New Approaches to Creating and Testing Internationalized Software

Harry J. Robinson

Sankar L. Chakrabarti

Creating high-quality software that runs in any language is a big challenge. By changing our development process to stress early defect detection and by using the World Wide Web as a collaboration tool, we have dramatically improved the quality of our internationalized software.

Most software companies want to sell their software in every country in the world. Since users prefer to use software in their native language, it makes good marketing sense to develop software that can run in those languages. After the initial investment has been made to write and test software in English, a software company can make significant profits by reselling the same software to other countries if the cost of conversion into other languages can be kept low.

Traditionally, software testing occurs at the end of the development cycle. This kind of testing works against creating high-quality software. When bugs are found late in the cycle, there is little time to fix them. This is especially true in internationalized software development where the developers, testers, and translators are spread all over the world. Our new approach allows a team

Harry J. Robinson

A software design engineer at the HP Corvallis Imaging Operation, Harry Robinson was a test engineer for the HP Common Desktop Environment internationalization project. He recently left HP to become a lead test engineer at Microsoft Corporation. He has a BA degree in religion (1980) from Dartmouth College and BSEE (1985) and MSEE (1988) degrees from Cooper Union. He is interested in all aspects of software testing. Born in Staten Island, New York, he is married and has three children.

Sankar L. Chakrabarti

Sankar Chakrabarti is a member of the technical staff at the HP InkJet Business Unit. Currently, he is responsible for developing software for a print quality testing tool. Sankar received a doctorate in chemistry in 1974 from the Tata Institute of Fundamental Research in Bombay, India. He joined HP in 1981 after receiving an MS degree in computer science from Oregon State University. Born in Azimganj, West Bengal, India, Sankar is married, has two children, and enjoys traveling and hiking.

to test the software during the entire process and to release foreign language versions simultaneously with the English version.

Developing Internationalized Software

The I18N Approach

One common method of creating internationalized software at a reasonable cost is called I18N.* The essence of the I18N approach is to separate the executable code from any character strings that the user will see. User messages are placed into files called message catalogs. Two numbers, the set number and the message number, index each string in the message catalog. The executable code uses these numbers to retrieve strings.

For example, every C language programmer knows the classic hello, world program:

```
hello.c:
      main()
      {
           printf(”hello, world\n”);
      }
```

In the I18N methodology,[1] this program would be written as follows:

```
hello.c:
      main()
      {
        my_cat=catopen(”hello.cat”, NL_LOCALE);
        printf(catgets(my_cat, 1, 5,
        ”hello, world\n”) );
        catclose(my_cat);
      }
```

The program accesses the string “hello, world\n” by retrieving set 1, message 5 of the “hello.cat” message catalog file.

```
hello.cat:
        $set 1
        5 hello, world\n
```

The string “hello, world\n” that appears in the printf statement is a default string that is used if no message catalog file can be found.

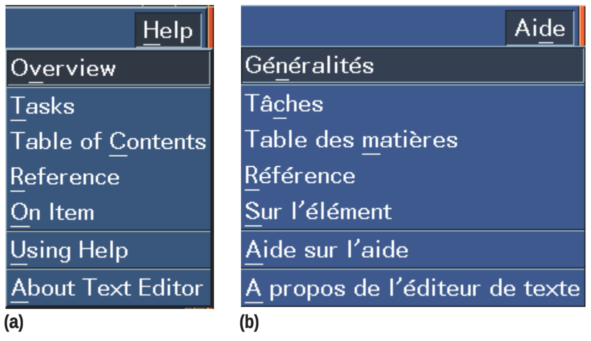
Separating executable code from user-visible strings is very useful when working with translations. If we want to run our hello, world program in French, a translator merely changes the string in the message catalog to:

```
$set 1
5 bonjour, le monde\n
```

* I18N = I[nternationalizatio]N. 18 is the number of letters between the I and N.

Figure 1

A typical application Help menu in (a) English and (b) French.



Users running the hello.c program with the translated message catalog will see bonjour, le monde as their output. No changes are made to the executable code to support the French version. Only the user interface strings need to be translated. The executable code remains unchanged.

Figure 1 shows an example of what a typical application Help menu looks like in both English and French. The same executable code was used to generate each menu, and only the message catalog was changed.

Process Flow

The idea behind creating internationalized software is conceptually simple, especially when the number of languages is small and the application is as simple as hello.c. On the other hand, producing real-world applications in a dozen languages can pose several challenges to a development team.

