Journal of Information Systems Education, Vol. 19(3)

Classroom Minicases

Five Data Validation Cases

Mark G. Simkin
College of Business Administration
University of Nevada
Reno, Nevada 89557
MarkGSimkin@yahoo.com

ABSTRACT

Data-validation routines enable computer applications to test data to ensure their accuracy, completeness, and conformance to industry or proprietary standards. This paper presents five programming cases that require students to validate five different types of data: (1) simple user data entries, (2) UPC codes, (3) passwords, (4) ISBN numbers, and (5) credit card numbers.

Keywords: Data Validation, Computer Programming, Check Digits, ISBN Numbers, UPC Codes

1. INTRODUCTION

It is difficult to overstate the importance of creating data validation routines as an integral component of most application software. This observation applies to health care systems (Barlow, 2006), engineering systems (Butts, 2007; Griebenow and Caudill, 2005), airline baggage-handling systems (Bailey, 2007), and mortgage banking (2007), but is also a vital part of applications in sports, medicine, and academia.

As used here, the term "data validation" refers to methods for ensuring that the data entered into a computer system are complete, accurate, and reasonable. In many business settings, such validation also requires conformance to industry standards—for example, consistency with the 16-digit standard of consumer credit cards or the 10-digit ISBN numbers used by book publishers. In other cases, proprietary data formats are used, necessitating custom validation code to ensure their accuracy. In all cases, it is difficult to identify a business or government activity in which inaccurate data entry is even remotely acceptable.

Most computer programmers are well aware of the comparatively large amounts of code usually required to validate input data, but such programming burdens are typically unfamiliar to students in entry-level programming classes. A similar comment applies to the *methodologies* required to perform such data validations—for example, the computational methods for validating ISBN codes.

The techniques for data validation tasks vary with the application, but are usually essential to the development process. Typically, they require an understanding of string parsing, looping, and of course selection code for creating the requisite validation tests. Selection code is also required to decide whether or not the data pass such test procedures. Because data validation tasks require a wide array of programming skills, such problems are useful not only as examples of real-world programming tasks but also as integrative exercises that encourage students to draw upon the skills they've acquired from disparate chapters of standard programming texts.

The purpose of this paper is to outline the computational strategies required to validate user values in five, common data-entry applications. In order of presentation, they are cases requiring the validation of (1) simple data entry, (2) UPC codes, (3) passwords, (4) ISBN numbers, and (5) credit card numbers. Each case is independent of the others, and can therefore be assigned individually.

The paper presents these cases in approximate order of programming difficulty, although this is a somewhat subjective judgment. The author has used each of them successfully in at least five different Visual Basic programming classes, although the solutions to these cases can also be developed in such other procedural programming languages as C++ or Java.

2. CASE 1: VALIDATING SIMPLE USER ENTRIES

2.1 Description:

Many data entry screens allow users to enter name, address, or similar personal information. The screen in Figure 1 is an example. This interface allows users who have placed catalog orders to also purchase monogramming, gift wrapping, or even a singing telegram. The application also requires developers to validate the inputs from several types of input controls, including radio buttons, check boxes, and textboxes.

2.2 The Customize Order Screen

The application works as follows. The user enters data in the interface shown in Figure 1. When the form first appears, the radio buttons for the various singing telegram options are

disabled. If the user clicks on the Clear button, the system unchecks all the checkboxes and radio buttons and clears all the textboxes on the form. If the user checks the checkbox for a singing telegram, the system enables the radio buttons. If the user unchecks the checkbox for a singing telegram, the system disables the radio buttons.

When the user clicks on the Compute Total button, the system first validates the entries on this form as follows. (1) If the user has checked the Monogramming checkbox, the system ensures that there are three or less letters to monogram and that the number of items to monogram is a positive number. (2) The maximum number of items to monogram is ten. (3) If the user checks the GiftWrapping checkbox, the system ensures that the number of items to gift wrap is a positive number less than or equal to 20. (4) If the user checks the "Send a Singing Telegram" checkbox, the system ensures that the user also selects one of the associated radio buttons.

Figure 1. An interface that allows users to customize an order.

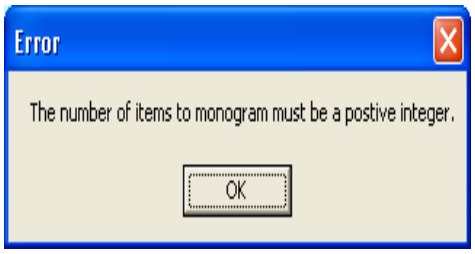

Figure 2. Error message indicating a problem with a data entry in Figure 1.

