OOP + Metadata = Flexibility

Combining objects with meta-data that contains code using those objects can provide easy-to-change solutions

Tamar E. Granor, Ph.D.

Over the last few years, I've found many situations where data-driving code makes an application easier to maintain. But sometimes, data-driving alone can lead to repeated code and more difficult maintenance. In such cases, combining good object-oriented practices with data-driving may provide a better solution.

An application I've been working on has to accept Canadian health numbers. Each province in Canada issues a personal health number (PHN) to each individual to give them access to the provincial health care system. (The application in question doesn't have to accept PHNs from Quebec, so I don't know whether the information below applies for that province.)

The application needs to validate PHNs. While there's no way to ensure that the specified PHN is the right number for that individual, short of checking provincial records, there are two fairly easy tests.

The first is to make sure that the PHN has the right number of digits. The right number varies by province, and a simple look-up does the job.

In addition, each province uses a check digit for the PHN. A check digit is an additional digit derived from the other digits, which confirms that the specified PHN is in the right format.

With VFP's string-handling capabilities, computing a check digit is simple. However, different provinces have different own rules for computing the check digit. In addition, the rules have changed over time, and it's reasonable to assume that they will change again.

So I looked for a general solution. What I ended up was a data-driven class. While you may not need to compute check digits for Canadian health numbers, the basic ideas are more widely applicable.

Computing check digits

Before exploring my solution, let's take a look at the kind of calculation that's needed. For the most part, the algorithms fall into two broad groups. One group doubles some of the digits and then adds them all up, then subtracts the units (the ones digit) from 10 to get a check digit.

For example, New Brunswick has a 9-digit PHN, where the last digit is the check digit. To compute the check digit, you add up the digits in odd positions (digits 1, 3, 5 and 7). Then double the digits in even positions (positions 2, 4, 6 and 8) and add those together; if any of the doubled values is more than 9, add the tens and the ones digits from the doubling separately. Next, add the totals from the odds and the evens together and keep only the ones digit of the result. Subtract that number from 10 to get the check digit. Listing 1 shows an example:

Listing 1. To compute a check digit, New Brunswick adds the odd digits, then doubles the even digits and adds them to the total, including tens and ones digits separately. The ones digit of the total is subtracted from 10 to provide the check digit.

```
PHN without check digit = 23947578

Add digits in odd positions: 2 + 9 + 7 + 7 = 25
Double and add digits in even positions: 6 + 8
+ (1 + 0) + (1 + 6) = 22
Total: 25 + 22 = 47

Subtract ones digit from 10 for check digit:
10 - 7 = 3
Full PHN = 239475783
```

The second approach to the problem involves multiplying each digit of the PHN by a specified value, then adding the results. The total is divided by 11 and the remainder from that division is the check digit.

Manitoba uses this sort of algorithm. It has a 9-digit PHN with the check digit at the end. The multipliers for the first 8 digits are, in order, 29, 23, 19, 17, 13, 7, 5 and 3. Listing 2 shows the same example as **Listing 1**.

Listing 2. In Manitoba, each digit is multiplied by a specified value. Those results are totaled and divided by 11. The remainder is the check digit.

```
PHN without check digit = 23947578

Multiply and total:
2*29 + 3*23 + 9*19 + 4*17 + 7*13 + 5*7 + 7*5 +
8*3 = 551

Find remainder mod 11: 1

Full PHN = 239475781
```

The list of multipliers varies by province using this scheme.

In addition to these two broad categories, there are differences in where the check digit appears in the PHN and in how the check digit is applied. For example, in most provinces where the check digit becomes part of the PHN, it's the last character, but in Alberta, it's the middle character. In some of the provinces that use the MOD 11 type checking, the goal is for the computed check digit to be 0, while others (like Manitoba) make the check digit part of the PHN.

Making it Generic

When I tackled this problem, it seemed clear to me that I needed a class that could hide the details of checking an id, and that I'd need to look up the right province to figure out what algorithm to use. That is, a CheckID method could look in a table to find out how to check an id for a specified province.

The first step in solving the problem was to find the common parts. Among the provinces using algorithms like New Brunswick's, some doubled the odd digits while others doubled the even digits. But they all then added up the digits of the results. So I created a generic MultiplyAndAddDigits method that accepts a character string and an array of multipliers. It multiplies each digit in the string by the corresponding multiplier. It then adds the tens and the ones digits of the result to the running total. The code is shown in Listing 3.

Listing 3. The MultiplyAndAddDigits method multiplies each digit in the PHN by a specified multiplier and adds the results. This is the first step in several of the check digit algorithms, though the set of multipliers varies.

```
PROCEDURE MultiplyAndAddDigits(;
   cID, aMultipliers)
* Multiply digits by the specified factors,
* then add all digits

LOCAL nDigits, nDigit, nResult, nProduct

nDigits = ALEN(aMultipliers)
nResult = 0
FOR nDigit = 1 TO nDigits
   nProduct = VAL(SUBSTR(cID,nDigit,1)) * ;
              aMultipliers[nDigit]
   nResult = nResult + MOD(nProduct, 10) + ;
             FLOOR(nProduct/10)
ENDFOR

RETURN nResult
```

Next, I created two methods that call on MultiplyAndAddDigits: As their names suggest, DoubleEvenDigitsFrom10 adds the odd digits unchanged and doubles the even digits, while DoubleOddDigitsFrom10 adds the even digits unchanged and doubles the odd digits. Each method sets half the multipliers to 1 and the other half to 2.