The shaded area in the diagram in **Figure 2** shows the traditional process flow for developing an internationalized application.

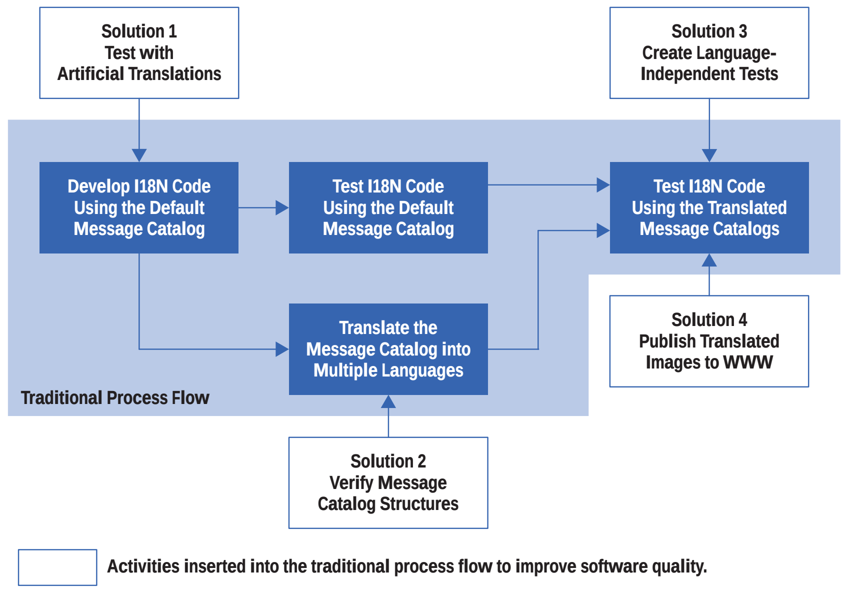
1. The programmers write an application with the appropriate I18N calls for fetching strings from the message catalog. They also produce the original message catalog in English.

2. The message catalog is sent to translators (called *localizers*) who translate each string into a target language, such as French.

3. The application (with the original message catalog) is delivered to the test team, who verify that everything works correctly.

Figure 2

Process flow for developing an internationalized application.



4. The localizers provide the translated message catalogs to the test team. The testers must now verify that the application works in the intended languages.

The Team

Our development team designed and implemented the graphical user interface for Hewlett-Packard's UNIX® workstations. This interface is made up of several applications and runs in a dozen different languages: English, French, Spanish, Italian, German, Swedish, Korean, two forms of Japanese, and three forms of Chinese. The sheer scale of our work causes problems in creating internationalized software because of the wide range of skills and resources needed and the distances between involved parties.

The Programmers. Our entire programming team is located in Corvallis, Oregon. They are software specialists and are not expected to know multiple languages. After they have written their internationalized code, they must wait for the message catalogs to be translated before they can verify that their I18N features are correctly implemented.

The Localizers. Our localizers live in widely separated areas of the world. Most of them are contractors who have never met the rest of our development team. The localizers are not expected to have programming or testing expertise. In fact, they may not even have UNIX workstations on which to run the applications. Often, because the applications they are translating are still in development, they are unlikely to be familiar with the application when they are creating their translations.

The Testers. The test team is located in Corvallis. They are test specialists and are not expected to speak multiple languages. Although I18N methodology is a boon for programmers, it can be a nightmare for the testers. In regular software testing, there is rarely enough time to test an application thoroughly. In the I18N arena, the test team must provide assurance that the software operates correctly in the dozen languages in which it will run. Furthermore, it is very difficult to verify correct operation in a language that one does not speak because mistakes that would be obvious to a native speaker can easily pass unnoticed by the test team.

Challenges and Solutions

We have developed several strategies to deal with many of the challenges we have encountered while creating I18N-enabled code.

Challenge 1: Developers cannot test the I18N-enabled code easily. There are several common mistakes people make when writing I18N-enabled code. One mistake is to neglect to leave enough room in message buffers for the translated message string. Some languages, such as German, require more space than English for the same message. The length of the translated message string cannot be known until run time, though a good rule of thumb is to allow for 60% text growth during translation. If a message string still exceeds the buffer length provided, the program should usually truncate the string.

A second common mistake is when developers neglect to accommodate languages, such as Japanese, that require two bytes to store a single character. Most Western languages require only one byte of storage per character, but several Far Eastern languages have large character sets and need more than one byte per character. If the code does not handle double-byte characters, the results could range from corrupted characters to a crash of the application.

