Paper 15-26

Preemptive DATA CLEANING: Techniques

Malachy J Foley

University of North Carolina at Chapel Hill, NC

ABSTRACT

Data cleaning is a necessary evil... Or is it? Many techniques exist that prevent data errors at data gathering and/or at data input. The use of these techniques eliminates or minimizes the errors that actually get into your computer files. One such technique is check digits. Check digits can eliminate up to 99% of all transcription and keying errors. Other techniques include proper form design, data monitoring, double keying, and hash totals. This tutorial examines these error-prevention techniques through examples and where applicable, it includes SAS code.

INTRODUCTION

Since "to err is human", some people think that to have errors in data files is inevitable. Yet, in some applications clean data is critical. For example, in banking, it is not acceptable to debit a wrong account, or credit an account with an incorrect amount of money. In applications such as statistical surveys, having clean data is less critical, but still very important. In fact, dirty survey data generally weakens and biases statistical results.

This paper is based on the philosophy that erroneous, missing and duplicate data needs never to get into a computer file. Furthermore, it is easier, faster, and less expensive to avoid errors at the outset than to try to get rid of them once they are in a computer file.

Programmers have a vested interest in data quality control for two reasons. First, they are often called upon to program quality checks into data entry systems or into computer file checking systems. Second, many of them spend an inordinate amount of their time (some estimates are as high as 70%-80%) correcting or working around dirty computer files.

Often, there are many processing steps in getting data from the point of data capture to a computer file. For example, there is form completion, data monitoring, data shipping, data reception, coding, and data entry. Each of these steps provides at once an opportunity to make an error and an opportunity to embed error-avoidance methods into the procedure.

The trick to quality data is to overwhelm the errors with error-avoidance techniques.

As an example of errors, consider what happens at data-entry time. The data collection form can have faulty skip patterns or scan patterns (see the glossary), which can confuse the person keying the information into the computer (the keyer). The information written on the form is often illegible or ambiguous. Furthermore, the keyer can read a character incorrectly or press the wrong key. Frequently, keyers transpose two adjacent characters or repeat the wrong adjacent repeated characters (ex: key 52<u>66</u>91 instead of 52<u>69</u>91).

Fortunately, there are a variety of techniques used to counter such sources of errors. Some of these techniques are proper form design, pre-printed labels, checking allowable ranges of values for a field, checking for inconsistencies between fields, double keying, hash totals (see glossary), and check digits.

This paper reviews many such error-avoidance or quality-insurance methods. Special emphasis is given two techniques: form design and check digits.

There is a glossary at the end of this article that defines terminology and provides additional information about data checking.

FORM DESIGN

Perhaps the most important component in collecting clean data is good form design. Too often poor form design interferes with or prohibits the proper capturing of data.

How can a form cause errors? Take a name field for instance. Assume that a person's name is requested on the form. The area for the response can be designed in different ways. One way that is frequently seen on forms is to follow the word "Name:" by a line. Such a design is bad news for data entry and data processing. The respondent can and <u>will</u> fill out the line however they please. You will literally get every possible combination of names imaginable. You will get nicknames, formal names, initials or no initials, Jr., III, pre-titles like Dr., post-titles like Ph.D., script writing, printing, and so on. If the response is written in script (cursive), it will be difficult to read. This in turn will lead to errors, extra time in reading the name, and extra cost. Once the names are in a computer file, putting them in alphabetical order would be a tedious, costly, error-prone programming task.

A better way to design a name field for computing is to use a square or a rectangle to outline each letter. Human factors (see glossary) dictate that the minimum size of a square should be 0.25 by 0.25 inches. A smaller square (and you see them all the time) will force people to write too small to be legible, causing errors in the data transmission. Squares cue people to print. Nonetheless, it is advisable to request that the name be printed in capital letters and perhaps even provide them with an example of such print. Each field should be separated: Last name, first name, middle initial, title, and so on. Separate fields make it easy and inexpensive to manipulate the names once they are in a computer file.