If the user violates *any* of these requirements, the system should display a customized message (Figure 2) indicating the problem and *not* compute any totals. If the user enters valid data, the system should compute the proper costs for monogramming, gift-wrapping, and singing telegrams, and also the grand total for all these items. In the Total Cost Summary portion of the interface, the various cost components should be right justified and formatted to currency values.

2.3 Case Deliverables

Test your application with the following test cases. For each set of good test data, screen-capture the user interface *after* the system has computed the total charge for each set of user selections. For each set of bad test data, screen capture the error message displayed by your application at run time.

Test Case	Mono-gram Letters	Number of Monogram Items	Number of Gift-wrapped items	Singing Telegram Choice	Comment
1	AAA	2	None	Happy Birthday	Good data
2	BBB	3	5	Congratu-lations	Good data
3	None	None	3	None	Good data
4	CCC	12	2	None	Monogram mistake
5	DDD	None	23	None	Gift wrap mistake
6	EEEE	2	None	None	Monogram mistake
7	FF	2	-3	None	Gift wrap mistake

3. CASE 2: VALIDATING UPC CODES

3.1 Description

The universal product code (UPC) is a 12-digit code that appears on the packaging of most items sold in U.S. retail stores. A bar code typically accompanies the numeric code, enabling a retailer's bar code reader to capture the numeric values electronically. To ensure accurate reading, the UPC includes a check digit that a computer system can test internally to validate the code. The purpose of this assignment is to create a computer application that simulates such a validation procedure.

3.2 How a UPC Check Digit Works

The final digit of the universal product code is a check digit that can be computed from the other digits as follows:

1. Separate the last digit from the rest of the UPC code. This is the check digit.
2. Add the digits up to, but not including, the check digit in the odd-numbered positions (i.e., the numbers in the first, third, fifth, etc., position) together and multiply by three.
3. Add the digits up to, but not including, the check digit in the even-numbered positions (second, fourth, sixth, etc.). Add this value to the result found in step 2.
4. If the last digit of the result is 0, then the check digit is 0. If the last digit of the result is not zero, then subtract the last digit from 10.
5. Compare results. The computed check digit must equal the initial check digit from step 1.

To illustrate, suppose the UPC code is "036000291452." Here are the steps for this example.

1. The last digit is the check digit "2." If the other numbers in the bar code are correct, then the check digit calculation must produce 2.
2. Add the odd number digits 0+6+0+2+1+5 = 14, and multiply by 3 to get 14 × 3 = 42.

3. Add the even number digits to this result = 42 + 3 + 0 + 0 + 9 + 4 = 58.
4. The last digit is 8, so the check digit is not 0. Subtract 8 from 10 to get "2."
5. The computed value of "2" matches the check digit. We therefore conclude that the UPC code is valid.

3.3 Case Deliverables

Create a computer application with an interface similar to Figure 3 that validates the UPC code entered in a textbox. When the user clicks on the "Test UPC" button, the system performs this validation work and displays the result in a label on the form. Document your work with a screen capture of your user interface and provide a hard copy of your code or formulas. Test the following UPC codes: (1) 718103049788, (2) 654249600271, (3) 654249600233, (4) 198761542312, (5) 187653416523, and (6) 048109352491. For each number, indicate whether the UPC number is valid or invalid.

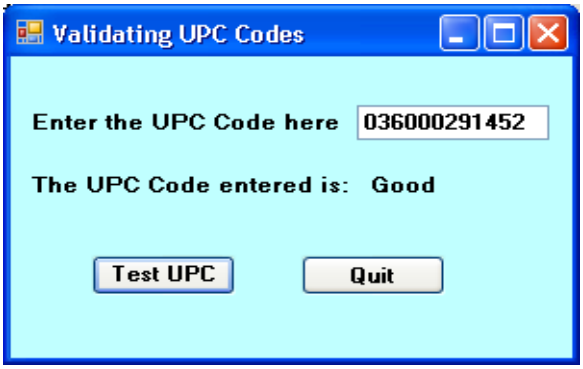

Figure 3. A user interface for validating a UPC value.