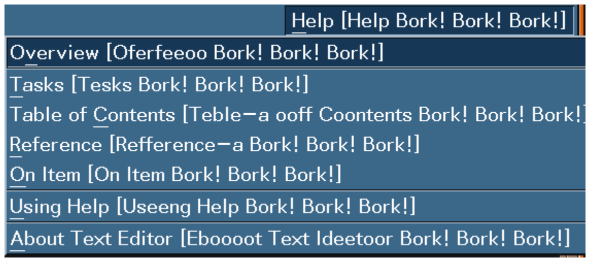
It would be very handy to provide a way to test I18N-enabled code early in the development process, perhaps even as soon as the code is written. The chief difficulty in early testing of I18N-enabled code is that an actual translation may not be available. Message catalog translation is time-consuming. The development team may have to wait several weeks before getting a translated message catalog back from a localizer, losing precious testing and debugging time.

Solution 1: Test with artificial translations.[2] Our solution is to construct artificial message strings that mimic the kinds of problems we see in real translated message strings. This instantaneous creation of a translated message string against which to test our software provides us with quick feedback about how our application will perform with translated components.

For instance, to simulate languages with long text strings, we created a message catalog in a language we call the *Swedish chef*. Using a freeware Internet utility called the Encheferizer,[3] we appended long nonsense strings onto each English string. The results can be seen in **Figure 3**.

Figure 3

Nonsense strings appended to English words to create the Swedish chef language.


Help [Help Bork! Bork! Bork!]
Overview [Oferfeeoo Bork! Bork! Bork!]
Tasks [Tesks Bork! Bork! Bork!]
Table of Contents [Teble—a ooff Coontents Bork! Bork! Bork!]
Reference [Refference—a Bork! Bork! Bork!]
On Item [On Item Bork! Bork! Bork!]
Using Help [Useeng Help Bork! Bork! Bork!]
About Text Editor [Eboooot Text Ideetoor Bork! Bork! Bork!]

Each Encheferized string is at least double the size of the corresponding English string.

Likewise, to simulate double-byte characters such as those used in Japanese, we used a small program that maps ASCII characters into a double-byte format as shown in **Figure 4**. This translation was easy to use and allowed us to detect whether the code handled double-byte strings properly.

Despite these tools, many developers still felt reluctant to test in a language that they did not know. To overcome this reluctance, we demonstrated that it is possible to test even in the worst case imaginable: we created a Klingon target language.[4] Words, chosen at random from a Klingon version of Shakespeare's Hamlet, were inserted into an application's message catalog as shown in **Figure 5**. Even though this version of the application might not make

Figure 4

Text strings mapped into a double-byte format to simulate double-byte characters.

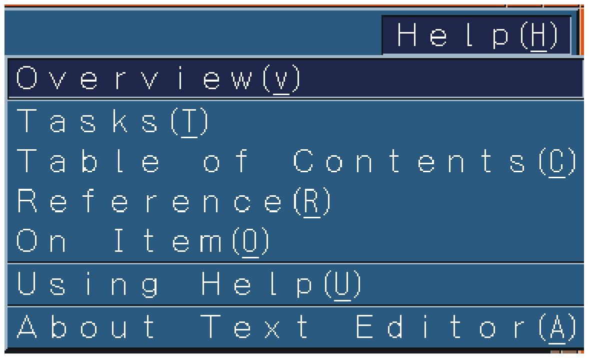
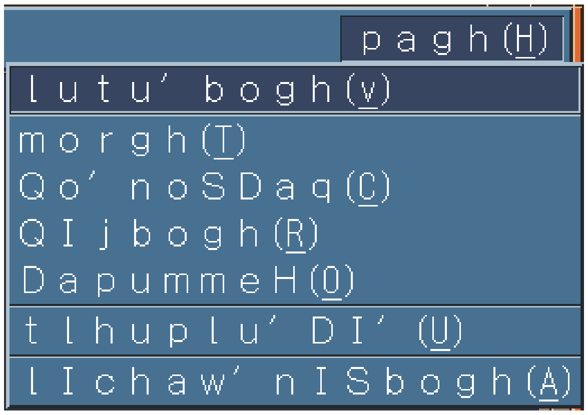


Figure 5

An excerpt from a Klingon version of Hamlet.



sense to anyone, it can still be tested. This test case provided one more level of confidence to our testing.