Enough squares need to be provided for each field, so valuable data does not get truncated. For example, the author suggests at least 15 squares for a last name. In Latin America where longer names are used, more space must be provided. In fact, rectangles that are at least 0.25 inches wide and somewhat taller (say 0.3 inches) are even better than squares for collecting data.

Finally, as another example of form design, what color should the form be? Well, white paper is fine. White paper is readily available. White contrasts with almost any other color. Black print (on white paper) is another matter. Black will conceal almost any color pen's stroke. Thus, whatever is written with a pen on the form that happens to run over a square or other print on the form will be concealed to the person trying to read the information. This happens all the time. A better choice in print color is a light green. Green will not hide pen strokes written in black or blue (the usual color of pens and pencils).

These are just a few issues regarding form design. There are many other issues such as how to handle decimal points, units of measure, instructions, scan patterns (see glossary), skip patterns, reading levels required to understand questions, etc. It is beyond the scope of this paper to cover all these issues. In fact, books have been written on this subject.

The point of this section was twofold. First, poor form design can and does cause error in data. (In the conference presentation, the author will give actual examples of how insufficient form design causes error.) Second, a small investment in good form design will save you large quantities of time and money in detecting and correcting data errors later on in the data processing trail. The truth be told, no amount of follow-ups, data massaging etc. will fully correct the damage done by poor form design.

INTERMEDIATE PROCESSING

For the purposes of this paper, intermediate processing is defined as all the data processing steps that occur from the time the form is filled out until it is actually entered into a computer file.

With the advent of inexpensive and portable computers, some times no intermediate processing occurs and data is scanned or keyed directly into the computer. This happens all the time at the ATM or when a credit card purchase is made.

On the other end of the spectrum, much intermediate processing can occur. A paper form can be filled out, visually checked, coded, transported to another location for keying, counted, stamped, logged in, acknowledged, etc until it is finally keyed or scanned into a computer. The following paragraphs review some of these steps and how they relate to data quality control.

The first step in intermediate processing is to record data on the form (paper or electronic). If someone is hired for this purpose, they should be meticulous and patient, and have the ability to follow instructions. In the case of paper forms, they should also be able to print clearly. For forms that require keying, they should be able to key with very little error. This paragraph may seem intuitive. However, it is surprising how often haste, poor penmanship, etc. cause errors or missing data.

While speaking of haste, it is good to mention that quality processing in any step along the data trail requires time. When managers and supervisors demand results too quickly, corners will be cut and data quality will suffer. Probably the number one reason for poor design and mis-keyings is haste.

In some incidents, the person filling out the form must also interview the subject (the person having the information). When this happens, the interviewer will need to have social skills. Also special training is usually called for, since all interviewers should ask questions in a uniform and unbiased manner.

DATA MONITORING

Once the data is recorded on the form, the entries can be visually checked for completeness and accuracy. In clinical trials, such checking is customary and the person doing the checking is called the data monitor. In other data collecting situations, to have such a person may be advisable.

The need for a data monitor depends on what other steps exists in the data processing trail and when they occur.

Like data monitors, computers can check for data completeness and check that the specific data values fall into acceptable ranges. However, in some areas humans are better at checking data. This is because humans, unlike computers can have experience with the data being collected, can detect unexpected situations, and can provide judgement.

DATA SHIPPING & RECEPTION

Many times forms are physically or electronically shipped from the data capture point to one or more data processing sites. When data is shipped several techniques can be used to assure that the transmission is completed and that data is not lost.

The most important technique is to copy or backup your data at the point of capture before shipping. Some IT professionals contend that the first law of computing is to "always backup".

Another shipping technique is to maintain a log with a count of how many records were shipped, date sent, tracking numbers, etc. A heads-up message is customarily sent to the receiving site so that they know that the data is coming and what is contained in the shipment.

