I ran across the following document from many years ago. But perhaps it might serve as an example of how object-oriented programming might occur in a (more or less) real situation.

But first, some background…

You may recall a genre of text-based[[1]](#footnote-1) computer games, called generically “Adventure Games” or “Interactive Fiction”, based upon the original “Colossal Cave” program (<https://en.wikipedia.org/wiki/Colossal_Cave_Adventure>). This in turn led to a game called “Zork” which eventually was released as the first of over three dozen games by the company *Infocom* (<https://en.wikipedia.org/wiki/Infocom>).

I really enjoyed played Zork 1 (the original mainframe Zork program was chopped into Zork 1, 2 and 3 to fit onto early microcomputers), and almost finished the game, but there was one puzzle I. Just. Could. Not. Solve!

Note: Zorks 1, 2 and 3 were released to the public domain many years ago. You can find them at <http://infocom-if.org/downloads/downloads.html>. For Macs, you can find the interpreters you’ll need at <https://inform-fiction.org/zmachine/macos.html>. Of course I haven’t tried any of these, but the Frotz program works fairly well under Windows.

Well, long story short, I’d read a bit about some of the technology behind the game and ultimately, after months of computer forensic research, I understood enough to be able to write programs (in that then-newish language “C”) to do things like

* Dump the program’s vocabulary (it was semi-encrypted, and I had to figure out the technique they used)
* Find the (also encrypted) text messages in the program
* List the source for the program in an assembler-like language
* Decode the verbs in the vocabulary table so I could annotate the source program listing with comments like “This routine implements the *take* verb”
* Probably a couple of other things
* And the ultimate utility, decompile the program into a C-like language[[2]](#footnote-2)

And all this eventually let me figure out how to get the last treasure and win the game. Yeah, it took literally about a year (hey, I had other distractions along the way, most notable a job!), but there was a definite sense of accomplishment for finishing the game. But also for the fun of having totally cracked open the internals of the game.

Now jump forward maybe a decade or so. I ran across someone who had done something somewhat similar. He called his program *DisInformation[[3]](#footnote-3).* So I emailed him and we started corresponding. It turns out he was a talented but fairly inexperienced programmer, and when I looked at the source code he sent me, I could see multiple ways that the program could be improved. Not in terms of adding features, but in terms of programming techniques that he wasn’t aware of.

So I wrote *DimTNG[[4]](#footnote-4)* version 0.1 in C, to show him how to approach the problem better. Then I wrote version 0.2 in C++, using classes. Hence the document below.

But instead of figuring things out from scratch like I did, he used a program called *TXD* (<https://inform-fiction.org/zmachine/ztools.html>) that did the complex job of turning the binary file into assembler language, dumped the symbol table, and provided other tools. And to make my program more familiar to him, I did my processing his way, using TXD output as a basis, rather than my original way of being totally self-contained.

Just for the sake of concreteness, here’s an example of TXD output. But I don’t expect you to be able to make a whole lot of sense from it of it.

Routine a7b8, 1 local

 a7b9: c1 8f 01 93 26 48 je local0 "l" ~a7c5

 a7bf: b2 ... print "look"

 a7c4: b0 rtrue

 a7c5: c1 8f 01 a4 b1 48 je local0 "z" ~a7d1

 a7cb: b2 ... print "wait"

 a7d0: b0 rtrue

 a7d1: c1 8f 01 a4 7b 4a je local0 "x" ~a7df

 a7d7: b2 ... print "examine"

 a7de: b0 rtrue

 a7df: c1 80 01 91 fd 92 57 92 60 4a

 je local0 "i" "inv" "inventory" ~a7f1

 a7e9: b2 ... print "inventory"

 a7f0: b0 rtrue

 a7f1: b1 rfalse

Just a few more terms:

* Inform – a compiler that can be used to write Infocom-compatible games
* Z-Machine – Just as Java and C# and other languages that aim to be cross-platform by compiling their source code into binary machine language that can then be interpreted or otherwise executed on different systems, all Infocom games are compiled to Z-Machine instructions.
	+ The TXD output above shows the Z-Machine binary to the left of each line (preceded by its location in memory), followed by a symbolic representation (e.g. “je” for “jump if equal”) of the binary.