Listing 4. DoubleEvenDigitsFrom10 uses MultiplyAndAddDigits to compute the check digit for the algorithm used in New Brunswick and several other provinces and territories.

```
PROCEDURE DoubleEvenUnitsFromTen
* Double even digits, sum,
* then subtract units from 10
LPARAMETERS cID

LOCAL nTotal, nDigits, nDigit, nUnits, ;
      nCheckDigit, lReturn, aMultipliers[1]

nDigits = LEN(TRIM(cID))
DIMENSION aMultipliers[nDigits]
FOR nDigit = 1 TO nDigits-1
   IF MOD(nDigit,2) = 1
      aMultipliers[nDigit] = 1
   ELSE
      aMultipliers[nDigit] = 2
   ENDIF
ENDFOR
aMultipliers[nDigits] = 0

nTotal = This.MultiplyAndAddDigits(cID, ;
   @aMultipliers)

nUnits = MOD(nTotal, 10)
IF nUnits = 0
   nUnits = 10
ENDIF
nCheckDigit = 10-nUnits

lReturn = (nCheckDigit = ;
          VAL(SUBSTR(cID, nDigits, 1)))
RETURN lReturn
```

DoubleOddDigitsFrom10 is the same, except for the assignment of the multipliers. (Reviewing this code now, I see that I could have pulled the last part into a common routine, as well.)

Although the MOD 11 check digit algorithm is the same everywhere (except for the multipliers), it seemed possible to me that in future, there could be another algorithm that multiplies each digit by a specified multiplier and adds all the results, but does something different with the result. So I wrote another method, MultiplyAndAdd (shown in Listing 5). It receives a character string for the health ID, and a comma-separated list of multipliers; it multiplies each digit by the specified multiplier and returns the total of those numbers. (This version sums the multiplication results as is; it doesn't split out the tens and the ones.)

Listing 5. MultiplyAndAdd multiplies each digit by the specified value and totals the results.

```
PROCEDURE MultiplyAndAdd(cID, cMultipliers)
* Multiply digits by the specified factors
* and sum results

LOCAL nDigits, nDigit, nProduct, nTotal, l
      aMultipliers[1]

nDigits = ALINES(aMultipliers, ;
                 cMultipliers, ",")
FOR nDigit = 1 TO nDigits
   aMultipliers[nDigit] = ;
      VAL(aMultipliers[nDigit])
ENDFOR
```

```
nTotal = 0

FOR nDigit = 1 TO nDigits
   nProduct = VAL(SUBSTR(cID, nDigit, 1)) * ;
              aMultipliers[nDigit]
   nTotal = nTotal + nProduct
ENDFOR

RETURN nTotal
```

Then I created Mod11Check, which calls on MultiplyAndAdd to compute the check digit. It's shown in **Listing 6**.

Listing 6. The code to compute a check digit with the MOD 11 method is quite simple, since it calls on MultiplyAndAdd for most of the work.

```
PROCEDURE Mod11Check(cID, cMultipliers)
* Multiply digits by the specified factors,
* add results and return result mod 11

LOCAL nTotal, nResult

nTotal = This.MultiplyAndAdd(cID, ;
                             cMultipliers)

nResult = MOD(nTotal, 11)

RETURN nResult
```

You might wonder why I use an array of multipliers for the first algorithm, but pass a comma-separated list of them for the second. In the MOD 11 case, the list of multipliers is passed from outside this object; that is, whatever calls this code has to put the list together. For the odd/even strategy, the list of multipliers is created internally (in Double*XX*DigitsFrom10) . Passing arrays is a bit of a pain in VFP, so using a string for the external call made sense to me. The string is converted to an array before we use it in MultiplyAndAdd.

There's one other difference between the two algorithms. The Double*XX*DigitsFrom10 methods actually check whether the PHN passed in has a valid check digit and return a logical value. Mod11Check returns the computed check digit.

The reason they work differently is that all the provinces and territories that use the first strategy incorporate the check digit in the PHN. The provinces and territories using the MOD 11 strategy vary as to whether the check digit is part of the PHN, or should be 0.

Using the common code

The next step was to find a way to put all this code to use for the individual provinces. I could have simply created an additional set of methods, one per province that called on the common code. However, that would have meant that each time a province changed its check digit algorithm, the application would have to be updated.

Instead, I stored that "method" code in a table. CheckDigitCode has the structure shown in Listing 7. cProvince contains the two-digit abbreviation for the province. lRecip indicates whether there's a reciprocal coverage agreement with that province. The code is stored in the memo field, mCode. nIDLen indicates the length of the PHN for that province.