When the data arrives, several actions can be taken to insure data quality. To name a few, the forms can be counted, date stamped, sequence number stamped, and logged into a book. What was received can be checked against what the shipper said was sent. Also a reception notice can be sent to the sender of the forms to assure them that the forms arrived as expected.

The next section details one of these techniques and how it contributes to data quality.

SIMPLE COUNTS

One way to know that everyone or everything is included in a file is to use a simple count. This technique is applicable to both paper and electronic forms.

For example, assume that each week you are supposed to receive a report from each of 20 regional offices. Further assume that there is one observation or record per report. Before processing the reports, you want to verify that each office has sent in their report. One way to perform this verification is to simply check that you have 20 observations in your file.

While the simple count is a good way to check if a file or report is complete, it can miss some problems. For example, if you have 20 offices, and one office sent in a duplicate report and another forgot their report, you would still have 20 observations. Thus, the compensating errors of a duplicate and a missing report would get past a simple count.

HASH TOTALS

Another way to check that files are complete is to use hash totals. A hash total will detect the compensating errors just described. A hash total is a sum of values on paper forms, or in an electronic file, where the total itself is meaningless (hash) yet useful for checking that a process is correct.

To see how hash totals work, we continue the example of 20 regional offices started in the previous section. Assume that each regional office has an office identification number. In effect, the 20 offices are numbered from 101 to 120. Again, before processing the reports, you want to verify that each office has sent in their report and that those reports are in your file.

This time in addition to doing a simple count you do a hash total. Namely, you add the value of all the office IDs in your file. They should add up to 2210 (101+102+...+120). If they don't add up you could be missing an office's report and/or have a duplicate report or have a wrong office ID in a report. Notice how the hash total actually checks several aspects of the data at once.

Normally, it is important to do a simple count in addition to a hash total. The simple count confirms that all the expected components are present in the hash total.

In last example, a hash total was used to confirm that all the input data was available. In a similar fashion, one can verify that all the output is present in a report. For instance, hash totals can be used in payrolls or billings. In payrolls, a hash total of all employee social security numbers can confirm that every employee has a paycheck.

In billings, the use of hash totals is a bit trickier. First, when creating the invoice records, make a null invoice when a client has a zero balance. Then a hash total of all client account numbers on the invoices will confirm that everyone has an invoice. Null invoices need not be printed.

Finally, the values of hash totals often change each time they are used. In the payroll example, it is likely that some employees have left the company and new ones

have been hired since the last payroll. In this case, the hash total needs to be derived before each payroll is processed. However, inasmuch as the payroll is taken from a master file, it would be self-defeating to calculate the hash total directly from the same master file. Rather, the hash needs an independent calculation.

DATA ENTRY SYSTEMS

Another way to eliminate errors before they get into computer files is to have a very smart data entry system (DES). Such a DES usually runs on computers and makes a variety of checks on the data while it is keyed. Typically a good data entry system checks each field for valid values. Furthermore, it would check for duplicate ID's, inconsistencies in data across fields, hash totals, skip patterns, check digits, etc.

A good DES, just like a good form would be designed to be self-explanatory and easy to use. There would be no awkward keying sequences. Like all software, the DES should be robust and have technical support.

A good DES would stop virtually every machine-detectable error in its tracks. The author has worked in many environments where these kinds of DES's were in use. In most of these environments, the rule was NOT to let any form into a computer file that did not pass the DES checking. Rather the form was sent back to the source for correction. When this rule is implemented, it is amazing to see how quickly the data quality improves on the forms.

KEY FIELDS

Most data processing applications involve assigning one or more key variables to a person or thing. Key variable(s) are sometimes called key field(s) or just key(s). A key is one or more variables used to uniquely identify a data item. To illustrate, a person has a social security number, and an automobile has a vehicle identification number (VIN). Parts have part numbers and accounts have account numbers. These 'numbers' can involve letters as well. For example, a patient's ID could be A073.

