Is Top-Down Design a Waste of Time? Is Bottom-Up Coding Faster?

by Erik Zoltán

It is widely agreed that object oriented (OO) designs tend to become more stable as they mature, because objects are so much more reusable than subroutines. Objects are actually strengthened when they are subjected to new requirements. By contrast, traditional procedural programs tend to encounter difficulties when asked to fulfill tasks for which they were not originally designed. The bottom-up design approach exploits this advantage of OO systems by integrating the tasks of design and coding into a single unified process.

Part I

This is intended to be the first of two articles on object oriented design. In the first part, I will try to stand traditional structured programming principles on their collective head by arguing in favor of bottom-up design. I will point out some of the strengths and weaknesses of the two competing methodologies, and will end by drawing the paradoxical conclusion that even a mostly-wrong bottom-up design is better than a mostly-right top-down design, because bottom-up designs are more self-correcting. The second part, to be published later, will contain more specific advice on how to create a good bottom-up design. The first step in the process involves creating several competing "rough sketches" of the system, and selecting the least-controversial components to implement first. This provides valuable information that makes the rest of the design process easier and less risky. Part II will then present some more concrete examples of this implementation strategy using an OOM development system called EsiObjects(tm) ("Easy Objects").

Principles of Top-Down Design

- Develop a large-scale concept of the system before getting too detailed
- Finding the "correct" design makes it easier to write the code
- Solve all the problems on the drawing board before you start programming
- Little of value is learned during implementation

What's Wrong with Top-Down Design?

The software design methodology I learned in college could be summarized as follows: "Always design your programs starting with the highest-level components first. Begin with a general picture of what the program does and break it down into smaller and smaller subroutines, until each subroutine becomes easy to write." *This top-down approach makes a great deal of sense for a number of reasons.* First, it encourages modularity and simplicity. A programmer who has been thoroughly indoctrinated into this approach becomes much less prone to the dangers of producing unreadable "spaghetti" code. Second, whenever something truly difficult presents itself, you can just create a subroutine for it, and worry about how it works later. Often the problem will seem much less troublesome when you tackle it in isolation. Third, it may improve performance in the long run, since procedural modularity means that each subroutine can be optimized separately without necessarily affecting the others. Finally, the top-down approach can even make it possible for many programmers to work together on a problem: just give each one a well-defined subtask, after clearly describing any interactions that will be necessary between them.

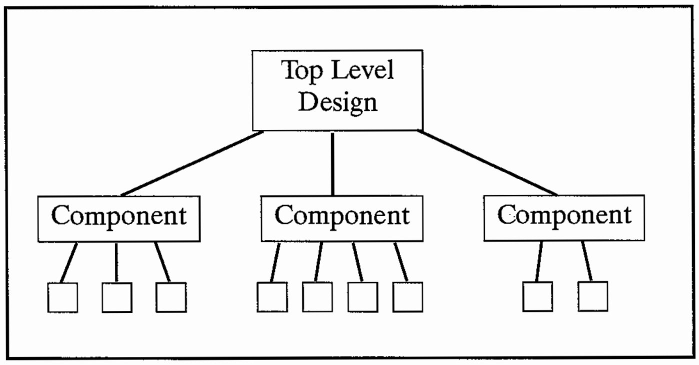

Fig. 1 Traditional top-down design methodologies are said to be the basis of procedural programming.

However, there are some clear *drawbacks* to the top-down approach. The most obvious of these is brittleness. Very often, when one is "down in the trenches" coding some subroutine, it suddenly becomes obvious that a completely different approach would have been better. For example, the parameters that have been passed in don't convey enough information, necessitating absurd complications that could have been avoided with another approach. Or it turns out that several different subroutines need to perform redundant calculations in order to fulfill their supposedly "discrete" missions. Such discoveries may ultimately require large parts of the program to be redesigned. Some experienced top-down programmers will tell you to write the program, then throw it away and rewrite it twice: on the third draft you will (hopefully) have finally gotten it right. Hence the concept of "prototyping." The prototype is, among other things, a useful throw-away version from which to learn.