So the ultimate goal of the decompilation process would be to take the TXD snippet above and output something like:

bool Routine\_A7B8(char \* local0) {

 if (strcmp(local0, “l”) == 0) {

 printf(“look”);

 return true;

 }

 if (strcmp(local0, “z”) == 0) {

 printf(“wait”);

 return true;

 }

 if (strcmp(local0, “x”) == 0) {

 printf(“examine”);

 return true;

 }

 if ( (strcmp(local0, “i”) == 0)

 || (strcmp(local0, “inv”) == 0)

 || (strcmp(local0, “inventory”) == 0)) {

 printf(“inventory”);

 return true;

 }

 return false;

}

Finally, this document semi-assumes you’ve got the source code for DIMTNG 0.2 available. If I were to dig deep enough, I can probably find a copy of this (and maybe even 0.1). But hopefully you can get some value from this document without the source code. However I should mention that in 0.1 (and its .doc file), I pointed out that he was using the C routines *malloc* and *free* with gay abandon, which would probably have led to storage fragmentation and risk running out of memory for the next *malloc* request. So I showed him how to implement a simple stack that could help. I called this a Push-Only-Stack[[5]](#footnote-5), or POStack.

Oh, and realize that the syntax for classes and methods in C++, as described in the original document, differs a bit from the syntax used in C#. And C# struct’s and class’s are *almost* identical to the way C++ approaches them, but there are some minor differences.

So here’s my original document…

DIMTNG 0.2

Assuming that what I sent you was DIMTNG 0.1, here’s what I’ve added for release 0.2.

Except that I can’t call it “adding”. It’s much more like “repackaging”.

I alluded to the following a bit in my original DIMTNG.doc file, but it’s worth re-emphasizing. I think that the main problem with your original version of DIM is that there wasn’t a clear delineation between, say, decompiling the subroutine, scanning the input line, and maybe other things such as looking up subroutine names in some kind of symbol table.

You were trying to juggle too many things at once.

One of the main things, perhaps *the* main thing that I’d like you to get from my program is how various pieces of the program are insulated[[6]](#footnote-6) from each other. For example, scanning an input TXD line is separate from everything else. The overlap between it and, say, the decompilation process is nil, nada, zip, zero, etc. This is considered good programming style. If you have a problem scanning the input line, you just need to look in a small number of subroutines to isolate the problem. OTOH, if you must change the scanning routine, it’s self-contained.

The decompilation process may (and indeed, does) use the results of scanning the individual TXD lines. But it’s considered good design to arrange things so that the scanning (or, as I prefer to call it, the parsing) routines take as *input* an unparsed line, and as *output* produce a more structured version of the line. This in turn is *input* into the decompilation process, which produces whatever its *output* is (which may, or may not, be readable ASCII).

So let’s talk a bit more about encapsulation, which in turn brings us back to “repackaging”.

Traditionally, encapsulation has been embodied in the concept of a subroutine. But as we’ve seen in, say, the TXD line parser of DIM 0.1, parsing a line actually consists of a main subroutine, ParseTXDLine(), but also sub-subroutines, such as GetMnemonic(), as well as GetToken().

Now while we could talk about how to encapsulate those, I’d rather start with a simpler example. The whole Push-Only Stack paradigm consists not only of the functions involved, but includes the data structures it references as well.

We’ll use C++ to do this. I’ll spend quite a while on this, but after that’s done, we’ll pick up the pace.

# Encapsulating Push-Only Stack

So let’s start by encapsulating the Push-Only Stack. What are the pieces? Well, first, the user-interface to it consists of a series of subroutine calls. So if all the user sees is subroutine calls, presumably the way these subroutines are implemented are irrelevant.

For example, in that full-text indexing program you did, perhaps you designed it so that you had a Lookup() function. When you changed from a linear search to a hashed search, did the user-interface (i.e. the Lookup function, its parameters, etc) have to change? Ideally, no. The internal details are none of the caller’s business.

So we’ll start with declaring a struct called POStack, having the same parameters as in DIMTNG 0.1. But we’ll use the C++ keyword of “private” to indicate something new. (Note: the syntax is “private:”. The colon makes it look like a label, but it’s not. It’s perhaps unfortunate that the C++ designers chose to re-cycle this syntax, but it does seem to work OK in practice.) The “private:” designation indicates the following. If the normal program tries to reference any of these variables, the compiler will generate an error message!