While it is important that all data be entered correctly into the computer, it is critical that key variables be correct. In fact, an error in your account number may mean that your deposit is placed in someone else's bank account. A mistake in your patient ID at a hospital may mean you get the wrong medicine.

An error in an ordinary field means that one field is incorrect. An error in a key field means that the whole record will be displaced leading to duplicate and missing records (groups of fields).

One way to avoid key field duplications is to pre-make just one label for each key field ID. For example, the VIN numbers are stamped on metal plates that are attached to your car. Only one plate is made for each number. Thus, it is impossible to have duplicate VIN numbers coming from the factor.

In a similar fashion, ID numbers in other computing situations can be pre-printed on self-adhesive labels, one number per label. If these labels are used on a form, they guarantee that there will be no duplicate ID's. Moreover, since they are computer generated, they will be legible.

KEYING THE DATA

Often it is said, "anyone can key data". While virtually anyone can key data, not everyone can key data quickly and accurately. A qualified keyer must have the right temperament, an eye for detail, knowledge of all the usual keying traps and how to avoid these traps in practice. Once again haste can also make keying unreliable.

The worst case of keying the author ever saw was where a full 8% of all fields keyed were in error. That was one or more errors in every other record. In other words, 50% of all records had an error. In the shop where this took place, the keyers were basically untrained and were paid by the number of records keyed. Thus, the keyers had every incentive to key quickly and no incentive to key correctly.

DOUBLE KEYING

One commonly used method to avoid the data entry problems is double keying. Double keying is also known as verifying. It is the process of having the data entered into the computer twice: once by the keyer and once by the verifier. The keyed data is then compared with the verified data, and differences between the two keyings are resolved. This process works best when the keyer and verifier are different people.

Sometimes verifying is a curse more than a blessing. There are people who think that verifying guarantees that there are no keying errors. But it certainly does not. In the worst case mentioned in the previous section, a full 25% of all records had one or more error in them after verifying!

The best case of double keying the author ever has seen is where 0.01% of all records were correct after verifying. This was where many error-avoiding techniques were used throughout all the data processing steps. The form

was well designed, people were well trained and motivated, keyers were paid by the hour, etc.

Unfortunately, double keying is not a magic bullet. Verifying can only detect and correct some keying errors. There are transcription and keying errors that can get by double keying. For instance, if the first keyer mistakenly interprets a poorly written 8 as a 3, the second keyer (or verifier) may also mistakenly take the 8 for a 3. In this manner, the poorly written "8" slips past the double keying as a "3".

Such slips on critical key fields, like patient ID's, are unacceptable. For this reason, other data-checking techniques are frequently used along with double keying. The most popular of these techniques is the check digit.

CHECK DIGITS

A check digit is one or more digits created for a particular data value that is then attached to the data value. Once attached, the check digit accompanies the data value throughout various stages of processing so that at any of these stages the legitimacy of the value can be checked or validated. Typically one check digit (0-9) is attached to the end of a data value and accompanies the data value throughout every stage of data processing. The check digit is usually calculated based on the number it is checking.

For instance, let's say you have a patient ID of 5682. 5682 identifies a specific person in a hospital, and it is the ID you would usually use. Perhaps it is a sequential number assigned to the patient based on when he or she entered the hospital. Since this patient ID does not contain a check digit, the variable is named SHRT_ID to indicate that it is a short ID that does not contain a check digit.

One way to calculate a check digit is to get the remainder after dividing the number by 7. In this case, you divide 5682 by 7 to get a result of 811 with a remainder of 5. You can calculate the remainder in SAS with the MOD function as follows:

```
REMAIN=MOD(SHRT_ID,7);
```

Thus, 5=MOD(5682,7). The check digit is then attached to the end of the original ID, so the full ID becomes 5682-5, or, if you prefer, 56825. 5682-5, or 56825, is the actual patient ID used by everybody in the hospital. It is written on the forms and it is keyed into the computer.