Another side of the same coin is that it becomes ***harder and harder to add new features***. Many years ago, for example, I was asked to add a seemingly "simple" cut-and-paste feature to an old MUMPS routine editor. (It was still called MUMPS in those days.) It turned out that this feature affected all cursor migration operations, which now needed to support the possibility of highlighting the selected text. The entire editor had to be re-written from scratch, even though the initial design seemed to work fine. However, the re-designed editor was never put into service, because additional design changes caused further problems. A second rewrite from scratch was out of the question, and the first rewrite had gradually become unstable. We had to face the painful prospect of using the initial "no frills" version with all of its drawbacks.

Another drawback of the top-down approach is that it *sometimes impedes performance.* At the top level, one is frequently insulated from the sort of implementation details that can make a real difference in terms of execution speed. By the time implementation actually occurs, the wrong decisions may already have been made. For example, one top-down design I created early in my programming career suffered from unacceptable performance problems. Despite extensive efforts at optimization, this elegant program ran almost one hundred times slower than a rather ugly public-domain version for which source code was available. Examining the PD code revealed a more efficient low-level strategy so "obvious" it made my jaw drop. Once again, it seemed easier to start over from scratch, because the top-level design was now obsolete and the PD version was riddled with many "quirky" programming practices.

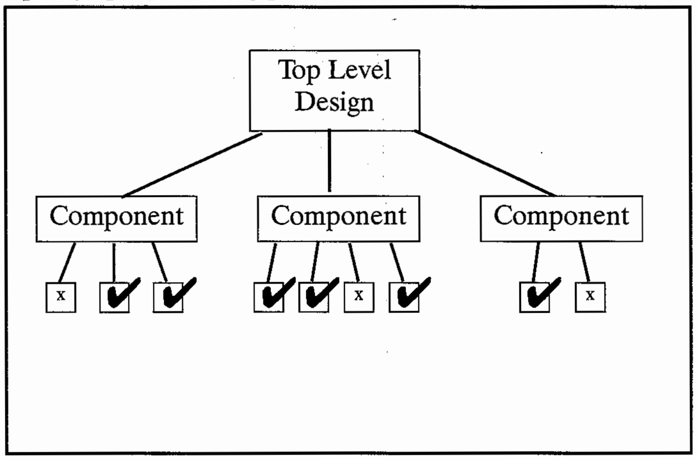

Fig. 2 If design flaws are present, they are most likely to emerge during the implementation process.

The bottom line is as follows: the traditional top-down procedural design methodology works well as long as your initial design is correct, and unexpected new features will not later require extensive re-design work. Unfortunately, *procedural programs tend to become less stable over time*, as more and more enhancements are inevitably required.

How is it, then, that the best "top-down" procedural M programmers are able to create high-performance code that doesn't eventually become unstable? Presumably they can somehow produce better designs in the first place, because they are able to clearly visualize the low-level components as they are designing the high-level ones. In other words, by virtue of their ability and experience they are able to design with both the top-level and the bottom-level details in mind—they're really designing "from the edges" towards the middle. In addition, they may instinctively design in certain kinds of flexibility, so that future enhancements will not cause too many problems. That's not really much different from the approach being advocated here, it's just too hard for "the rest of us"—the majority of programmers who get headaches from trying to visualize such grand schemes in a purely abstract way.

In some ways it's useless to talk about top-down design because nobody really ever does it anyway, just as we never wrote out all those flowcharts, back when doing that so was popular. It takes much too long to thoroughly design the entire thing before you start typing code! Neither programmers nor their managers are patient enough to wait around for extensive design work. Something in us rebels against doing it the slow way, and we rush to our keyboards for the comfort of typing code, in much the same way that an impatient child cannot resist biting into a Tootsie Pop to get at the tasty center.