I know I’ve told you the following story before, but it’s worth repeating. Several years ago I was working on a large project for a company. We’d just finished Phase I and needed more manpower for Phase II. Fortunately, we had another project that was winding down, so we transferred some of the programmers on that project over to ours.

Of course, it took the newcomers some time to come up to speed on our system. But we had a small lucky break at one point. One day I was going past the cubicle of one of the new guys, Dan, and happened to glance at his PC. There, on the screen, was one of my #include files. Of course curiosity got the better of me, and I asked him why he was looking at it. He said that he needed a certain piece of information to implement one of his subroutines, and he thought it was in one of my structures.

What he didn’t know (because we goofed and hadn’t told him) was that our project had a naming convention that certain fields in #include files were not supposed to be referenced by anyone other than the owner of that structure (in this case, me). But he didn’t know that, and if I hadn’t been wandering by just as he was looking at my code, he probably would have just gone ahead and extracted the information from the structure.

Which would have been a bad move, because we’d already decided that one of the enhancements for Phase II was to rewrite that routine of mine. So when I got around to doing it, Dan’s code would have blown up.

What would have helped was if the *compiler* could have helped. If we could have told the compiler that some of the fields in the structure were *not* generally available, then Dan could have got an error message the first time he tried to reference these fields, and we wouldn’t have had to rely upon the happenstance of my walking past at the right time!

Well, C++ to the rescue! When you put the pseudo-label “private:” in a structure, all variables after that are not available to any code, except those that are members of the struct (more on this “member” concept a bit later). (Note: Well, not quite. There’s also the pseudo-label “public:” (and another one called “protected:”, but I’m not going to discuss that) that makes everything after that to be generally visible).

Which brings us to the C++ keyword *class*. The terms *class* and *struct* are totally synonymous, with one exception.

A *class* is a *struct* in which all the fields are, by default, *private*.

Thus a *struct* is a *class* in which all the fields are, by default, *public*.

That’s it. That’s the *only* difference between *class* and *struct*.

Thus:

 class foo {

 int x;

 };

is *exactly* the same as

 struct foo {

 private:

 int x;

 };

Alternatively,

 struct goo {

 int y;

 };

is the same as

 class goo {

 public:

 int y;

 };

# OK, Getting Back to Push-Only Stack

Our initial definition of the Push-Only Stack (POS) structure was

struct POStack {

 char \*base, // The start of the stack

 \*top, // The highest address in it. You

// can use \*top, but not \*(top+1)

 \*NIP; // Next Insertion Point - where a

// Push tries to put things.

 // Initially, NIP = base;

 int GrowthIncrement; // Amount to grow if we

// run out of room

 int HWM; // High WaterMark - How much data

// was ever used at once

};

We also had functions that manipulated a POS:

char \*PushVoid(POStack \*pPOS, void \*p, int length); // Returns pointer into stack

char \*PushString(POStack \*pPOS, char \*p);

// Special case. We'll calculate the length

void ResetPOS(POStack \*pPOS);

void InitPOS(POStack \*pPOS, int InitSize, int GrowthIncrement);

void FreePOS(POStack \*pPOS);

So we’ll now do two things. First, we’ll turn the *struct* into a *class*, allowing all the fields to default to *private*. Second, we’ll incorporate our five functions into the class as members. We’ll wind up with this:

class POStack {

 char \*base, // The start of the stack

 \*top, // The highest address in it. You

// can use \*top, but not \*(top+1)

 \*NIP; // Next Insertion Point - where a

// Push tries to put things.

 // Initially, NIP = base;

 int GrowthIncrement; // Amount to grow if we

// run out of room

 int HWM; // High WaterMark - How much data

// was ever used at once

public:

char \*PushVoid(void \*p, int length);// Returns pointer into stack

char \*PushString(char \*p); // Special case. We'll calculate the length

void ResetPOS();

void InitPOS(int InitSize, int GrowtIncrement);

void FreePOS();

};

The things to notice here are:

1. It’s now *class* POStack, not *struct* POStack (although we could have achieved exactly the same effect by sticking in “*private:”* at the beginning. Still, that’s not the normal way to write it.)
2. Because the variables are private, there is no way the user of the class (a programmer) can see them, retrieve their values, modify them, or anything else. If he tries, he’ll get a compile-time error.
3. The functions are included in the class definition and are identified as *public*.