Here is an example of a Data Step that creates check digits for several ID's (SHRT_ID) with values from 5680 to 5689. This Data Step also attaches the check digit to the short ID to create a complete ID called LONG_ID.

```
Exhibit 1a: Creating Mod-7 Check Digits
---------------------------------------
DATA _NULL_;
   DO SHRT_ID=5680 TO 5689;
      CAL_CD=MOD(SHRT_ID,7);
      LONG_ID=SHRT_ID*10+CAL_CD;
      PUT SHRT_ID= LONG_ID=;
   END;
RUN;
---------------------------------------
Exhibit 1b: Creating Mod-7 Check Digits
            (Output from Exhibit 1a)
---------------------------------------
SHRT_ID=5680 LONG_ID=56803
SHRT_ID=5681 LONG_ID=56814
SHRT_ID=5682 LONG_ID=56825
SHRT_ID=5683 LONG_ID=56836
SHRT_ID=5684 LONG_ID=56840
SHRT_ID=5685 LONG_ID=56851
SHRT_ID=5686 LONG_ID=56862
SHRT_ID=5687 LONG_ID=56873
SHRT_ID=5688 LONG_ID=56884
SHRT_ID=5689 LONG_ID=56895
---------------------------------------
```

HOW TO PUBLISH FULL ID's

Once you create your list of ID's with check digits, the list of full ID's needs to be published so everyone can use them. Typically, the list of complete ID's is sent to the person who will initially assign ID's. Exhibit 2 demonstrates such a list.

```
Exhibit 2: List of Full ID's
----------------------------------------
  56803
  56814
  56825
  56836
  56840
  56851
  56862
  56873
  56884
  56895
----------------------------------------
```

The list should contain many more ID's than the expected number needed. The extra ID's cover mistakes and possible overruns.

A letter or memo often accompanies the list. The letter explains how to assign an ID from the list and it provides other relevant information.

The person that collects data then uses the full ID's from the list when filling out forms.

HOW CHECK DIGITS VALIDATE KEY FIELDS

Once a full ID with a check digit is on the form, the check digit can be used to check the validity of the ID entered into the computer. In the section entitled double keying, we saw how both the keyer and verifier mistook a poorly written 8 as a 3 in an ID. In this example, an

incorrect ID was inadvertently entered into the computer.

Now consider the ID of 5682-5. Assume that the 8 in the ID is poorly written and mistaken for a 3 by both the keyer and verifier. In this case, they would key in the value of 5632-5 or 56325. When the computer reads this ID, it breaks it up and checks it. First, it separates the full ID into its 2 parts (the core ID and the check digit). The computer would get 5632 for the core ID and 5 for the check digit. Then, it would calculate the value of the check digit based on the core ID that was actually keyed. It would find that MOD(5632,7) is a 4. Finally the computer compares the calculated check digit (4) with the keyed check digit (5) and finds that they are not the same. So something is wrong with the ID. Thus, while the poorly written 8 slipped past both the keyer and verifier, the computer catches it.

In SAS, this process can be accomplished with the following code when the ID is a numeric variable.

```
Exhibit 3: SAS to Validate Check Digits.
----------------------------------------
DATA _NULL_;
   LONG_ID=56325;
   SHRT_ID=INT(LONG_ID/10);
   KEY_CD= LONG_ID-(10*SHRT_ID);
   CAL_CD= MOD(SHRT_ID,7);
   IF KEY_CD ne CAL_CD THEN
    PUT "*** Err in ID " KEY_CD= CAL_CD=;
RUN;
----------------------------------------
*** Err in ID KEY_CD=5 CAL_CD=4
----------------------------------------
```

When the ID consists of digits (0-9) but is stored in a SAS character variable, the character variable can be changed to a numeric variable using the INPUT function. Then the above algorithm could be applied to the numeric variable.

