What is Optimum?

by Don Gall

Introduction

A number of years ago, I heard a presentation from a gentleman who had spent about 40 years working in the fields of optimization and optimal design. He made the statement that "anything is optimal, as long as I get to decide the criteria used to determine what is optimal." His comment seems to be a generalization of the "beauty is in the eye of the beholder" theory.

Engineers and programmers, whether they understand it or not, are in the business of attempting to create optimal designs. It is important to understand that what is optimal to one person may be unacceptable to someone else. How many "nearly perfect" programs have you seen rejected by the first person who attempted to use them?

It is also obvious that what is optimal at one time may be entirely unacceptable at some later time. Try selling a roll and scroll user interface today.

Let me place a disclaimer here. If you think that this article will give you a methodology which will allow you to take an application, run it through an algorithm that will return a numerical grade and then tell you how to improve that grade, move along to the next article in this journal right now. The goal of this article is to get you to think about the positive and negative ramifications that programming decisions have on the application as a whole. The really good programmers among you already do this and may also move on to the next article. The rest of you should keep reading!

Old and Modern Examples

Several centuries ago, the British set out to get tea from India to London. The optimal design for the containers was a cube. This minimized the amount of tin needed to enclose a given volume of tea. Although a spherical design would have used less tin, the cost of manufacturing a spherical container would have been excessive. The captains of the ships hauling the tea might also have objected to having all those balls of tea rolling around below decks. Thus, the best trade-off was a cubic container.

Many years later, we walk into a supermarket to buy Wheaties. Why are they now in these skinny rectangular packages? Cubic containers would save them millions of dollars annually in the cost of cardboard alone. However, the value in advertising for the big frontal area outweighs the increased cost of the cardboard. Thus, the optimal design is a skinny rectangular package. If the tree-hugging lobby starts a boycott of breakfast cereals in skinny rectangular packages, remember that you read it here first.

And please don't ask me why tea is still sold in nearly cubic containers. This is my story and I'm sticking to it!

The DOHS Theory of Design

Back in the 1800's Oliver Wendell Holmes wrote the poem "The Deacon's Masterpiece," which reads in part:

> Have you heard of the wonderful one-hoss shay,
> That was built in such a logical way
> It ran a hundred years to a day?

For the benefit of those of you who live east of the Mississippi, a number of us in the hinterlands of Arizona continue to ride a hoss from time to time. The poem tells the story of the Deacon's one-hoss shay that ran perfectly and without repair for exactly one hundred years and then totally disintegrated.

The Deacon's One-Hoss Shay (DOHS) theory of design is truly an optimal design.

In these more modern times, we have an automobile that has oil that lasts 3000 miles, tires that last 30,000 miles, a body that lasts 6 years, an engine that lasts 150,000 miles and a driver that can only go 2 hours between rest stops. On the other hand, we have been able to create computer programs that the original programmer never thought would last until the year 2000.

A more common, and usually less quantifiable, term for the DOHS theory of design is "trade-off." How much of result A are you willing to give up to get more of result B? In most situations, we would be happy if there were only two opposing criteria. More often than not there are at least N such interrelated criteria. In most cases, we would be sorely pressed to accurately determine the value of N.

In an earlier life, I worried a lot about ways to quantify a so-called optimal design. I was a college professor at the time, so you can guess what I did. My personal optimal solution was to write a number of papers about the subject so that I would be promoted!

The problem of attempting to quantify things that are inherently not quantifiable can be expressed simply as the problem of comparing apples with oranges. Even though there is a federal statute that prohibits doing this, you can compare them in a number of quantitative ways:

- How many apples does it take to contain the vitamin C of an orange
- How many oranges or apples can you buy with a dollar
- How many of your apples will you give me for my six oranges

The answers to these questions depend upon many factors. A few of these are:

- The variety of the apples and oranges in the example
- The time of the year
- The part of the country or what country we are in
- The particular store we are in
- Your personal preference for apples and oranges

The DOHS methodology suggests that we rank each of the multiple criteria on a scale from 1 to 10, with 10 representing the best possible outcome. You then ask the questions:

- From 1 to 10, how would you rank having a dozen oranges
- How many apples would it take to produce the same ranking

This tends to take the bartering element out of the question, "how many apples will you give me for my six oranges?"

A specific solution to a given design problem will result in a ranking value for each of the design outcome criterion. The DOHS principle states that the optimal design is one in which the rankings of all criteria are equal and that this resultant ranking is the maximum for all sets that produce equal rankings. This is not always possible. However, in many design problems, it does provide a quantitative way to compare attributes that are difficult to quantify.