Now we’re not done. For one thing, we still have to define the member functions (sometimes called *methods*). We also have to know how to refer to them. We’ll take the latter first.

In C, if you had *struct foo { int x; };* and you wanted a couple of *foo*s, you might write

struct foo foo\_1, foo\_2;

In C++ we can abbreviate that a bit. We can leave off the *struct*, and just write

foo foo\_1, foo\_2;

We’d refer to the member variable of these as foo\_1.x and foo\_2.x. Of course the two x’s are different, being qualified by which instance of foo they’re in.

We use the same syntax for referencing the member functions of a struct/class. Suppose we had

 POStack stk\_1, stk\_2;

and we wanted to reset each. We’d write

 stk\_1.ResetPOS();

 stk\_2.ResetPOS();

OK, that part’s simple enough.

So how do we define them? The original code for ResetPOS() was simple enough.

void ResetPOS(void) {

 NIP = base;

}

But now we’ve got a problem. There’s nothing that says that ResetPOS() is associated with the POStack class. “But it’s declared in POStack.c” (I hear you cry)! So what? In the foo class we had a variable called “x”. We could have had a goo class with another x. But if we tried to write

 x = 42;

the compiler would have complained. And similarly, if we had two different classes with a member function called ResetPOS (unlikely, I know, but more likely if we wanted to call the function just Reset), the compiler wouldn’t know which ResetPOS we were talking about.

So we have to use a new syntax to associate a member function with its class. We do this by prefixing the name of the function with the class name and two colons.

void POStack::ResetPOS(void) {

 NIP = base;

}

There are a couple of things to notice here.

1. We don’t seem to have defined NIP nor base. Shouldn’t we get a compile-time error? No, because we know (from “POStack::”) that we’re talking about the POStack class. So the compiler can and does realize that these are the variables defined in the class.
2. But these variables are private. Why doesn’t the compiler complain? Because, as I said, member functions can access them. It’s just that other functions (e.g. main()) can’t.
3. The next one is more subtle. We have stk\_1 and stk\_2 defined above. That means that there are *two* NIPs and two base’s, one for each stack. Which one are we referring to in our definition of ResetPOS? The answer lies in the fact that we can’t just write *ResetPOS()*, we have to write *stk\_1.ResetPOS()*. And the compiler will now know to reset the NIP variable for stk\_1, and leave the one for stk\_2 alone.

OK, there’s lot’s more, but look at the code for class POStack and you’ll see what’s going on.

# The other classes.

This isn’t going to be an exhaustive list. I’ll just hit the highlights.

The classes I came up with were the following. Keep in mind one thing. I have, as a long-term goal, to produce a Windows-based MDI application that lets us see multiple subroutines at once. And I can sort of see the desirability to show subroutines from other games at the same time. So some of the following classes are built with this in mind.

1. class InfGame
	1. Represents an Inform game. From our POV this includes the TXD listing file, and the gameinfo.dbg file (with the symbol table info from Inform’s “-k” parameter). Even if we have multiple subroutines displayed at once, there will be only one InfGame if they are all from the same game. If we do allow multiple games to be displayed simultaneously, we’d have more than one InfGame.
2. class TXDFile
	1. All requests for accessing a given TXD file go through this.
3. class InfDebugFile
	1. Holds all the symbol table info. In my testing, it’s already full filled in, thanks to the compiler. In real life, we’ll have to add routines so that the user can add subroutine names, variable names, etc, and write it all out to be read in during the next decompiling session.
4. class TxdTOC (TOC == Table of Contents)
	1. Once the user gets a complete basic decompilation, he’s probably going to be concentrating on a small number of subroutines at any given time. Rather than have to repeatedly scan the TXD file every time, we do the following. We make an initial scan, building a table. For each Z-Machine subroutine, we store its address and the offset into the TXD file. We then write the entries to a disk file. The next time the user runs against the same game file, we’ll quickly load this file, and can now go directly to any given subroutine.
5. struct TxdTOCEntry
	1. Simply contains the two fields above, address and offset. I made this a struct to say that everybody can access these fields.
6. class ZOpcode
	1. Represents a Z-Machine assembler statement in a more structured form. As we process each subroutine, we’ll build up an array of these. We can then go back and do whatever we want to the opcodes in an easy fashion, since they’re all in front of us, just a subscript away.
7. ZSub
	1. Represents a subroutine. It’s initial job is to build that vector of ZOpcodes. Once we’ve got that, we can then manipulate it. We can give a fairly raw dump of the data, similar to TXD, but at least we’ll be able to write “x = y + z;”, rather than “add G43, local0 -> local1”.
	2. We could then go on to analyze it further to form more complex expressions, discover higher-level constructs such as *for*, *while*, and maybe even *objectloop*, and so on.
	3. As a small exercise, I wrote a simple cross-reference utility. For each subroutine in the game, I produced a list of all the subroutines it called. To do this took under a dozen lines of code! All I had to do was to read in each subroutine in turn, then go through each ZOpcode. When I came to a *call* opcode, I just displayed the first operand, which is the name of the subroutine being called!