Of course, check digits can be more sophisticated. For example, if you have a key variable with an alphanumeric value (like A073), you can convert the letter to its position in the alphabet and then calculate the check digit. Thus, A073 becomes 01073 and L397 becomes 12397 before the check digit is calculated.

The following code converts a letter to a number representing its place in the alphabet (1-26). This code is written so that it is case insensitive and can be implemented on both ASCII and EBCDIC-based computers.

```
Exhibit 4: Converting a letter to a digit
-----------------------------------------
DATA _NULL_;
   ARRAY LETR(5) $1 _TEMPORARY_
     ("A","B","c","d","Z");
   DO i=1 TO 5;
     ALPHA=LETR(i);
     PLACE=RANK(UPCASE(ALPHA))
               -RANK("A")+1;
     PUT "*** " ALPHA= PLACE=;
   END;
RUN;
```

Ehibit 4 (continued)

```
----------------------------------------
*** ALPHA=A PLACE=1
*** ALPHA=B PLACE=2
*** ALPHA=c PLACE=3
*** ALPHA=d PLACE=4
*** ALPHA=Z PLACE=26
----------------------------------------
```

OTHER CHECK DIGITS

So far, we looked at one specific check digit, namely the Mod-7 (or Modulo-7) check digit. This check digit is so named because it is created using the modulo 7 (see glossary) arithmetic operation. In other words, the Mod-7 check digit is calculated based on the remainder after division by 7.

Aside from the Mod-7, there are many other ways to generate check digits. In fact, the Mod-7 check digit is just one in a class of the modulo-n check digits.

Another popular check digit is the Mod-11 check digit. This check digit was even implemented in cardpunch and verifying machines in the early 1970's. One drawback of the Mod-11 check digit is that it creates 11 possible values (0-10) and then there is the question of how to represent the 11 values. Traditionally, values 0-9 are represented by themselves (0-9), and the number 10 is represented by an "X" or an "A". Another way to represent these values would be by using 2 check digits, namely 00-10.

Still another possible modulo-n check digit is the modulo-23, which would generate numbers from 0 to 22. These 23 numbers could then be appended to the original number as is or converted to the letters A-W. (When letters such as A-W are used for checking, the term "check digit" is still often used. However, it is more appropriate to call them check letters or "check symbols".)

The modulo-n class of check digits are just one kind of check digits. There are many others types of check digits from the simple to the sophisticated. An example of a simple check digit algorithm is to sum the digits of the original number and take the last digit of the sum as the check digit. Block (see References), on the other hand, is an example of a sophisticated check digit.

POWER OF CHECK DIGITS

There are so many methods for creating check digits that one wonders which one to use. The short answer is to choose the check digit that has the checking power you need and can fit into the resources you can afford.

On the surface the checking power of a particular kind

of check digit seems fairly obvious. If a specific type of algorithm yields 10 possible check digits, the possibility of an error getting past your check digit is 1 in 10 or 10%. If you have 100 possible check digits (e.g. 00-99) the check digits would detect 99% of all errors and lets 1% through. This is true if the check digits were assigned at random.

However, it is not quite that simple. One of the main reasons to use a check digit is to detect keying errors. There are two common keying errors. They are transpositions between adjoining digits (keying 56935 instead of 56395) and repetition of the wrong repeating character (keying 2699341 instead of 2693341). These two errors are so prevalent that they will often pass verifying. A third error, the single digit substitution will sometimes get by verifying, especially when the substituted digit is one next to actual digit on the keyboard.

A good check digit catches these kinds of frequent errors at a better percentage rate than would a random check digit. For example, the modulo-7 and modulo-10 check digits will let transposition errors slip by, but a modulo-11 will not.

It is beyond the scope of this paper to explain why some check digits work better than others are. However, the following table describes the power of several modulo-n type check digits.

