pointer to its virtual function table, a pointer for each of its contained interfaces and replicates its immediate contained interfaces.

4.0 Conclusions

In this paper we have reviewed the motivation for separating the interface to an object from the implementation of it. We have described how a separate interface lattice and multiple implementation lattices can be achieved in a C++ program. We have described some minor language extensions that make the separation model apparent to the C++ programmer. We have implemented a preprocessor for the language extensions.

We have described how the separation model is realized in C++. As such, the separation model could be viewed as a C++ programming style. However, C++ could support this style, without imposing it, more directly. In order to eliminate the awkward binding of implementations, described in section 3.4, pure virtual functions should be distinguished in C++ inheritance semantics. In particular, a derived class should be able to inherit a pure virtual function. Furthermore, no ambiguity should result if a pure virtual function and an implemented function of the same name are inherited; the implemented function should be inherited in favor of the pure virtual function declaration. These changes would make this a more palatable C++ programming style.

The separation model forces a programmer to create at least two names -- one for the interface and one for each implementation. The use of pure virtual functions in C++ does not. Requiring a separate name for the interface is what results in modular, flexible, distributed code.

Acknowledgments

I would like to thank Peter Canning, Mike Cannon, William Cook, Warren Harris, Walt Hill, Jussi Ketonen, Bob Shaw and Alan Snyder for commenting on the C++ separation model.

References

- [1] Peter Canning, William Cook, Walt Hill and Walter Olthoff. "Interfaces for Strongly-typed Object-oriented Programming", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 457--467, 1989. Also Technical Report STL-89-6, Hewlett-Packard Labs.
- [2] David Detlefs, Maurice Herlihy, Karen Kietzke and Jeannette Wing. "Avalon/C++: C++ Extensions for Transaction-based Programming", In *USENIX C++ Workshop*, 1987.
- [3] Yvon Gourhant and Marc Shapiro. "FOG/C++: A Fragmented Object Generator", In *Proceedings of the 1990 Usenix C++ Conference*, April 1990.
- [4] John Interrante and Mark Linton. "Runtime Access to Type Information in C++", In *Proceedings of the 1990 Usenix C++ Conference*, April 1990.
- [5] Stan Lipman. Electronic mail from Stan Lipman, ATT Bell Laboratories, April 1990.
- [6] Bruce Martin, Charles Bergan, Walter Burkhard and J.F. Paris. "Experience with PARPC", In *Proceedings of the 1989 Winter USENIX Technical Conference*. Usenix Association, 1989.
- [7] S. K. Shrivastava, G. N. Dixon, F. Hedayati, G. D. Parrington and S. M. Wheeler. "A Technical Overview of Arjuna: a System for Reliable Distributed Computing", Technical Report 262, University of Newcastle upon Tyne, July 1988.
- [8] Robert Seliger. "Extended C++", In *Proceedings of the 1990 Usenix C++ Conference*, April 1990.
- [9] Alan Snyder. "Encapsulation and Inheritance in Object-oriented Programming Languages", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. Association of Computing Machinery, 1986.
- [10] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986.
- [11] Bjarne Stroustrup. "Multiple inheritance for C++", In *Proceedings of the EUUG Spring 1987 Conference*, May 1987.

Appendix A: Generated C++ code for muni_stop:

```
// Interfaces:

class PeopleQueue {
public:
    virtual int enq(person *)=0;
    virtual person *deq()=0;
    virtual int size()=0;
};

class Port {
public:
    virtual city *dest()=0;
    virtual time *departs()=0;
};

class BusStop : public virtual PeopleQueue, public virtual Port {
public:
    virtual boolean covered()=0;
    virtual int enq(person *)=0;
    virtual person *deq()=0;
    virtual int size()=0;
    virtual city *dest()=0;
    virtual time *departs()=0;
};

// Implementations:

class linked_people : public virtual PeopleQueue {
    class recptr *head ,*tail;
public:
    linked_people () { head =tail =NULL ; }
    virtual int enq (person *);
    virtual person *deq ();
    virtual int size ();
};

class muni_stop : public virtual BusStop, public linked_people {
    boolean shelter ;
public:
    muni_stop (boolean cover ) { shelter =cover;}
    virtual int enq(person *p0) {
        return linked_people::enq(p0);
    }
    virtual person *deq() { return linked_people::deq(); }
    virtual int size() { return linked_people::size(); }
    virtual city *dest();
    virtual time *departs();
    virtual boolean covered() { return shelter; }
};
```

