Bottoms Up!

Why Top-Down Design may be a Waste of Time, Part II

by Erik Zoltán

In many ways, the implementation phase is the most interesting and exciting part of the development process. For it is during implementation that vague promises must be backed up with specific lines of code. And it is during this phase that unexpected twists often occur, which can result in great improvements to the application's overall design. The cliche that "hindsight is 20-20" reflects the fact that design work is often different the second time around. The bottom-up approach exploits this fact by integrating design and coding into a single, unified process.

*In **Part I** of this article (available at **www.esitechnology.com**), I outlined the benefits of bottom-up design, advancing the claim that traditional top-down design techniques are dangerously vulnerable to costly mistakes. In this part, I will show a real application in which the bottom-up approach was successful, thereby lending support to my earlier claims.*

Principles of Bottom-Up Design

As outlined in Part I, the bottom-up design process has the following steps.

• Analyze the problem domain in detail, without a solution in mind.

• Create a "thumbnail sketch" of your proposed solution with some of its major components.

• Consider alternatives, and develop a certainty analysis of the components.

• Implement the most certain design element.

• Revisit the top-level sketch, and flesh it out in more detail before implementing the next part.

Creating a Report Generator

In my example application, the problem was to create a report generation tool in Windows that could talk to an M global database. The long-term goal was to migrate the database to a native OOM format, but the short-term need was to create a report-generation tool for Windows users. **EsiObjects™** ("Easy Objects") was chosen for this project because it is an M-based Windows development system that provides a flexible set of tools for object-oriented implementation and many aspects of the bottom-up design process.

The M global database consists of medical information centered around a patient file. The most important reports include patient and prescription data, but it should be noted that many different files will eventually be included. At present, only the following files are included. The arrows indicate pointer relationships.

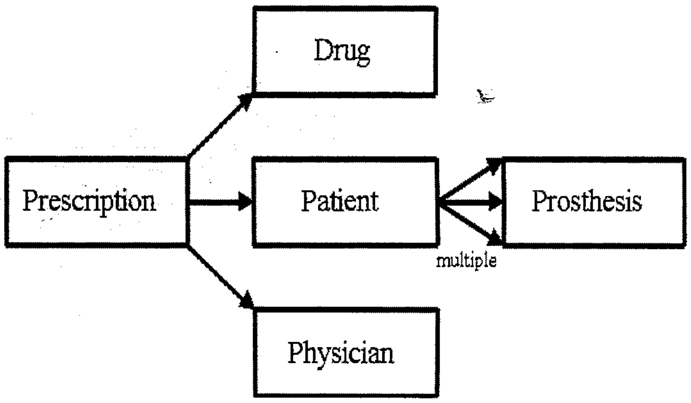

Figure 1

The user needs to be able to specify the target file and to select the fields for the report, as well as the sorting fields. A more detailed query engine is planned for the future.

Design Sketch

Time was of the essence when the project was first started, since there was a commitment to use the system as an in-class exercise in only two weeks, and the development team consisted of only one programmer. A quick thumbnail sketch of the application looked as follows:

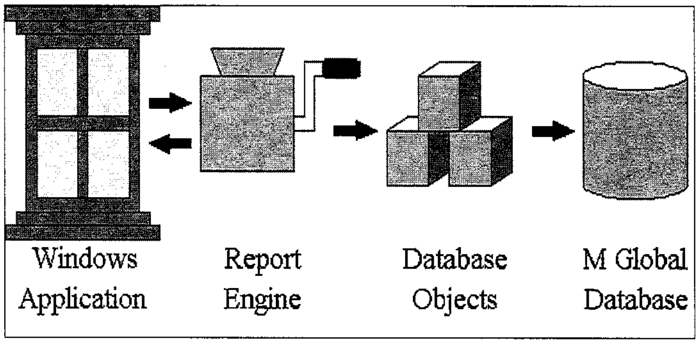

Figure 2

In Figure 2, the arrows indicate the direction of messaging. My first guess was that the application would interact with some kind of report generation object that would produce the report. The user could utilize the application to specify the details of the report. The application would provide this information to the report generator, which would query the file for the appropriate records, which it would then pass back to the application for display or printing purposes.

Encapsulation is an important principle of object oriented systems. Basically, it means that each object is responsible for its own internal data, and it forces external objects to obey a specific protocol when interacting with the object. But the use of M globals presents a problem, since access to the data in M globals is not restricted—any routine (or any object, in an OOM system) can modify any global.