Exhibit 5: Power of modulo-n check digits

MOD	CHECK DIGIT RANGE	←DETECTS→ TRANS-POSE	RE-PEATS	SINGLE SUBST	% ERR DETECT
7	0- 6	Some	Some	Some	86%
10	0- 9	Few	Few	Few	90%
11	0-10	All	All	All	91%
23	0-22	All	All	All	96%
97	0-96	All	All	All	99%

The next section gives the pros and cons of various check digits and some recommendations.

CHECK DIGIT RECOMMENDATIONS

When choosing a check digit, the author generally recommends the modulo-23 check digit converted to a letter. It detects all transposition, repetition, and single digit substitution errors. It is easy to program and more powerful than modulo-11. Since the check symbol is a letter, it is distinguishable from the core number. Mod-23 uses little computer overhead and requires only one keystroke during data entry. SAS code to create the modulo-23 check digit is contained in the appendix. The only drawback to the modulo-23 check symbol is that it requires a letter to be keyed instead of a digit. This can slow the keying process somewhat.

The modulo-97 is also a good choice if you are willing to pay for the 2 keystrokes that this check digit requires. It has all the benefits of the modulo-23 check digit and catches 99% of random errors.

The method suggested by Block (see References) is also good. It renders one check digit and therefore one keystroke. However, it requires a lot of programming time, and a little more computer overhead. Block's method is said to detect all single digit substitutions and all transposition errors. It catches 90% of all random errors.

The modulo-7 check digit was employed in this paper to illustrate the use of check digits. While it has many good characteristics for illustration, it is not a good check digit for the real world. It has low power and does not detect all transposition, repetition, and single digit substitutions.

Here are a few more recommendations.

Sometimes you have more than one key field in your file. For example, you have both a patient ID and a hospital ID. When this happens, use some trick so that all the ID's are distinguishable. One easy trick is to have core ID numbers that are in different ranges. Another trick would be to add a digit to the ID to identify the type of ID it is. For example, if the first digit of the ID is 1, then it is a hospital ID, and if the first digit is 2, then it is an ID. The reason to make this distinction is so that you can check that the right ID was placed in the right field.

Finally, the author recommends a simple technique to increase the power of any check digit, namely use a range check on the original number. For example, if the raw ID runs from 1 to 3800, then your ID list would include ID numbers from 1 to 3,800 along with their corresponding check digits. When you verify the check digit, add a range check on the raw ID, to see if it is in the 1 to 3,800 range. This simple range check will eliminate 6,200 or 62% of all possible erroneous keyings.

CONCLUSION

Clean (or nearly error-free) data is possible. The banking industry proves this everyday.

The way to obtain clean data is to properly plan and design the data collection system. Typical data collections systems consist of many steps from the point of data capture to the point where data becomes part of a computer file. Every step can either facilitate data errors or facilitate data accuracy.

Most data processing steps can have built-in data checks.

In addition to good data processing design and checking, the human factor is important. Data quality depends on hiring practices, management style, and corporate culture.

This paper showed, through several dozen examples that erroneous, missing and duplicate data needs never to get into a computer file.

To preempt or avoid errors at every data processing step is easier, faster, and less expensive than to try to get rid of them once they are in a computer file. To quote Ben Franklin, "an ounce of prevention is worth a pound of cure".

GLOSSARY

Check digits – also known as check numbers. A check digit is a check symbol, where the symbol is one or more digits. Sometimes a check symbol other than a digit (ex: letter) is inaccurately called a check digit.

Check symbol – one or more characters, including letters and digits, created for a specific data value and then attached to it. Once attached, the check symbol accompanies the data value through various stages of processing so that at any of those stages the legitimacy of the value can be checked or validated. A check

Data monitor – the person who reviews the responses on a paper or electronic form for completeness and accuracy.

Digit – the integer values of 0 to 9.

Double keying – also known as verifying. The process of entering data into the computer twice: once by the keyer and once by the verifier. The purpose of this process is to detect and correct keying errors. The keyer and verifier should be different people.