In all three cases, we used small scripts to create message catalogs from the default English catalog so that the "pseudo-translated" messages were available as soon as the code was ready to be tested (see **Figure 2**). Because the translations we chose were simple to use, somewhat whimsical, and not at all intimidating, programmers found it easy to perform I18N testing.

Challenge 2: Translators can introduce structural defects into the message catalogs. Each message catalog has a structure. Sometimes the structure is as simple as the set and message numbers, and at other times it can be complex. If the structure of the message catalog is changed during translation, the software will not behave correctly. For instance, suppose the original English message catalog contains:

```
$set 1
5 hello, world\n
```

and the French translation contains:

```
$set 1
55 bonjour, le monde\n
```

The message number has been inadvertently changed from 5 to 55. The application will not work correctly because it cannot find the translated string. These types of mistakes are very hard to catch because of the complexity of some message catalog structures.

Solution 2: Verify message catalog structures. We created small utility programs to verify catalog structures automatically.[5] In our organization, we call these utilities *poka-yokes* after the Japanese quality assurance method that inspired their use.[6] These programs often take less than an hour to write. One typical poka-yoke program might check that each message in the translated catalog corresponds to a message in the original English catalog. Such a check will detect the error in the above example in which the message number is changed from 5 to 55.

These utilities are surprisingly effective. When we ran the utilities on eight applications that had been translated into twelve languages, we found 833 defects in the message catalogs. An even bigger benefit was that the utilities did not need an application. The message catalogs could be checked as soon as they were received from the localizers, and any defects could be fixed immediately.

Poka-yoke scripts are also very useful for finding mistakes overlooked by visual verification. For example, in the French menu in **Figure 1b** there is an error in the "shortcut keys" or mnemonics designated for the last two items in the menu. Shortcut keys must be unique within a menu. The letter A is designated as the shortcut key for both of the last two items, violating the uniqueness rule. People often miss these type of errors, but a poka-yoke script can catch them automatically.

Challenge 3: Testers cannot manually test each application in a dozen languages. After the localizers return the translated message catalogs, it is time to system test the applications in each language. Testing software applications thoroughly is always a challenge. Trying to test the same application in twelve languages can be a catastrophe.

For instance, if we wanted to test the output of the hello.c program listed above, we would typically create a test case that looked something like this:

Step 1: Set the target language to English
Step 2: Run the hello.c program
Step 3: Verify that the output is hello, world

If we wanted to test the French version, we would need to change the test case:

Step 1: Set the target language to French
Step 2: Run the hello.c program
Step 3: Verify that the output is bonjour, le monde

What do we do when we need to test the same program in the other ten languages? Will we need to write a dozen versions of each test? Also, how will we run the tests? Hiring additional people to test the foreign language versions is costly, and traditional record-and-playback test methods do not port well across languages. What is needed is an automated test method in which the cost of testing does not become prohibitive as the number of supported languages grows.

Solution 3: Create language-independent tests.[7] We determined that I18N-enabled tests were needed to test I18N-enabled applications. So, instead of creating a multitude of static tests that would each check for a different string such as hello, world or bonjour, le monde, we created tests that could use the application's own message catalogs to verify output. When an internationalized test needs to verify a program's output, it retrieves the expected message from the message catalog just as the application does. The new I18N form of the automated test for the hello world program would look as follows:

Step 1: Set the target to the desired language
Step 2: Run the hello.c program
Step 3: Retrieve the string stored in set 1, message of hello.cat
Step 4: Verify that the program's output matches the retrieved string
Step 5: Choose the next language to test
Step 6: Repeat until all languages have been tested

This approach can be used to verify that the program is working correctly by iterating through all available languages. Internationalized tests are particularly useful in automatically verifying basic functionality.

Challenge 4: Testers cannot always detect errors in unfamiliar languages. One significant problem in testing internationalized software is that testers unfamiliar with a language usually cannot see errors that might be obvious to a native speaker of the language. For example, **Figure 6** shows part of an application window in Japanese. The string in front of the (V) is transliterated Hyoji and means View. **Figure 7** shows that same window except that the string in front of the (V) has been corrupted by a typo in the script that was used to compile and build the application. Thus, the string in **Figure 7** is meaningless.

This corruption error is immediately apparent to someone fluent in Japanese. Yet, how many non-Japanese speakers would recognize the error, especially since the corrupted form of the string is still in Japanese?