For this reason, part of my thumbnail sketch was to create a series of "database objects" to represent the information in the M global. This was also in keeping with the long-term migration to objects that was part of the project's initial specification.

Certainty Analysis

The design is never truly finished until the project has been fully implemented, so each component of the initial thumbnail design sketch is subject to a certain degree of uncertainty. There are four major components of the thumbnail design, and each one has a different level of uncertainty (See Figure 3).

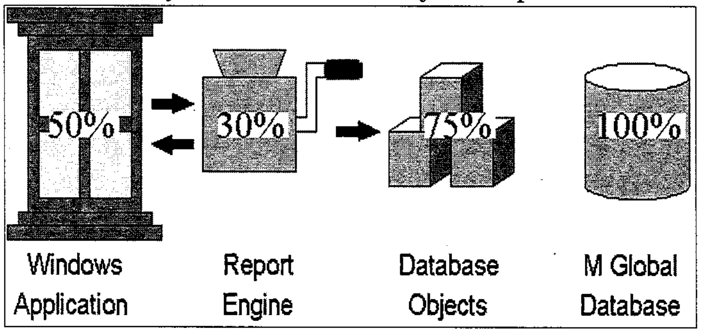

Figure 3

Two things seemed relatively clear at the start of the implementation process. First, the legacy globals had been around for years, so there was no doubt about the global structure. The application requirements were already on paper, although one could foresee that they might change as the project continued to evolve beyond the "proof of concept" level.

Of all the components needing to be implemented, I was most certain of the database objects for two reasons:

• They represented the information in the globals, which was already well-defined.

• They would provide services to other objects and not vice-versa.

For these reasons, I decided to tackle the database objects first.

The database objects simply mirror the structure of the database globals (see diagram below). There is a file, and each file contains a number of entries. The file may also use indexes to sort the entries in different orders. For example, an oversimplified patient file might contain information about any number of patients, sorted by Name and SSN. (In reality many more indexes are used.)

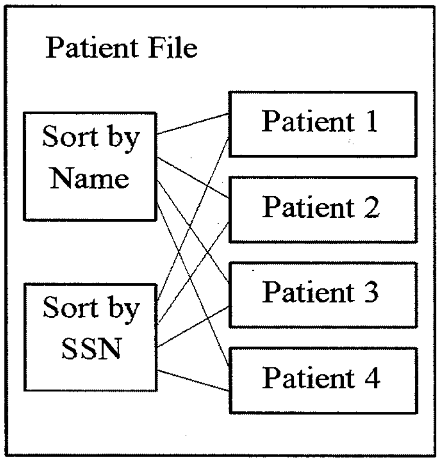

Figure 4

For this reason, I decided on three basic kinds of object. There would be one kind of object to represent the information in a single file, and another kind to represent the information in an entry (patient or prescription). Finally there would be a sorting object. The "file" object would respond to queries for entries and would somehow contain a large number of "entry" objects. I decided to have a separate class for each kind of entry and to have a single class to represent the information in all the files. I postponed the question of sorting for later.

I decided to start with the prescription entry object, since the file object was too high-level and the patient entry object contained a multiple pointer (prosthesis), which I wasn't certain about.

Implementation, Part I

It was still day one of the project, and I was already writing the first lines of code! The database objects seemed fairly straightforward, at first. The requirements of reporting meant looking up data, but not modifying it.

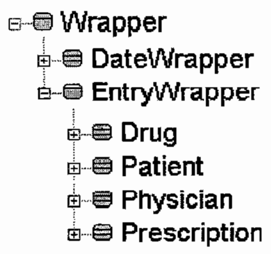

Figure 5

According to the project requirements, I only had to support a few of the fields in the prescription file. I quickly created a prescription object and a property for each field. In testing the object, I noticed that several of the fields (patient, drug, physician) were pointers to other files, and that one field (date issued) was a date. These fields returned raw data (pointer numbers and raw date fields), so I modified my design. The class tree in **EsiObjects** is shown in figure 5 above. At first the class-structure was flat; later I used *drag-and-drop* to create the hierarchy shown above. This is another illustration of the bottom-up style.

