# The Other Side of Complexity

by Chuck Allison

ISTOCKPHOTO

*Civilization advances by extending the number of important operations we can perform without thinking.*
*– Alfred North Whitehead*

Software development has always been an exercise in managing complexity. It has progressed as a discipline as good minds have created abstractions that transform complexity into simplicity.

There was a day, for example, when you had to write something like this:

```
mov  eax, DWORD PTR _x$[ebp]
add  eax, DWORD PTR _y$[ebp]
mov  DWORD PTR _z$[ebp], eax
```

instead of this:

```
int z = x + y;
```

High-level languages encapsulated the complexity of programming at the machine level. Writing logical and quantitative expressions became simple again.

Just as a statement is an abstraction of lower-level instructions, a function, or subroutine, is an abstraction of groups of related statements. Subroutines were the first software abstraction to facilitate things we take for granted today, such as refactoring and reusing code. Hence, functions enable the crucial practice of "separation of concerns" at the procedural level, so developers can build functional units independently.

## Function Abstraction

If writing a function constitutes "statement abstraction," what is "function abstraction"? I see a number of ways of looking at this. One approach is to group related functions into a single, logical unit, like the functions in C's <string.h> header or the static methods in Java's `Character` class. But does this really represent a substantively higher level of abstraction? Perhaps function overloading is a slightly better example of function abstraction—but only slightly. Instead of coming up with unique function names on your own, the compiler does it for you.

About the same time Dennis Ritchie was designing the C language, researchers in Scotland were working on ML, a strongly typed functional language. ML was the first statically typed language to generalize function definitions across *types*.

```
fun reverse nil = nil
|   reverse (h::t) = reverse t @ [h];
```

**Listing 1**

Consider the function in listing 1 to reverse a list.

ML uses pattern matching to select which branch of a function definition to execute. In the case of `reverse`, if the argument is the empty list `(nil)`, then `nil` is returned. Otherwise the first list element is bound to the variable `h` and the rest of the list to the variable `t` to set up a recursive call. The point of this example, however, is revealed in the `reverse` function's type signature:

```
val reverse = fn : 'a list -> 'a list
```

The token `'a` is a *type variable*, like a template parameter in C++. The `reverse` function can process a homogeneous list of elements of *any type*, as you can see in the sample execution in listing 2.

```
- reverse[1,2,3];
val it = [3,2,1] : int list
- reverse["one","two","three"];
val it = ["three","two","one"] : string list
```

**Listing 2**

Now you know where C++ got the idea for template type parameters. Accommodating a *family* of related functions in a single definition surely qualifies as function abstraction.

## Data Abstraction and Beyond

You would think that the next logical step would be to group data and related functions together into a coherent unit. It would have been, too, if it hadn't already been done almost a decade earlier in the form of *classes* in Simula-67. Simula-67 introduced classes, inheritance, and *subtype polymorphism*, the type of function abstraction you achieve every time you override a (virtual) method in a subclass.

```
Connection*con = net_connect(…);
if (con) {
    int err_code = use_net(con);
    net_close(db);
    if (err_code)
        // Report network error
}
else {
    // Report connection failure
}
```

**Listing 3**

```
Connection*con = net_connect(…);    // May throw
use_net(con);    // May throw
net_close(db);
```

**Listing 4**

```
Connection*con = net_connect(…);    // May throw
try {
    use_net(con);    // May throw
    net_close(db);
}
catch (…) {
    net_close(db);
}
```

**Listing 5**

```
try {
    use_net(con);    // May throw
}
finally {
    net_close(db);
}
```

**Listing 6**

```
Connection con(…);    // An object; not a pointer
use_net(con);
```

**Listing 7**

```
void deleter(Connection* p) {
    net_close(p);
}

shared_ptr<Connection> con(new Connection(…), &deleter);
use_net(con.get());
```

**Listing 8**

There are still more abstractions out there, however.

If you were programming in the late 1980s to early 1990s when object-oriented programming was starting to catch on, you were probably using C++. When writing a network application, you probably still had to use a C-like API since vendors were slow to catch up with the OO revolution. Listing 3 shows some typical code.

The function `use_net` must check for errors on every call to the network API. In the case of an error, it returns an error code that you have to pass on to your caller—after closing the connection. Very tedious.

Then along came exceptions to make the job easier by letting errors propagate up to the caller. The code simplifies considerably, as shown in listing 4.

Oops! If `use_net` fails, the connection is left open. No problem—you can catch the exception on its way up, like in listing 5.

But now you have to close the database on each possible path. Yuck. As shown in listing 6, Java makes it a little easier.

But there is still code here that you shouldn't have to write. A better solution uses *deterministic destruction*, wherein the connection is encapsulated in a class with a destructor that closes the database, like in listing 7.

Now you're down to two simple lines—code that is easier to understand and easier to test. The C++ term for this important practice is "resource acquisition is initialization" (RAII), which suggests its usage pattern: *acquire resources in constructors*, *release them in destructors*. Languages in common use that support this idiom include C++, D, and C#.

Even if you have to create a connection on the heap, you can use C++'s new `shared_ptr` as an RAII wrapper, as the code in listing 8 illustrates.

No matter how execution leaves this scope, the `shared_ptr` destructor calls `deleter`, which closes the connection. There's no need for `if` statements, a `finally` clause, or explicit exception handling.

## Reification, Patterns, and System Architecture

Suppose you have a linked list of objects of some type. How would an early 1990s programmer traverse it? The traditional solution consisted of a pair of "first-next" functions, something like listing 9.

Not too shabby, but why should we reveal that we're using pointers to traverse the list? What if we change our mind later and use some other mechanism? Users shouldn't have to change the code when we change our implementation.

Another downside to this solution is that users can have only one active traversal at a time, since the list itself does the advancing. Such a container isn't sure what kind of abstraction it is: Does it *hold* things or does it *traverse* things? To do both violates the important design principle of *cohesion*: Abstractions should do one thing well. A better design moves the notion of traversal into its own abstraction: an *iterator*.

In current terminology, the Iterator design pattern *reifies* sequence traversal. To reify is to make a

```
List<int> theList;
// After populating the list…
int* pItem = theList.first();
while (pItem != NULL) {
    // Use *pItem, then…
    pItem = theList.next();
}
```

**Listing 9**

concept concrete, or, as Wikipedia states, "Reification allows a computer to process an abstraction as if it were any other data." Moving iteration into concrete objects external to the sequences under traversal simplifies the container while allowing multiple, simultaneous iterations. It also allows the defining of algorithms in terms of iterators instead of containers, so you can apply algorithms to any type of sequence, be it a list, a vector, or a built-in array. This is the substance of the revolution in generic programming inaugurated by C++'s standard template library, yet another step up the abstraction ladder. All patterns—design patterns such as Iterator or Visitor, or architectural patterns like Layers or Pipes and Filters—encapsulate complexity by reifying some crucial system concept.

## Summary

In *The Mythical Man Month*, Frederick Brooks quotes Steve Lukasik of Northrop: "Yesterday's complexity is tomorrow's order… I believe that someday the 'complexity' of software will be understood in terms of some higher order notions." You and I are in the midst of this evolutionary phenomenon. Software will *always* be an exercise in managing complexity, because there appears to be no end to the problems to which we can apply automatic computation. To be successful, you and I need to acquaint ourselves with the abstractions already in circulation, and when we hit the next wall of complexity, we can reuse appropriate abstractions or invent new ones. **{end}**

**Do you see yourself as an abstractionist as well as a coder? At what "level" do you typically code?**

Follow the link on the StickyMinds.com homepage to join the conversation.