Hash totals – the addition or summing of values in a file where the total itself is meaningless but is useful for checking that a process is correct. For example, adding all the staff personnel numbers in a file to check that all the staff is included in the file.

Human engineering – an approach to designing things so that the user is more efficient and less likely to make mistakes. This includes making the articles more convenient, more comfortable, less confusing, less exasperating, and less fatiguing to the user. (See References)

Human factors (HR) – the physical and psychological requirements of human beings that must be considered when designing computer systems and programs in order to make them workable, less error-prone, efficient and easy to use. A field within Human Engineering (see definition).

Key – a slightly ambiguous word. A key can refer to: (1) A key variable; (2) The act of transmitting data via a keyboard; or, (3) A button on a keyboard.

Key field – see key variable.

Key information – see key variable.

Key operator – synonym for keyer. See keyer.

Key variable – one or more variables used to locate and/or identify a record or observation. Also known as a key field, key information or just a key.

Keyer – also know as a key operator or a data entry person. A person who reads information from a paper form and transcribes it into another medium via a keyboard.

Modulo – an arithmetic operation in which the result is the reminder after the first operand is divided by the second operand, e.g. "29 modulo 7" is 1. In SAS the modulo operation is implemented via the MOD function.

Scan pattern – the layout of data entry values on the page of a form. Specifically, the way the eye must pass over the form to find the data that needs keying. A straight-line scan simplifies and speeds data entry, as well as makes missing a field less likely.

Skip pattern – the way data entry values are passed over based on another data entry value. For example, if the participant answers NO to the question "do you smoke" then a series of smoking related questions are passed over or skipped and usually do no need to be keyed.

Verifying – see double keying.

REFERENCES

Block, Hans. (1977) "A New Check Digit Computation Method", *Statistick Tidscript, Vol 5, 413-428.*

Woodson, Wesley E. and Conover, Donald W. (1964) Human Engineering Guide, University of California Press, California.

APPENDIX

The following code (Exhibit 6) creates a modulo-23 check symbol. Since modulo-23 creates 23 numbers (00-22) these numbers are converted to letters so that there is only one character to key. For example, 00 is converted to A, 01 is converted to B, and so on. Furthermore in the following routine the letters I and O are converted to X and Y. This conversion guarantees that the letters I and O are not confused with the digits 1 and 0 during keying. See the Recommendations section for more information.

```
Exhibit 6:
-----------------------------------------
DATA _NULL_;
   LENGTH LETR $1 LONG_ID $5;
   NUM_A=RANK("A");
   DO SHRT_ID=5680 TO 5689;
      CAL_CD=MOD(SHRT_ID,23);
      LETR=BYTE(CAL_CD+NUM_A);
      IF LETR="I" THEN LETR="X";
      IF LETR="O" THEN LETR="Y";
      LONG_ID=TRIM(PUT(SHRT_ID,4.))||LETR;
      PUT "*** " SHRT_ID= LONG_ID=;
   END;
RUN;
-----------------------------------------

     SHRT_ID=5680     LONG_ID=5680W
     SHRT_ID=5681     LONG_ID=5681A
     SHRT_ID=5682     LONG_ID=5682B
     SHRT_ID=5683     LONG_ID=5683C
     SHRT_ID=5684     LONG_ID=5684D
     SHRT_ID=5685     LONG_ID=5685E
     SHRT_ID=5686     LONG_ID=5686F
     SHRT_ID=5687     LONG_ID=5687G
     SHRT_ID=5688     LONG_ID=5688H
     SHRT_ID=5689     LONG_ID=5689X
-----------------------------------------
```

TRADEMARKS

AUTHOR CONTACT

The author welcomes comments, questions, corrections and suggestions.

Malachy J. Foley
2502 Foxwood Dr.
Chapel Hill, NC 27514

Email: FOLEY@unc.edu