This required me to do some extra work—I had to create objects for the physician and drug files and a special date object to represent a raw date value. These changes took longer than expected, so I only provided one or two properties for each of the new objects. I figured I could always go back and flesh them out later, as needed.

It might seem odd to make a special kind of object just for dates. But the raw date form was not easy to understand, and the date object supported operations like "Short Date," "Long Date," "Month Name," etc. This makes it easier for every other object in the system to work with date fields, as the following sample code illustrates.

Standard M Approach

```
SET RAWDT=$PIECE(^PRE(IEN,0),"^",6
SET LONGDT=$$LONGDATE^DTCONV(RAWDT)
```

EsiObjects™ Approach

```
SET Scrip=PrescFile.Entries(IEN)
SET LongDate=Scrip.DateIssued.LongDate
```

(Note how much more *self-documenting* the second approach is.)

On day two the prescription object was completed: I was able to create a prescription object, tied to an entry in the prescription global. When I asked it for the drug prescribed, it returned a drug object, representing an entry in the drug file. When I asked for the prescription date, it returned a date object, which supported operations like ShortDate ("Aug 15, 1997"), LongDate ("Friday, August 15, 1997") and HVal (representing the raw $H, e.g., "57205," sometimes useful in interval calculations).

The next task was the patient object. It had its own date field (DOB), so I simply *re-used* the date object I had created for the prescription file. It also had a multiple field prosthesis, which contained a pointer to the prosthesis file. I decided to represent the multiple property as though it were an array of prosthesis objects. This took further time to implement and ultimately required a new kind of object to represent sub-file entries.

Design Modifications

Having implemented some objects, I decided to revisit the initial design sketch. I now realized that I would need to develop a broad strategy for managing relational jumps between files, and that this strategy would affect both the user interface and the communications between the remaining objects.

I decided to flesh out the report process in a little more detail. Ignoring the application itself, I focused on the interactions between the report generator and some other objects.

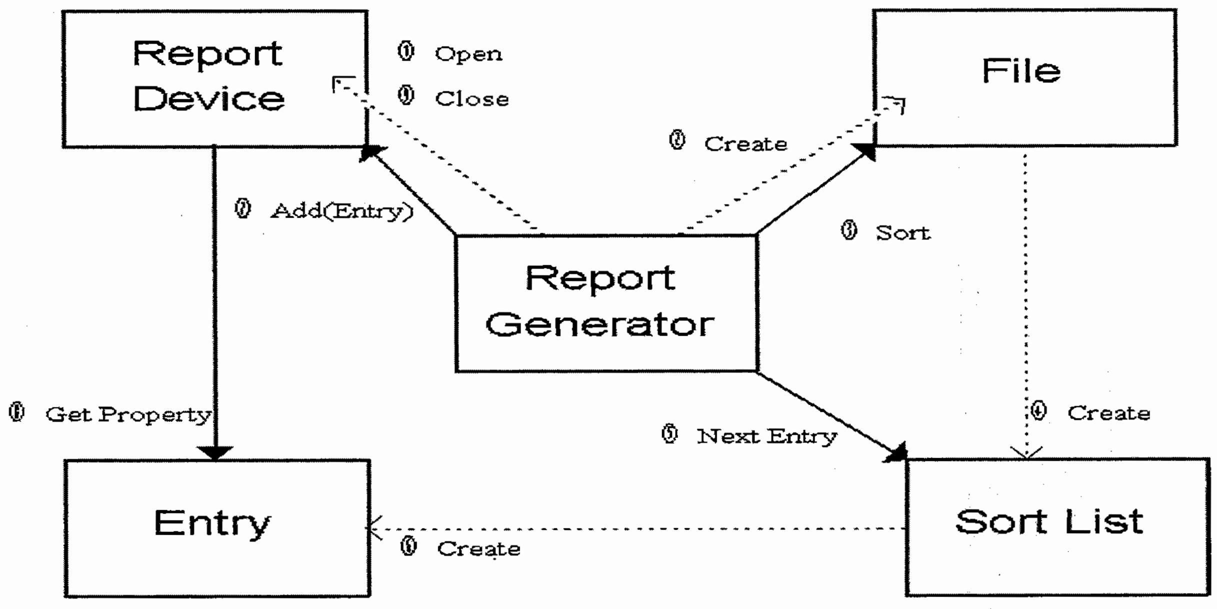

Figure 6

I decided that a **ReportGenerator** object would be responsible for driving the reporting process.