As mentioned above, we don't require that our software testers know multiple languages, and it is certainly unlikely that we could find a test team fluent in the dozen languages our software supports. On the other hand, the localizers who do speak those languages are unlikely to see the error because they do not have the current revisions of the code or the UNIX workstations on which to run the software. Even if they did have the code and a workstation, it would be difficult to know whether they had adequately exercised the application to display all the relevant screens. Previously, one common solution was to fly the translators to our Corvallis site and have them spend several days watching testers execute tests for their particular language. This method was inconvenient and costly.

How could we provide a medium of collaboration between the testers who run the software and the language experts who could judge whether the output was correct?

Solution 4: Publish translated images to the World Wide Web.[8] The World Wide Web can be used to distribute test outputs to those who can help judge if the output looks correct. We call this approach a "traveler tour" because it reminds us of a traveler who visits other countries and then returns home and displays the pictures taken on the trip.

Figure 6

Part of an application window in Japanese containing the correct string preceding the (V).

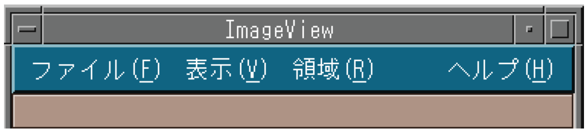


Figure 7

The same character string shown in Figure 6 but with an error in the string preceding the (V).

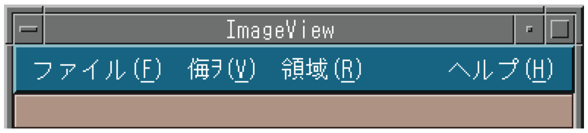
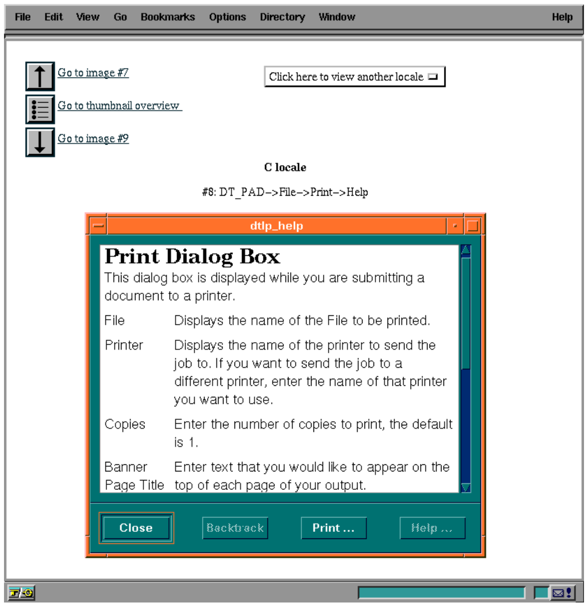


Figure 8

HTML version of an English help page used for comparing screen images during automated testing.



C locale

#8: DT_PAD–>File–>Print–>Help

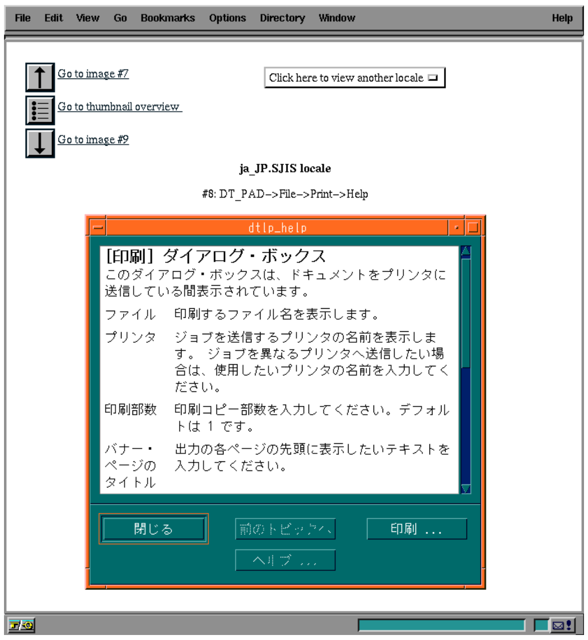
Print Dialog Box

This dialog box is displayed while you are submitting a document to a printer.

File	Displays the name of the File to be printed.
Printer	Displays the name of the printer to send the job to. If you want to send the job to a different printer, enter the name of that printer you want to use.
Copies	Enter the number of copies to print, the default is 1.
Banner Page Title	Enter text that you would like to appear on the top of each page of your output.