Principles of Bottom-Up Design

The difference between top-down and bottom-up design is like the difference between a dictatorship and a grass-roots movement:

- A bad design is easier to produce than a good one—never assume that your design is correct;
- Always start working on a tough problem at its weakest, most critical point;
- The implementation process can yield valuable new information relevant to the overall design;
- Therefore, design and implement the most obvious components first;
- Later you can synthesize these components into a vision of "the big picture."

You Mean, Start at the Bottom?

To many people, designing from the bottom up seems ridiculous at first—somewhat like reading the text of this article backwards, starting with the last paragraph. *You may not accept the idea of bottom-up design if you don't believe the claim that the most costly design flaws are those uncovered during the implementation process.* This claim makes a great deal of sense to me, because it is precisely during implementation that vague promises must be backed up with specific lines of code. It also makes sense because it agrees with my not-infrequent experience of stopping in the middle of a routine, muttering under my breath, and suddenly deciding that lunch time has arrived!

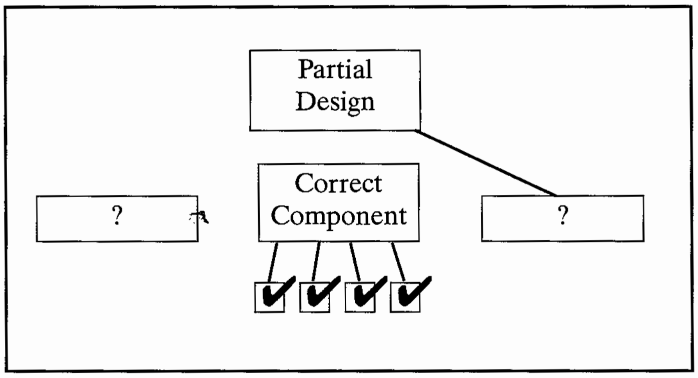

Fig. 3 Bottom-up designers first design and implement the most critical components they know will be required.

Like top-down structured programming, the object oriented (OO) approach also emphasizes modularity. A system is broken down into a number of different objects, each having a clearly-defined area of responsibility. But unlike subroutines, objects represent groupings of logically-related data, along with all the relevant code to operate on that data. A much-touted advantage of objects is that they represent easily-understood real-world entities, and that by grouping the relevant code along with the data upon which it operates, they make it easier to assign blame when problems arise. (Ideally this should be correct, but in practice it depends on the quality of your OO design, so please read on...)

In OO systems, there is no need to map components of the problem domain into data structures and procedures. Instead, you begin by creating objects that look like the things in the problem domain, and then find sensible attributes and behaviors for these objects. *The procedural interactions between objects flow naturally from their collective needs and responsibilities.*

The top-down designer says, "this is what I somehow want," while the bottom-up designer says, "this is what I can effectively achieve."

In many M systems, the closest thing to an OO design is the data dictionary. The data dictionary is emphatically not object oriented, but it does describe the layout and organization of data in the system, and it also links this data to certain components of related program code for tasks like input validation and indexing. If you think of each file as a collection of objects, all having the same type, then you will have made a good start. A true OO system introduces a number of new concepts that go far beyond the data dictionary. But imagine creating the data dictionary first, and designing the rest of the system entirely around it, and you will have a preliminary idea of the OO design process.

When Problems Arise

Two primary pitfalls arise when highly-trained top-down procedural programmers try to design OO systems. The first is procedural, and the second is top-down. First, there is a tendency to procedurally modularize the code but not the data. This can result in one kind of object where two would have been better. It can also result in object A performing operations that ought to be the responsibility of object B. Second, there is a tendency to perform a top-down design of the entire system, all in one go. This produces an unnecessarily complex design containing many mistakes, resulting in time-consuming redesign that could have been avoided by starting out more simply. ESI President Terry Wiechmann is fond of saying that learning OO design requires "brain surgery."

Imagine that you have fully designed a system, and that the design is *flawed* in some way. The flaws will generally reveal themselves in one of three ways:

- They may become obvious while contemplating the finished design,
- They will *most likely* become obvious when the design is being implemented, and
- They may only become clear much later on, when unexpected new features need to be added.

