

## Re: Abstract Data Types – Separating Interface from Implementation

*Source:* <http://coding.derkeiler.com/Archive/C/ CPP/comp.lang.cpp/2003-12/3098.html>

---

*From:* Jeff Schwab ([jeffplus\\_at\\_comcast.net](mailto:jeffplus_at_comcast.net))

*Date:* 12/24/03

Date: Wed, 24 Dec 2003 01:50:38 -0500

Anon Email wrote:

> *Thanks guys.*

>

>

>> *You seem to be confusing implementation code and client code.*

>

>

> *Yes, you are right – I am confused. I'm confused by the terms*

> *implementation code, client code and interface. And I'm unsure as to*

> *why abstract data types are called "abstract".*

>

> *1) Put simply, is client code non-class code?*

Those are unrelated issues. Here's how it usually works for me, when I'm working on my own. In reality, there are often several people involved in various steps of this process, e.g. step 1 might be done by a manager or committee, steps 2 and 3 might be done by me, and step 4 might be done by someone assigned to help me.

1) I decide I need a program to do something, e.g. provide some information or some service I can't easily get from my system already.

2) I write a basic, simplistic program to show the outline of what I want to do. When the program needs to do something complicated, I pretend I have a function or object that does the complicated bit. I decide how I'd like to be able to call the function, what the inputs and outputs should be. To make my skeletal program compile, I declare the functions and classes; for example, if I need to send data over a network, I might make up a function that looks like this:

```
void send_data_to_host( class Datum&, std::string const& host_name );
```

If I want to be able to do complicated things to the network connection, I might define a class that looks like this:

```
class Packet;
```

```
class Network_Connecton
{
public:
    enum Protocol { tcp, udp };

    Network_Connection(
        std::string const& host_name,
        Protocol const& protocol =tcp );

    ~Network_Connection( );

    void open(
        std::string const& host_name,
        Protocol const& protocol =tcp );

    void close( ) throw( );

    void send( Packet& );

    void receive( Packet& );

    // ... other operations ...
};
```

I don't define any of the functions yet, I just declare them and compile my program to object code (with my compiler's `-c` flag). I keep doing this and thinking about what sorts of functions and data structures I'd like to have available. I work out the high-level issues, e.g. exactly what outputs the program will provide, what inputs will be needed, how I can break the program into discrete parts, etc.

3) I take all the made-up functions and classes I've declared and separate them into categories, putting all the bits involving each category into a separate ".hh" file. For example, I might find myself with a "network.hh" file, a "user\_interface.hh" file, and a "math.hh" file. Each such file defines an \*interface\*. Whatever's left of the original program (once I've moved my made-up declarations into separate files) is called \*client code\*.

4) For each interface file, I write a corresponding ".cc" file that defines all the functions I've declared. I write the definitions one at a time. I don't worry too much about making the functions fast, I just try to make sure that each one will provide exactly the output I wanted when I was writing the client code. Most of the functions are straight-forward, and I can write them pretty quickly. If a function is taking an especially long time to write, or I think a function's definition is getting too complicated, I do the same thing I did in step 1: I try to break the task into smaller bits, and declare new functions to perform small parts of the function's task. The new, "helper" functions don't need to be declared in the original interface files, since the client code does not directly depend on them. The collection

of all these function definitions, helper functions, etc. is called the \*implementation code\*. The combination of an interface file and its corresponding implementation file is called a \*module\*. As I'm working on a given module, I compile to object code now and then, so the compiler has a chance to point out the mistakes I make as I'm working.

5) I take all the modules, and try to compile them together. At this point, the compiler executes another program called a "linker." The linker tries to make sure that each function and variable I used in the client code has actually been defined in one of the modules.

6) I test the code to make sure it's right, and that I didn't make mistakes like forgetting to release memory or file handles when they were no longer needed. At this point, I usually wish I already had written code to do the testing for me. (I'm trying now to get into the habit of writing testing code before I even start writing the program.)

7) Once I'm pretty sure the program is working correctly, I start \*profiling\* it to determine where it's spending its time. Usually, the program spends most of its time in only a few of the functions. I pick one of those functions, try to make it faster, and go back to step 5.

> 2) *Is the aim to make a class interface like a "skeleton" of the class > implementation?*

Yes, that's the basic idea.

> 3) *Is the class interface typically found in your header file?*

Yes.

> 3) *Is the class implementation typically found in your source file?*

Both headers and implementation files are "source files."

> 4) *Are abstract data types "abstract" because they are specified > separate to implementation?*

No. An abstract class is special, in that it does not have definitions for all of its methods. Such an undefined method is called "pure virtual." Since the class is not completely defined, it can never be instantiated; that's what makes it "abstract." Such a class is useful because each class "derived" from it can provide a different implementation of each virtual function. This feature supports a design style called "polymorphism." To understand polymorphism, you first need to have a basic understanding of a technique called "inheritance." See chapter 12 of TC++PL.

> 5) *In Bjarne Stroustrup's "The C++ Programming Language" on p317, he > talks about the class "I\_box," and seems to use the terms "interface" > and "implementation" interchangeably. Is there such a thing as an*

## comp.lang.c++. Re: Abstract Data Types – Separating Interface from Implementation

> *"implementation interface", as opposed to an interface?*

No, although an interface file may actually contain part of the implementation of a module. Bjarne is not using the terms interchangeably; please feel free to post any quotes from the book that you find confusing.

> *The following is confusing for me:*

>

> *" Our implementation choice is to use an array of integers. It would be nice to be able to separate the interface from the implementation.*

> *In some languages it is actually possible. Not so in C++ (or, at least, not without some gymnastics). The reasons are mostly technological. The compiler would have to have knowledge about all the files involved in the project in order to allow for such separation.*

>

> *In C++, the best we can do is to separate the details of the implementation of some member functions into the implementation file.*

> *The interface file, however, must contain the definition of the class, and that involves specifying all the data members. Traditionally, interfaces are defined in header files with the .h extension. Here is the stack.h interface file."*

>

> *This comes from the following page on abstract data types:*

>

> <http://www.relisoft.com/book/lang/scopes/11abstr.html>

That's a load of garbage. C++ provides better support for safely separating interface from implementation than any other language I know.

In fact, that single fact is probably the reason C++ is my favorite language.

The first paragraph you listed does have a shred of truth: When a C++ module (or any client code) is compiled, the interfaces of all supporting modules must be available. This differs somewhat from interpreted languages like Java that support a feature called "reflection." However, the compiler does *not* need to know about all files in the project, thanks to "dynamic linking."

The second paragraph says that the data members of a class must be included in the interface file. This is certainly possible, and is often done for "concrete" data types, for which performance is critical.

For most classes, though, it is absolutely not necessary. Only the methods of the interface must be specified in the interface file. Classes implementing the interface are simply derived from the interface class, and all the details are hidden in implementation files.

> *I'm trying to envision his "ideal" situation, where the interface is separated completely from the implementation. Any further insight greatly appreciated.*

>

> *Cheers,*  
>  
> *Deets*

I hope I've managed to clear some of this up; I know it's confusing. C++ supports a lot of different design styles, and I think Bjarne tries to show in his book how these styles differ, and how they can be used together. Really, there is no substitute for writing your own programs to find out why all these different techniques are useful, and how an interface differs from an implementation. If you really want to understand how client code differs from library code, try writing your own library of classes and functions to make some difficult task seem easy. Then, try writing programs that use your library.

Good luck, and please feel free to criticize or question any part of the above explanation.

–Jeff