

A Technique for Building Robust, Maintainable Applications

Avoiding Logic Errors

Dealing With Data Errors
Tables as a Communication Mechanism

Tables are frequently used simply to reduce disk overheads and get large performance improvements. This is, however, by no means their only application. Efficient memory resident tables also make possible a table-driven approach to application design with considerable benefits in program robustness and maintainability. That is the topic of this document.

Using tables to get application logic out of programs

Quite a few programs contain a lot of logic for dealing with various business cases. Consider this example, in pseudocode, from an imaginary retail application:

```
if account status is bad
    refuse credit
calculate
    total = balance on account + cost of purchase
if total <= credit limit
    if no overdue balance
        allow purchase
    else if old customer and overdue balance less than $50
        allow purchase
    else if old customer and overdue balance more than $50
        refer to manager
    else if not old customer and overdue balance less than $50
        refer to manager
    else
        refuse credit
else
    total is over limit
    more if-else stuff goes here
```

Such code has a number of problems.

Making rules explicit

First, it does not make all the rules explicit. Consider someone who is within his credit limit, is not an old customer, and has an overdue balance of, say, \$65. There is no rule along the lines of:

```
else if not old customer and overdue balance more than $50
    do something-or-other
```

so this case "falls through" and is caught by the last else. The code above produces exactly the same results that it would if we added the rule:

```
else if not old customer and overdue balance more than $50
```

refuse credit

That being the case, most programmers would not code such a rule. This is a mixed blessing. Judiciously used, such things help produce efficient programs and the code involved is entirely readable, at least to programmers. Overdone, the same techniques can create logical constructions so obscure that even the perpetrator cannot untangle them later. In any case, they often make the logic hard for non-programmers to follow.

Contrast this with table-driven code where the key describes the input conditions. One design for our example would use four Yes/No fields for:

- over limit
- old customer status
- overdue balance exists
- overdue balance over \$50.

The rest of the row holds data on what to do, perhaps "accept", "refuse" or "refer" strings. In such a design, you simply build a 16-row table with all possible keys covered. All rules are explicit. The table might begin:

OVER LIMIT	OLD CUST.	BALANCE	BAL > 50	RESULT
N	N	N	N	accept
N	N	N	Y	error
N	N	Y	N	refer
N	N	Y	Y	refuse
N	Y	N	N	accept
N	Y	N	Y	error
N	Y	Y	N	accept
N	Y	Y	Y	refer
... and 8 more rows with Y in the first column.				

Every case is covered by an explicit rule and you do not need to be a programmer to see what rule applies.

Making rule sets complete

Notice also that one possible result is "error". This is returned when the input is inconsistent, "N" for existence of an overdue balance but "Y" for overdue balance greater than \$50. **Table-driven design encourages more thorough error-checking.**

Avoiding logic errors

Second, such code can be a little tricky to write and **bugs can easily creep in.**

The code above, for example, handles an overdue balance correctly if it is either less than or greater than \$50, *but not if it is exactly \$50*. The rules, as implemented above, for an old customer who wants to make a purchase within his credit limit (total \leq limit) but who has an overdue balance are:

```
...
    else if old customer and overdue balance less than $50
        allow purchase
    else if old customer and overdue balance more than $50
        refer to manager
    ...
    else
        refuse credit
...

```

- if overdue balance $<$ \$50, allow the purchase
- if overdue balance $>$ \$50, refer to manager
- if overdue balance = \$50, refuse credit
(because the code "falls through" to the last "else" when no other case applies)

Refusing credit in this last case makes no business sense; surely you should either allow the purchase or refer it to the manager.

Some people will spot the error in the code above easily, much as some will instantly notice spelling or grammar errors even in a complex document. Even among programmers, however, this talent is not common. Good proofreaders are hard to find, for any language. Putting data in tables, rather than in program code, both removes irrelevant complications, making errors easier to see, and allows non-programmers to have a look.

In this example, a **programming slip causes a business error** which will *almost certainly lose a sale and could easily lose a customer*. The error is small enough to be easily made, (though good programmers will usually avoid it). The circumstances that trigger it are unusual enough that it might slip through testing, (though good testing procedures concentrate on "boundary cases" and might well catch it).

The software will consistently repeat that error every chance it gets. In the example, the error-triggering combination of circumstances won't come up very often and the consequences of the error are not horrendous. (Or may not be, depending which customer it offends.) Other programming slips, however, can cause errors which are more frequent and/or more costly. Anything is possible, right up to getting something wrong every time or, in some cases, making an error that destroys the business the first time. Of course, really serious errors are both less likely to be made and more likely to be caught in testing, but they are possible.

Contrast this with a table-driven design which makes it **impossible to commit that type of error**. Table look-ups **cannot fall through to an unintended result** as the example code above does. If your table data does not cover some case, then a table lookup operation returns "not

found". The program sees that and does something sensible; the question of fall through never arises.

Of course a programmer could still get it wrong by failing to check for the "not found" result. The design approach protects you from a number of all-too-common programming slips, but it cannot help with this programming blunder.