By the way, note that most design flaws become apparent in steps 2 and 3, during the implementation process. In the top-down approach, we may have to go back to the drawing board when these flaws become apparent. But what if certain well-defined components of the system have been partially implemented *before* the overall design has even been finalized? Will the final design become clearer *after* these objects are already in place? Can a bottom-up design methodology result in a highly-flexible set of generally-useful objects that are certain to become part of the final design? Might this not *minimize* the possibility that serious design flaws would be uncovered during implementation? Is such a thing even *possible*?

The answer to all of these questions is a resounding YES. Sometimes one tends to get stuck in the assumption that one single design is "correct." However, there are generally many different ways to envision a system, and there are usually several competing designs that could serve the purpose. For example, there's the simplest-possible design that would match the current specs, and there's a more complex, fully-specified design that would include future needs. Beyond that, there may be several competing designs that work differently, and the choice among them is sometimes just a matter of style. And most importantly, there are some *just-plain-wrong designs* that somehow always tend to look much better to us at first. In bottom-up design, it is important to humbly admit how little you know about the best possible design. *The less you think you know, the better.*

One good way to achieve this "uncertain" state of mind is to *make rough sketches of several competing designs* without choosing among them. Common to all these designs, there will generally be a central core of uncontroversial things that you *do* know you will *definitely* need. If several competent programmers are voting by secret ballot, they will probably disagree on some portion of the design, but they are certain to agree on the "easy" part. And that's the place to begin. In bottom-up design, one gets to start by implementing a few simple well-understood components of the system.

Always Attack the Weakest Point

In a system to translate French into English, for example, you might begin with an object to represent the contents of a French language text file containing the source text, and this object might provide services like returning the text of a single word, sentence or paragraph. Of course, you could start by implementing objects to represent the nodes in an augmented transition network or maybe the parse tree, but "Why attack a lion when there is a lamb in the field?"

The first objects to implement are generally those that *provide* services rather than requiring them. For example, you should develop data-level objects before user-interface objects to display and modify them. General-purpose objects often come before highly-specialized ones, and small component objects before the larger-scale objects that will contain them.

After implementing some of the best-defined elements of the system, you can then look back at the competing design options with a greater degree of information. At this point, some flaws may already have become obvious, and a few fuzzy ideas should now seem a bit clearer. Always struggling to resist the dangerous temptation to revert to top-down design, you can still continue to find the most obvious things to work on.

Finally it becomes necessary to select from among the competing design options identified earlier. At this point, it is a good idea to spend some time fleshing out the design you have decided to choose. This is the riskiest phase of the bottom-up design process, and the most important thing is not to avoid mistakes, which is impossible anyway, but to make them as late in the process as possible, while at the same time identifying and correcting them as early in the process as possible.

In Part II of this article, an actual system will be created using bottom-up design. EsiObjects(tm) was chosen because it is an M-based Windows development system that provides a great set of tools for object oriented implementation, and many aspects of the bottom-up design process.

If a designer is capable of producing a flawless design from scratch, then any design approach will work. However, it is my opinion that an imperfect bottom-up design is better than an imperfect top-down design because the top-down approach leaves you vulnerable to time-consuming redesign when the flaws are finally uncovered. Bottom-up design, though counter-intuitive to some, minimizes the amount of time you spend going down blind alleys. This is achieved by making mistakes later in the process, rather than earlier, and by trying to correct them more quickly once they have been made. Central to this strategy is the observation that critical design mistakes are most frequently uncovered during implementation. To be continued... **M**

Erik Zoltán has been programming, writing, and teaching in the M community for the last 7 years. He is now also teaching EsiObjects™ programming classes for ESI Technology Corporation in Natick, MA.

The difference between problems and *solutions*.

> M Application Development
> Year 2000 Modifications
> Expert Technical Support
> Telephony Application Development
> Legacy Software Support
> Upgrades
> Interfaces
> Data/Platform Conversions
> GUI Application Development
> and much, much more.

Development Partners Inc.

Software solutions for the medical and financial marketplace.

561.362.4477
DevPartner@aol.com