The code’s all there for you to look at. But I must warn you of two things.

First, it’s one thing to do nothing but decompile the entire program. But if you want to do things like work with individual routines (as users are going to want to do), or do other processing (such as xref’s), then you *have* to build a more complex infrastructure. So be prepared for a certain amount of housekeeping that has nothing to do with decompiling per se.

Second, I’ve tried to encapsulate things. Yeah, we’re likely to be the only users of these classes, and I’d kind of like to think that we can be trusted. :-) Still, encapsulating things, even from yourself, will probably help you in the long run. Also, while the classes aren’t really all that self-contained, I can almost imagine someone, some day, wanting to do something to the game, and using these classes as a base. So either way, it’s the right thing to do.

What I’d suggest is that for starters you go through main(), and get a rough feel for the user interface. Try it out.

Then go through the header files and see what the classes are. In particular, create a new file (not part of the project), that has a) the name of the class and b) it’s public variables (very few) and its public methods. This will define, well, I guess you could almost call it an API for manipulating the subroutines in a game. All the rest is hidden from you (as long as you stick to main()), and are details that don’t affect you!

# Just a few parting words on Object-Oriented Design

Forget C++ for the moment. Pretend we’re talking about the whole DIM project in English (albeit with technical terms included in our vocabulary).

Suppose I asked you what we were trying to do, and how we were trying to do it. Well, I suppose you could say, “We’re trying to decompile an Inform program.” Fair enough, but that’s far too general, and we’re going to have to break down the process into a bit more detail.

So here’s a second cut. We have an *ASCII file* as the assembler for the game, courtesy of TXD. It consists of a number of *subroutines*, each prefixed by some *header information* (e.g. indicating the number of *local variables*). Oh, and the file itself has a *small header* that tells, for example, the addresses of the starting and ending subroutines. Each subroutine consists of one or more *assembler statements*. We process each routine in turn, decompiling it and displaying the *Inform source*. We also have an optional *file with debugging info* in it that will allow us to display symbolic names.

You’ll notice that I’ve italicized most of the nouns in the preceding paragraph. The rule-of-thumb in Object-Oriented design is that each noun will likely turn out to be a class. So our candidates are:

* TXD ASCII file
* TXD overall header
* Subroutine header
* Subroutine body
* Individual assembler lines
* Local variables
* Inform source
* Symbol table info

“Well, duh!” (I hear you cry.) “So what? That’s obvious. How’s that going to help me?”

Fair enough.

To be honest, it actually took me a lot longer than I’d expected to come up with the classes that I did (InfGame, ZSub, etc), and to get them *right*. The easy part of object-oriented programming is to define classes and methods. The hard part is to analyze your program well enough to get a set of classes that can be easily used and cleanly implemented. If you do the job wrong, you’ll be battling the classes you’ve defined, rather than having them make the job easier.

Whole books have been written about OO-design, and I’m not about to write another one here. But I do want you to realize that this aspect of program analysis is hard. And in some respects, for me, is the most enjoyable.

1. With commands like “go north”, “open door”, “take jewels”, “kill troll with Elvish sword”. [↑](#footnote-ref-1)
2. Ironically, I found out much later that the program was originally written in a LISP-like language! [↑](#footnote-ref-2)
3. Since the company that put out the games was *Infocom*. [↑](#footnote-ref-3)
4. **D**is**I**nfor**M**ation, **T**he **N**ext **G**eneration [↑](#footnote-ref-4)
5. It was actually a Push-*Mostly*-Stack, but the details aren’t important here. [↑](#footnote-ref-5)
6. Technically the term is *encapsulated*. [↑](#footnote-ref-6)