4. CASE 3: VALIDATING USER PASSWORDS

4.1 Description

The objective of this assignment is to create a simple computer application that allows a user to create a password and that includes code to validate this password. This assignment requires a good understanding of parsing techniques, For-Next loop constructs, user-defined functions, form-level variables, and programming logic.

4.2 Creating Passwords

In addition to validating existing passwords, many computer systems allow (or require) users to create or change them. Suppose, for example, that a computer system requires a user to create a password of the form AAAANN, where "A" is any alphabetic character (either uppercase or lowercase) and "N" is any numeric digit. An additional requirement is that the password contains both upper and lower case letters. Figure 4 shows a suggested user interface. When the user enters a password in the textboxes and clicks on the "Test Password" button, the system performs the following tests: (1) matching non-blank entries, (2) a proper length of six characters, (3) first four digits are alphabetic, (4) last two digits are numeric, and (5) password has at least one uppercase and one lowercase letter. If the user's entry violates any of these rules, the system provides an error message indicating the problem (Figure 5).

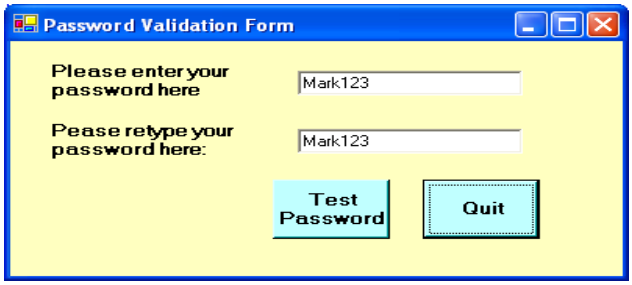

Figure 4. A user interface for creating a new password.

Figure 5. Examples of error messages for bad passwords.

Hints: (1) You should test for a valid password length first, because short passwords cannot be parsed properly. (2) You may find it useful to create a function that tests for a single valid alphabetic character and that you can call repeatedly in your code. (3) One way to test for the presence of both an uppercase and lowercase letter in the password is to use separate Boolean variables (e.g., "FoundUpper" and "FoundLower") to represent a "found" condition for each of them. If you initially set these variables to "false," you can later check whether these variables were reset to "true" with your code.

4.3 Case Deliverables

Screen-capture your user interface at run time and also print a copy of your code. Test your program with the following passwords: (1) MAR123, (2) MARK123, (3) Mark12, (4) Mark123, (5) mark123, (6) Mar123, (7) MArkXX, (8) 123MAR, (9) ZZZz12, and (10) passwords that do not match in first and second text boxes. For full credit, screen-capture the resulting error message for each incorrect password.

5. CASE 4: VALIDATING ISBN NUMBERS

5.1 Description

The International Standard Book Number (ISBN) used on most American books uses a weighted code and modulus-11 validation system. The purpose of this assignment is to create a Visual Basic application that validates such numbers.

5.2 Ten-Digit ISBN Numbers

ISBN numbers prior to 2007 use ten digits. Suppose, for example, a user enters the number 0-7637-2478-5. How can a computer application ensure that this is a valid number? The first step is to strip the "number" of all extraneous dashes, blanks, and similar characters. (Hint: parse the original "number" element by element, and then assemble a new string that contains only numeric values.) For this example, the resulting number is "0763724785."

The second step is to make sure that this new number contains exactly 10 digits. If it does not, there is no need to examine the number further—the ISBN number is invalid.

If the ISBN number contains 10 characters, the third step is to multiply each successive digit in the number by the weights of "10," "9," "8," and so forth, and add the cross-product terms. (Note: an X in an ISBN number stands for "10.") For the current example, this sum is "242" as illustrated in Figure 6.

The final step is to examine the sum and verify that this number is evenly divisible by 11. (Hint: use the Mod function to verify that "sum mod 11 = 0.") If so, the ISBN number is presumed valid. If not, the ISBN is presumed invalid. Your application should provide a message box for each possibility.

5.3 Deliverables

Create a user interface similar to the one in Figure 7. Document your application with a screen capture of the user interface at run time and a hard copy of your code. Also, test your application using (1) the example above (a valid ISBN number), (2) the ISBN number of your textbook (also a valid ISBN number), (3) the ISBN number 0-619-21631-X, and (4) the ISBN number 0-13-030654-X. Make sure that you enter the ISBN numbers with dashes or blanks. Provide a user interface and the message box for each of these four tests.