Listing 7. The code to compute the check digits is stored in a table, to make it easier to manage changes.

```
CPROVINCE       Character       2
LRECIP          Logical         1
MCODE           Memo            4
NIDLEN          Numeric         2
```

The code in mCode expects two parameters, the PHN to be checked, and a reference to the checker object. It returns a logical value, indicating whether the PHN passes the test.

The code in mCode for New Brunswick is shown in Listing 8. It's about as simple as possible. The PHN is passed unchanged, and the result of a single call to DoubleEvenDigitsFrom10 is returned.

Listing 8. The code to check a New Brunswick PHN is the simplest case.

```
LPARAMETERS cID, oCheck
RETURN oCheck.DoubleEvenUnitsFromTen(cID)
```

Alberta uses the same strategy for computing the check digit, but puts it in the middle rather than at the end of the PHN, so the code in mCode in that record (shown in Listing 9) is a little more complex.

Listing 9. To check an Alberta PHN, you have to pull the check digit out of the middle first.

```
LPARAMETERS cID, oCheck
LOCAL cAdjustedID

cAdjustedID = LEFT(cID, 4) + RIGHT(cID,4) + ;
              SUBSTR(cID, 5, 1)

RETURN ;
  oCheck.DoubleEvenUnitsFromTen(cAdjustedID)
```

Manitoba uses the MOD 11 strategy, and makes the check digit the final digit of its 9-digit ID. Since the MOD 11 algorithm could return 10, only the ones digit of the result is used. The code in Manitoba's record is shown in Listing 10.

Listing 10. Manitoba uses the MOD 11 strategy and includes the check digit as the last character of the PHN.

```
LPARAMETERS cID, oCheck
LOCAL cValues, nCheckDigit, lReturn

cValues = "29,23,19,17,13,7,5,3"

nCheckDigit = MOD(oCheck.Mod11Check(;
   cID, cValues), 10)
```

```
lReturn = (nCheckDigit = VAL(RIGHT(cID,1)))

RETURN lReturn
```

Saskatchewan also uses MOD 11, but rather than incorporating the check digit, it requires it to be 0. So the memo field contains the code in Listing 11.

Listing 11. Saskatchewan requires the MOD 11 check digit to be 0, so its code is fairly simple.

```
LPARAMETERS cID, oCheck
LOCAL cValues, nCheckDigit, lReturn

cValues = "9,8,7,6,5,4,3,2,1"
nCheckDigit = oCheck.Mod11Check(cID, cValues)

lReturn = (nCheckDigit = 0)

RETURN lReturn
```

One of the strengths of this approach is that it accommodates odd requirements well. For example, all British Columbia PHN's begin with 9, while those for Northwest Territories must begin with a letter. It's easy to code such rules in the mCode field, and should they change, easy to make those adjustments without rebuilding the application.

In fact, even the list of entities to which the rules apply can change. It's only a few years since the Northwest Territory was split to add Nunavut. A code-only solution would require a new build in that case. With this approach, just one table has to be updated and distributed.

Combining common code with data

The final step in the process is putting the stored code to work. To do that, I use a method in the same object as the common code. CheckID (Listing 12) receives the PHN and the province abbreviation as parameters. It finds the province data, checks the length of the passed PHN, then executes the appropriate code. For simplicity, the data in the table is pulled into an array (aCheckMethods) in the object's Init method, and the array is used here.

Listing 12. The CheckID method pulls the whole process together. The PHN's length is checked, and if it passes that test, the check digit is tested, according to the rules for that province.

```
PROCEDURE CheckID(cID, cProv)

LOCAL lReturn, nRow

* Find the right method and call it
nRow = ASCAN(This.aCheckMethods, ;
             UPPER(cProv), -1, -1, 1, 8)
IF nRow = 0
   * Return .F. if the province code
   * is no good.
   lReturn = .F.
ELSE
   * First check length
   IF LEN(ALLTRIM(cID)) = ;
       This.aCheckMethods[nRow, 3]
      cCode = This.aCheckMethods(nRow, 2)
      lReturn = EXECSCRIPT(cCode, ;
                           ALLTRIM(cID), ;
                           This)
   ELSE
      lReturn = .F.
   ENDIF
ENDIF

RETURN lReturn
```

The key code in CheckID is the EXECSCRIPT() line. It executes the code from the memo field, passing the two required parameters: the PHN, and a reference to the object.

The downloads include the code and the table, along with a program that demonstrates the use of the code.

Using this technique

I haven't used this strategy in another application yet. But several of the tools that come with VFP use similar approaches.

IntelliSense has a code component that includes a number of useful methods. It's driven by the FoxCode table. You can include executable code in the Data memo for certain record types. That code receives an object reference to the IntelliSense object, and thus can use the built-in methods. To see this in action, check out any of the records where type="S".

The Task Pane Manager has an engine object and a driving table. Code stored in the table can use methods of the engine object. As with IntelliSense and my PHN code, the code in the table receives as a parameter an object reference to the engine, so it has easy access to the code.

Keep this approach in mind any time you have some common code, but a number of ways of using it, especially in cases where those ways may change over time.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of ten books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL . Her latest collaboration is Making Sense of Sedna and SP2, coming out this year. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.