In our example, the suggested table has a Y/N field for overdue balance less than \$50. You will get a "not found" if it is set to anything but "Y" or "N", which protects you against many errors. Consider the possible ways the programmer could set it.

- He could make the same error that was in the original example code, setting this field with:

```
if overdue balance less than $50
    set field to "Y"
if overdue balance more than $50
    set field to "N"
```

The error is more obvious here than in the original code, in fact so obvious that in practice it is unlikely to occur.

- Much more likely, he would write either:

```
if overdue balance less than $50
    set field to "Y"
else
    set field to "N"
```

so a \$50 balance is treated as $< \$50$ and credit given.

- or

```
if overdue balance more than $50
    set field to "N"
else
    set field to "Y"
```

so a \$50 balance is treated as $> \$50$ and the customer referred to the manager.

Either of these produces a sensible result when the balance is exactly \$50. The first treats it as less than \$50 and the second as greater, but *neither of them refuses credit* as the original code did. You may not get exactly the result you want, but at least it is not utterly wrong as the original code's result was.

Moreover, the programmer knows perfectly well that he can code it either way, so he's likely to ask for a clarified specification. If he does, the client defines the correct result and the program delivers it. This sort of request for clarification should, and sometimes does, also happen with

non-table-driven programs. However, the programmer is more likely to notice the problem in a table-driven design. The code is simpler, making problems more visible, and he is explicitly setting a flag variable for the answer so he is forced to consider the question.

Dealing with data errors

You can get the data in a table wrong, just as you can get the logic in a program wrong. Table-driven design, however, **makes errors easier to avoid, detect, and fix** in several ways:

- Many data errors produce "not found" results.
An explicit error is more easily handled than an unanticipated fall-through.
- The data is more easily viewed and edited in a table than when it is embedded in code.
There are fewer complications; you can see just the data.
- Data in tables is accessible to non-programmers.
In particular, business data can be read and altered by the business experts.
- Data in tables is accessible to programs other than its primary users.
Various auditing, monitoring, consistency-checking and reporting programs are possible and can be deployed as required.

In short, table-driven design leads to **simpler and more reliable programs**.

Making rules visible

Third, when business rules change the code must be changed. Bugs may of course creep in whenever code is changed, but that is not the central problem. The real worry is having to change code *at all* to implement changes in business rules.

For example, what if a business manager decides to change one of the rules embedded in the code above? He or she wants a simple change, on something within his or her authority. From the computing side, it is also pretty simple. All you do is change a few lines, re-compile the software and install the new version.

But exactly who does that? Aye, there's the rub! *The business manager can make this decision, but neither he nor his staff can implement it.* That requires, at the least:

- a programmer to make the change and recompile
- a system administrator to install the new version

and in many organizations, there will be additional complications:

- Software test folk should check any changed program before it goes into production
- Documentation may need to be changed
- Everyone involved has other demands on his or her time, and a manager who may influence the schedule
- Various misunderstandings are possible when the decision maker and the implementer are separated geographically and/or organizationally.

- Some misunderstanding may be almost inevitable; the programmer often doesn't know the business, and the manager is unlikely to understand the code.
- Trying to avoid misunderstandings, you get paperwork: change requests, records, ...
- This means more work for both client and programmer.
- Likely also for support staff, who also have overloads, conflicts, managers, ...
- Conflicts of interest and shortages of resources come into play; the business manager and the programmer's manager are likely to have different priorities.
- In many organizations, some delay is inevitable.

Contrast this with the situation where the business manager makes the decision, one of his staff changes a table entry, and that's done. The whole process is within one manager's purview and, more important, *it is the right manager*. Of course this is faster and cheaper than the alternative, but the really big payoff is that it lets us build systems of applications in **which changes of business rules are controlled by the group responsible for the business**.

Meanwhile, the programmer goes on developing the next version, or doing something else, without even being aware of the table change. He and his manager are happy. There is less "maintenance programming" of the type that consists of just altering code to implement changes decided on elsewhere, so there is more time for everything else.

Tables as a communication mechanism

Finally, even in table-driven system not all changes can be made just by altering tables. The business folk and the software development department do still need to talk to each other, and to co-operate on the design of the next version of the software. Often tables provide a common language which is extremely useful in such discussions.

The business people may not understand the code, and the data processing types may not understand the business, but they can both understand the tables.

Consider our example table again:

OVER LIMIT	OLD CUST.	BALANCE	BAL > 50	RESULT
N	N	N	N	accept
N	N	N	Y	error
N	N	Y	N	refer
N	N	Y	Y	refuse
N	Y	N	N	accept
N	Y	N	Y	error
N	Y	Y	N	accept
N	Y	Y	Y	refer
... and 8 more rows with Y in the first column.				

This should be far clearer to a business manager than code would be. It may well be clearer to a

programmer than someone else's code would be. This becomes extremely important in maintenance programming, when much of the work involves someone else's code.

It is almost certainly considerably clearer to the programmer than the business manager's initial statement of the problem would be. Perhaps no clearer than the specification the manager might write after some analysis, but no worse either, and likely less work to produce.