One of the more interesting aspects of this approach is that it tends to become an iterative process. After seeing the results of the process, the designer tends to alter the ranking algorithm of one or more of the criteria. In other words, having seen the results, the designer changes his mind about what is really important.

To the uninitiated, this may seem to be a stupid and fruitless endeavor. However, as in many other areas, the thought process change that is produced may be more important than the quantitative results obtained.

The following sections contain some real-world examples from the software development industry and attempt to show how the DOHS principle may apply.

Functionality, Ease of Use and the Ability to Implement

Three of the generic goals of an application are:

- Maximize the power or functionality
- Make it as easy to use as possible
- Implement it as simply and quickly as possible

In most applications, these three criteria are at odds with one another.

One of the common complaints of almost any programming staff is that this new request from the client will be very difficult to implement. This is usually followed by, "Why do they insist on having it that way?" and "It would be much simpler to program it this way." In some of these cases, we return to the client with two proposals that both result in the same functionality. If we do it the way you proposed, it will cost you $4,000, but if we do it this way it will only cost $1,000. This is usually a very effective way to quantify a trade-off decision.

In other cases, we find the client's complaint, or suggestion depending upon how you look at it, will serve to enhance our product. Although our implementation cost may far exceed our charge to the client, we may elect to do this recognizing that the additional sales revenue will make us more money in the long run. In this case, we have made the financial trade-off decision.

At the MTA Meeting in 1996, one of the M implementers on a panel stated that if the M community were willing to give up the `$Order` function, strange hierarchical structures, the `eXecute` command and a few other items, it would make the implementation of M much simpler. The trade-off here was obviously not apparent to the speaker unless he had already found a job in a different industry.

Reusable Code

The concept of reusable program modules also has its trade-offs. One advantage is that we need to write and debug a module once. The criterion of minimizing implementation time gets a high ranking here. A second criterion that we must look at is the issue of performance. Will the use of this module significantly decrease the performance? The ranking on the performance criterion will depend on how well the programmer writes the module and how it is used.

Having the same or nearly the same blocks of code scattered throughout your software makes it hard to maintain. After a period of time, these similar blocks tend to get out of sync as changes are made to some but not all. Thus, a single, well-designed function that is called by a number of routines is far simpler to deal with and gets a much higher ranking for maintainability.

Almost everyone has good examples of how reusable code has been used in their applications. The single best example of reusable code in our OOP application is an inquiry function that gives us encapsulation support. The function uses the data dictionary that is an integral part of the definition of a class to return values of attributes from that class and may be called for any class in the application. The function is called by:

```
SET OUTPUT = $$^UINQ(Class,
Attr_Suppl, Attr_Values, Attr._Req)

Where,
  Class = the identifier for the
     class containing the informa-
     tion
  Attr_Suppl = the identifiers
     for the required class attri-
     butes
  Attr_Values = the values of
     each of the required class
     attributes
  Attr_Req= the identifiers
     for the requested class attri-
     butes
```

As an example,

```
SET OUTPUT = $^UINQ("AC","CLN",1234,
"CLNAME~ZIPCODE")
```

will return the variable OUTPUT equal to the Client's Name and Zip Code separated by the delimiter tilda, '~,' given the Client Class, AC, the Client I.D., Attribute identifier, CLN, and the value of the Client I.D., 1234.

The only negative aspect of using this inquiry function is that it requires more disk accessing and therefore its overall performance is inherently slower than the equivalent M code that would have directly accessed the global structure.

There are several positive aspects:

- Encapsulation is supported, thus changes to the data structure of Class AC will not require changes to the calling program

- The programming of the calling program is simpler

- The implicit documentation of the function in the calling program is clearer
- The Windows version of our software can populate screens very simply by a call to the same function

This example of reusable code causes us to sacrifice some computer performance in exchange for enhanced maintainability and decreased programmer time. This one function decreased the programmer time required for our Windows product by about 4 man-months.

There are so many examples of programs that have underutilized reusable code that there is no real reason to add any of ours to the list.

Is it possible to get too carried away with the concept? Of course! Let us now look at an example of too much use of reusable code. A number of years ago, I saw an application which contained a subroutine for printing a number on a report. Over a period of time, this subroutine became more and more complex with the addition of the ability to round the value, to print round, square, angular or curly brackets around the value and a number of other ideas that various programmers thought were "cool." The programming staff had been thoroughly indoctrinated about the virtues of reusable code and used this subroutine religiously. One particularly lengthy report with about eight columns of dollar values took about four hours to produce using this subroutine for each printed value. Replacing each of the subroutine calls with a simple M expression to write the value right adjusted with two decimal points reduced the running time to under an hour.