Figure 9

HTML version of a Japanese help page used for comparing screen images during automated testing.



ja_JP.SJIS locale

#8: DT_PAD–>File–>Print–>Help

[印刷] ダイアログ・ボックス

このダイアログ・ボックスは、ドキュメントをプリンタに送信している間表示されています。

ファイル	印刷するファイル名を表示します。
プリンタ	ジョブを送信するプリンタの名前を表示します。 ジョブを異なるプリンタへ送信したい場合は、使用したいプリンタの名前を入力してください。
印刷部数	印刷コピー部数を入力してください。デフォルトは 1 です。
バナー・ページのタイトル	出力の各ページの先頭に表示したいテキストを入力してください。

Using the I18N test techniques mentioned in solution 3, we set the target language and drive the application to display various user screens. As each screen is displayed, we capture the image to a file. After all desired images have been captured in all languages, we run a script that creates HTML pages automatically from the images.

Figures 8 and **9** show the HTML version of English and Japanese help pages. The format allows Web page visitors to move easily between the different language versions of an image, as well as move through the sequence of images in a single language.

After the application's images have been captured and moved to the World Wide Web, we invite interested parties, such as the translators and our partner labs, to access the Web pages and give us feedback about whether the translated applications are correct. This arrangement is very useful to all the teams. The testers can ensure that all the relevant screens are displayed, and the language experts can view the output without the expense and administrative overhead of maintaining workstation test systems at their sites.

The traveler tour approach also helps localizers translate the original default English messages into the desired foreign languages. Previously, the localizers were only provided with the message catalogs, which were difficult to translate because the user messages lacked the context of the application. Now, we send the localizers a traveler tour of the application in English along with the message catalogs.

Results

By changing our development and testing processes and creating tools to support early detection of defects, we have revolutionized the way internationalized software is developed at Hewlett-Packard. With these changes:

- Developers can test the I18N features of their code immediately.
- Poka-yoke utilities can catch message catalog defects at the translation stage.
- Language-independent test suites permit automatic testing in all languages.

- Traveler tours allow testers and localizers to work together to detect subtle translation bugs.

This new approach to developing I18N software has reduced the resources needed for testing by a factor of five. It has eliminated the travel costs previously incurred in bringing translators to the development site to assist in testing. The approach has dramatically increased the quality of our internationalized software while at the same time decreasing the time devoted to development, translation, and testing.

Finally, our translators and our partner labs are so pleased with the outcome of this work that they have asked that these changes be incorporated into our regular delivery mechanisms.

Conclusion

Internationalized software has great advantages for the marketplace and is a worthwhile and growing trend, but high quality levels can only be achieved if internationalization is integrated with the rest of the software development process. Current development models do not encourage easy integration of coding, localizing, and testing. We have designed tools to promote early detection of defects and collaboration among the different groups involved in software creation.

Acknowledgments

The authors would like to thank Ken Bronstein, Arne Thormodsen, Dan Williams, and Barbara Wingert-Burbach for their help in developing and promoting the techniques used in this testing program.

References

1. T. McFarland, *X Windows on the World*, Prentice-Hall, 1996.

2. H. Robinson and A. Thormodsen, "Parlez-Vous Klingon? Testing Internationalized Software with Artificial Locales," *Proceedings of the 1997 Pacific Northwest Software Quality Conference*, pp. 185-195.

3. J. Hagerman, *Ze Sveedish Chef*, http://www.almac.co.uk/chef/chef.html

4. M. Okrand, *The Klingon Dictionary: English/Klingon Klingon/English*, Pocket Books, 1992.

5. H. Robinson, "Using Poka-Yoke Techniques for Early Defect Detection," *Proceeding of the Sixth Annual Conference on Software Testing, Analysis and Review*, 1997, pp. 119-142.

6. S. Shingo, *Zero Quality Control: Source Inspection and the Poka-yoke System*, Productivity Press, 1986.

7. S. Chakrabarti and H. Robinson, "Testing CDE in Sixty Languages: One Test Is All It Takes," *Proceedings of the Fourteenth International Conference on Testing Computer Software*, 1997, pp. 419-428.

8. S. Chakrabarti and H. Robinson, "Catching Bugs in the Web: Using the World Wide Web to Detect Software Localization Defects," *Proceedings of the Tenth International Software Quality Week*, 1997, p. 7A2.