5.4 Thirteen-Digit ISBN Numbers

In 2007, book publishers migrated to a 13-digit ISBN number with a slightly different check-digit validation system. In this new system, the 13th digit is the check digit.

To validate the entire number, the system computes a new check digit from the first 12 digits and then compares the result to this last digit. To validate a 13-digit ISBN number, proceed as follows: (1) strip the input string of all extraneous blanks or dashes as before. (2) verify that the stripped number contains exactly 13 digits, (3) parse the 13-digit entry into two parts: the first 12 digits and the last, check digit, (4) reading from left to right, multiply the digits in the odd locations by "1" and the digits in the even locations by "3" and compute the sum of these multiples, (5) divide this sum by 10 and take the remainder (i.e., compute Sum Mod 10), (6) if the answer is greater than 0, subtract this remainder from 10, and (7) compare this value to the check digit found in step 3. If the two values match, the ISBN number is presumed to be valid, and invalid otherwise.

To illustrate, suppose the 13-digit ISBN number is 978-0-306-40615-7. The steps above lead to the following computations. (1) The stripped number is 9780306406157. (2) This number contains exactly 13 numeric digits. (3) The ISBN number is actually the first 12 digits, or978030640615, and the check digit is 7. (4) The multiplication is = 9*1 + 7*3 + 8*1 + 0*3 + 3*1 + 0*3 + 6*1 + 4*3 + 0*1 + 6*3 + 1*1 + 5*3 = 93. (5) Computing 93 Mod 10 = 3. (6) Subtracting: 10 – 3 = 7. (7) The computed value matches the check digit value and we conclude that the ISBN number is valid.

ISBN Number	Weight	Product
0	10	0
7	9	63
6	8	48
3	7	21
7	6	42
2	5	10
4	4	16
7	3	21
8	2	16
5	1	5
Sum:		242
Sum Mod 11:		0

Figure 6: The computations for validating a 10-digit ISBN number

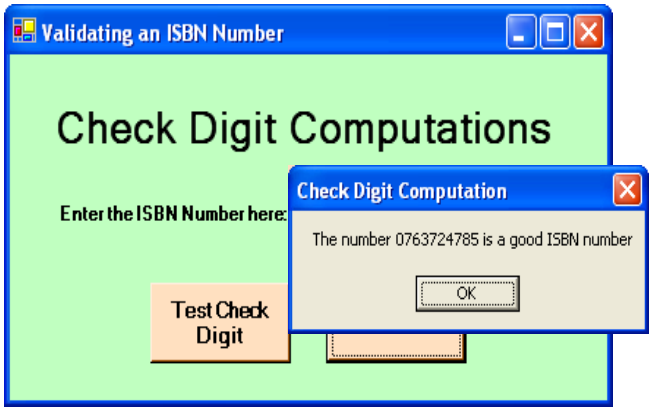

Figure 7. A user interface for entering ISBN numbers.

5.5 New Case Deliverables

Document your application with a screen capture of your user interface at run time and a hard copy of your code. Also, test your application using (1) the example above (a valid ISBN number), (2) the ISBN number 978-1-4188-3643-6 (also a valid ISBN number), (3) the ISBN number 978-0-07-330427-4, and (4) the ISBN number 1254-8732-9755-6. Make sure that you enter the ISBN numbers with dashes or blanks. Provide a user interface and the message box for each of these four tests.

6. CASE 5: VALIDATING CREDIT CARD NUMBERS

6.1 Description

Many web shopping applications require customers to enter credit card information. Although the check digit computations used to validate such information are proprietary, we can simulate them to provide an idea of how

they are performed. This case requires you to validate a 16-digit credit card number. This assignment requires a good understanding of parsing techniques, nested For-Next loops, and programming logic.

6.2 A Payment Interface

Figure 8 is an example of a payments screen. The user must select one of four types of credit cards for payment as shown by the radio buttons in the form. If the user clicks on the OK button, the system first validates the user's information as follows:

1. For the credit card number, each digit in the card number must be numeric, and its check digit must match the month code for the expiration date of the card, as explained below,
2. The expiration date must contain a valid month and the date must be greater than or equal to the system date.
3. One radio button (credit card type) must be selected in the interface.