1. The **Report Generator** *opens* a **Report Device** object to manage output.

2. It then *creates* a temporary **File** object to represent the global database.

3. It asks the **File** object to *sort* its entries and gets back a **Sort List**.

4. The **File** may need to perform sorting, or may be able to use an index in the global. Either way, it sets up the **Sort List** to provide the objects in the right order.

5. The **Report Generator** repeatedly uses the *Next Entry* method to obtain an entry from the **Sort List**.

6. The **Sort List** responds to the *Next Entry* method by *creating* an **Entry** object to represent a single entry in the file (e.g., a single patient's data).

7. This entry is then *added* to the **Report Device**.

8. The **Report Device** subsequently queries the entry for the *properties* in the report. Each property value is placed into an output column of the report.

9. The **Report Generator** *closes* the **Report Device**, ending the report.

By the way, I made a *critical design error* at this stage. I overlooked the fact that multiple fields would have an impact on the top-level design. It seems obvious now, in hindsight, but the approach outlined above simply doesn't work.

The problem: later on, I noticed that sorting by a multiple field caused the same object, with all of its multiple sub-entries, to appear too many times in the report. *The bottom line was that I later had to re-design the top-level report process.* But since the most critical sections of code had not yet been written, the impact of these changes was quite limited.

Continuing Implementation

Without dragging the reader through the entire implementation process, here is a summary of the remaining steps:

• The file and sort list objects had to be implemented. The file permitted sorting by any field (indexed or not), so it provided the one-time service of sorting all entries in a new way, if needed.

• The report device organized output into columns, for display on any kind of device (terminal window, printed page, scrolling window, etc.)

• The report generator coordinated the top-level report process by integrating all the other objects. It drove the entire reporting process.

• The report generator, file, and report device had to be enhanced to allow multiple fields.

• A problem with multiple fields (mentioned above) necessitated a re-write of the report generator and enhancements to several other objects. However, surprisingly little code was lost—in most cases new features were added without removing old code.

• A code review yielded the comment that some of the methods were too large. They were broken down into smaller modules. This resulted in significantly greater clarity and reusability.

• The front-end application was not part of the demo and is still unfinished. When complete, it will make the report generation process much more intuitive. I was actually surprised that the application was developed *last,* but in hindsight it makes sense.

Conclusions

This project caused me to draw the following conclusions:

• It is very *freeing* to admit you don't know what the final application will look like. It really takes the pressure off.

• When unexpected things happened, they seemed more like opportunities than threats.

• I never got the feeling that the application was a "prototype." I think most of this work would still be useful in a full-blown system.

It is important to note that this example, like most case studies of the design process, is not conclusive. It is nearly impossible to develop a fair comparison of two competing design approaches, because there are so many variables to control. Nonetheless, the bottom-up design approach mirrors the evolutionary development path of many real-world systems.

The bottom-up approach succeeded in this case and was fun to use. If this is the case, then why devote a huge amount of time to a rigorous "design phase?"

Modern systems are becoming too complex for top-down design to be useful much longer. Imagine trying to flowchart the human brain. It would take thousands of years, and where would you even begin? Instead of trying to hide our own ignorance, we should warmly embrace it. Instead of pretending to know all the answers, we should be constantly vigilant for innovative new ideas. Instead of trying to design the solution before writing any code, we should begin by implementing the most well-defined parts of the problem. Rather than trying to develop a grand scheme that anticipates everything in advance, we should always maintain a flexible approach that will maximize future reusability. Instead of starting at the top, we should start at the bottom, and work our way up! ***M***

Erik Zoltan teaches subjects including M and EsiObjects Programming for ESI Technology Corp. Visit www.esitechnology.com or send e-mail to ezoltan@esitechnology.com.

Coming Soon!

MSources, your Membership Directory and Resources Guide

CPU

The successful placement of **M/MUMPS** programmers at all levels has been our exclusive business for over 6 years.
We place in both Contract and Permanent positions, Nationwide.

Our service is:

Consistent, Professional, Up–to–date

Please contact **Cécile Marlowe**

(800) 600-3506
Fax: (619) 755-3066
E-Mail: jobs@cpumumps.com

Career Professionals Unlimited
424 Stratford Court, Ste. A32
Del Mar, CA 92014

Browse our Website at:
http://www.cpumumps.com