I have seen a subroutine in which the only M code was:

```
Read !,"Enter > ",X Q
```

No more need be said about this example.
Between the obviously good examples and the obviously bad examples are some questionable examples. When we began the development of our Object Oriented Programming application about 5 years ago, we decided to have an extrinsic function that would return the date for display or printing in a uniform format. The function accepted a date in YRMODA format and returned it in a MO-DA-YR format. The performance would have been improved had we used the `$E(DATE,3,4)_"-"_$E(DATE,5,6)_"-"_$E(DATE,1,2)` shown in most M texts. However, using the extrinsic function completely eliminated the Year 2000* problem for our application.

At one point, we began development of a generalized update function similar to the inquiry function. This would allow us to update data in any class from any other related class. It became apparent early on that the function was going to become so complex and its performance so miserable that its use would not offset the decreased programming time that we would have gained. Thus, we abandoned this otherwise "superb" idea.

Redundant Data in a Database

For the purposes of this section, let me define redundant data as one or more totals that could be obtained by adding or counting entries in the database. An example is the total amount owed by a customer which could also be determined by adding up all of the unpaid invoices.

This is a surprisingly controversial subject. The RDBMS purist insists that having redundant data in a database is never acceptable. Most of us have seen the results of this philosophy as the computer churns through a large volume of data to determine the current balance in a checking account. The performance ranking for accessing often-needed information for these systems is not good.

The major advantage of redundant data is to provide

* As a Y2000 aside, we do not use $Extract functions to get the year, month and day. Instead we use:

Day	=	DATE#100
Month	=	DATE\100#100
Year	=	DATE\10000 + 1900

In our application, the day following 991231 (12-31-99) is 1000101 (01-01-2000). At some point in the 21st century, people will want to see dates that appear as 03-15-02 instead of 03-15-2002 and 09-12-1997 instead of 09-12-97. We will need to change only one extrinsic function to accomplish this throughout our system.

rapid access to totals or counts which would otherwise require substantial computations or data accessing.

There are three disadvantages of having redundant data:

- Hardware or software failures can cause the redundant data to no longer agree with the primary source of these data
- Additional programming is necessary to ensure that the redundant data are maintained
- Additional programming is necessary to verify that the redundant data are correct

We have all seen systems in which reports from different sources produce different results.

When should an application have redundant data? Our rule of thumb is that if the end user has the need to repeatedly get at specific summarized information that would otherwise require a great amount of computer time and resources, we should provide this information as redundant data. We feel that a good example is our Summarized General Ledger Class that contains both the General Ledger Chart of Accounts and a summary by year and month of the beginning and ending balances and the activity by journal for each account. Our clients have immediate access to both the current balances and activity in every General Ledger Account as well as for prior months and years.

We have a single extrinsic function used by all of the journal classes to update this class. (Another plug for reusable code.)

As a bad example, we maintain a number of working, billing and income statistics by the combination of seven to nine different parameters because we were assured by a number of our clients that they would use this information all the time. Having ignored the fundamental programmer axiom of not believing anything that is preceded by the words all, always or never, we implemented this idea. We pay a large price for maintaining this structure and find that our clients almost never use it. It will disappear in our next software version.

Summary

Whether you are designing an entire application, a module in that application or one program in a module, your goal is to create the best design that you can. The issues of functionality, ease of use, modularity, performance, maintainability and adaptability to future requirements must always be considered at every level of the design. In general, these present the designer with conflicting criteria. But that is the very essence of design. If it were not for the need to make good trade-off decisions, everyone would be a great designer.

In many ways, writing computer software is a unique and challenging occupation. It is part mathematics, part science and part art with a whole lot of logic thrown in. The structures, concepts and expectations are changing so rapidly that it is impossible to keep an entire application on the leading edge. It is amazing how often I look at a program written two or three years ago and ask, "what blithering idiot wrote this?" Then, I see my own initials at the top of the program.

For some reason, I doubt that Goethe ever read one of his earlier works and said, "Welcher geblitheriner Trottel hat das geschrieben?" M

Don Gall is CEO of Omega Legal Systems in Phoenix, AZ and a member of the MTA Board of Directors.

MTA Nominations Open
1998 Board of Directors

Open two-year positions:
Chair
Executive Director
Two At-Large Seats

See page 27 for more details.