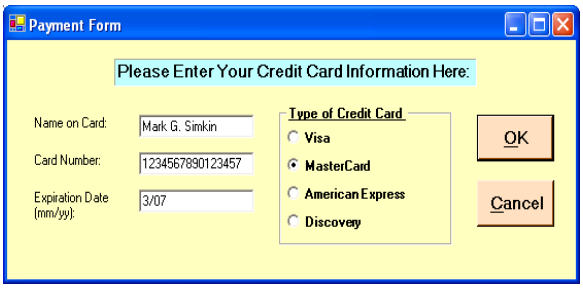

Figure 8. A user interface for entering a credit card number.

If the user fails to select one of the credit card radio buttons or enters data that fail any of the validation tests for this screen, the system should display a Message box similar to the ones in Figures 9 and 10. Note that the system displays the type of credit card in Figure 10.

Of special interest is the way in which the system computes a check digit for the credit card that must match the expiration date's month code. To perform this test, the system uses a *function* (that you write) that tests for card number length and numeric content. This function also repeatedly finds the sum of the numeric digits in the credit card number until it finds a single digit (a "check digit"). It then compares that value to the month code of the card. For example, if the credit card number entered was 1234567890123456, the sum of these digits would be "66," the sum of 6 + 6 is "12," and the sum of these digits is 1 + 2 = 3. Thus, the computed check digit is "3."

A similar computation is performed for the month code. For example, if the expiration month of the credit card were "12," then the month code would be 1 + 2 = 3. If the month code matches the check digit for the credit card, the input data is assumed to be accurate. However, if the check digit and month code do not match, the system should conclude that the credit card number is invalid.

Figure 9. Error message indicating that the user did not select a credit card (radio button) in the Payment Form.

Figure 10. Error message indicating that the check digit computed for the credit card number did not match the month code.

6.3 Case Deliverables

Test your program with the following credit cards and expiration dates. For each credit card, screen capture the user interface similar to the one in Figure 8, and also the error message in Figures 9 and 10 (if the credit card is not valid).

Credit Card Type	Credit Card Number	Expiration Date
Visa	1234567890123456	08/12
Mastercard	7645243289765523	11/11
Mastercard	8634252625242611	07/13
Discovery	1234567890121212	08/10
Discovery	1010101010101001	12/10

7. SUMMARY AND CONCLUSION

Data validation is an integral part of most business information systems, and understanding how and why such applications require data validation is itself useful knowledge. This paper presented five cases that introduce students to common data validation tasks. The author's experience in using such cases in the course of the last five years has been uniformly positive. Students report that they enjoy such tasks, relate to the applications in which they are used, appreciate the integrative nature of the work, and often prefer such assignments to alternate, end-of-chapter programming exercises that may lack such realism.

8. ACKNOWLEDGEMENTS

The author is indebted to two referees for several helpful comments and suggestions on ways to improve an earlier draft of this manuscript.

8. REFERENCES

Bailey, Jeff (2006), "In Airline Baggage Roulette, Travelers' Odds are Getting Worse" *The New York Times* Vol. CLVII, No. 54135, November 21, pp. A1 and A20.

Barlow, Rick Dana (2006), "Deep-Sixing Dirty Data" *Healthcare Purchasing News* Vol. 30, Issue 5, May, pp. 44-47.

Butts, Glen C. (2007) "Excel for Cost Engineers" *AACE International Transactions*,, p 7.

De la Villa Jaén, Antonio, Romero, Pedro Cruz, and Expósito, Antonio Gómez. (2005) "Substation Data Validation by a Local Three-Phase Generalized State Estimator" *IEEE Transactions on Power Systems*, Vol. 20, Issue 1, February, pp. 264-271.

Griebenow, Ron, and Caudill, Marcus (1999), "Precision Data Validation Boosts Process Optimization Benefits" *Power Engineering* Vol. 103, Issue 11, November, pp. 104-108.

AUTHOR BIOGRAPHY

Mark G. Simkin is a professor of Information Systems at the University of Nevada, Reno. He earned his MBA and Ph.D. degrees from the University of California, Berkeley. His research in end-user computing, computer education, and computer crime appears in over 100 academic journal articles, including *Decision Sciences, The Decision Sciences Journal of Innovative Education, The Journal of Accountancy, Communications of the ACM, and Communications of the Association for Information Systems.* Professor Simkin is also the author of 15 books, the most recent of which is *Core Concepts of Accounting Information Systems* (New York: John Wiley and Sons, 2008) with coauthors Nancy A. Bagrannof and Carolyn Norman Strand.

Information Systems & Computing
Academic Professionals

STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN 1055-3096