Paper presented at the Sixth International Conference on Software Testing Analysis and Review (STAR'97)

Using Poka-Yoke Techniques for Early Defect Detection

Harry Robinson

Abstract

Poka-yoke is a quality assurance technique developed by Japanese manufacturing engineer Shigeo Shingo. The aim of poka-yoke is to eliminate defects in a product by preventing or correcting mistakes as early as possible. Poka-yoke has been used most frequently in manufacturing environments.

Hewlett Packard currently develops its Common Desktop Environment software to run in twelve locales or languages. Traditional testing of this localized software is technically difficult and time-consuming. By introducing poka-yoke (mistake-proofing) into our software process, we have been able to prevent literally hundreds of software localization defects from reaching our customers.

This paper describes the poka-yoke quality approach in general, as well as our particular use of the technique in our localization efforts. Poka-yoke is providing a simple, robust and painless way for us to detect defects early in our localization efforts.

Poka-yoke

History

Poka-yoke (pronounced "POH-kah YOH-kay") [1] was invented by Shigeo Shingo in the 1960s. The term "poka-yoke" comes from the Japanese words "poka" (inadvertent mistake) and "yoke" (prevent) [2]. The essential idea of poka-yoke is to design your process so that mistakes are impossible or at least easily detected and corrected.

Shigeo Shingo was a leading proponent of statistical process control in Japanese manufacturing in the 1950s, but became frustrated with the statistical approach as he realized that it would never reduce product defects to zero. Statistical sampling implies that some products to go untested, with the result that some rate of defects would always reach the customer.

While visiting the Yamada Electric plant in 1961, Shingo was told of a problem that the factory had with one of its products. Part of the product was a small switch with two push-buttons supported by two springs. Occasionally, the worker assembling the switch would forget to insert a spring under each push-button. Sometimes the error would not be discovered until the unit reached a customer, and the factory would have to dispatch an engineer to the customer site to disassemble the switch, insert the missing spring, and re-assemble the switch. This problem of the missing spring was both costly and embarrassing. Management at the factory would warn the employees to pay more attention to their work, but despite everyone's best intentions, the missing spring problem would eventually re-appear.

Shingo suggested a solution that became the first poka-yoke device [3]:

- In the old method, a worker began by taking two springs out of a large parts box and then assembled a switch.
- In the new approach, a small dish is placed in front of the parts box and the worker's first task is to take two springs out of the box and place them on the dish. Then the worker assembles the switch. If any spring remains on the dish, then the worker knows that he or she has forgotten to insert it.

The new procedure completely eliminated the problem of the missing springs.

Shingo went on to develop this mistake-proofing concept for the next three decades. One crucial distinction he made was between a mistake and a defect. Mistakes are inevitable; people are human and cannot be expected to concentrate all the time on the work in front of them or to understand completely the instructions they are given. Defects result from allowing a mistake to reach the customer, and defects are entirely avoidable. The goal of poka-yoke is to engineer the process so that mistakes can be prevented or immediately detected and corrected. Poka-yoke devices proliferated in Japanese plants over the next three decades, causing one observer to note [4]:

It is not one device, but the application of hundreds and thousands of these very simple "fail-safing" mechanisms that day after day has brought the quality miracle to Japan. Each one is relatively simple -- something you easily could do on your own. It is the totality, the hundreds of devices, that is almost frightening to behold.

Categories of poka-yoke devices

Poka-yoke devices fall into two major categories: *prevention* and *detection*.

A *prevention* device engineers the process so that it is impossible to make a mistake at all. A classic example of a prevention device is the design of a 3.5 inch computer diskette. The diskette is carefully engineered to be slightly asymmetrical so that it will not fit into the disk drive in any orientation other than

the correct one. Prevention devices remove the need to correct a mistake, since the user cannot make the mistake in the first place.

A *detection* device signals the user when a mistake has been made, so that the user can quickly correct the problem. The small dish used at the Yamada Electric plant was a detection device; it alerted the worker when a spring had been forgotten. Detection devices typically warn the user of a problem, but they do not enforce the correction.

We are surrounded every day by both detection and prevention poka-yoke devices, though we may not usually think of them as such. My microwave will not work if the door is open (a prevention device). My car beeps if I leave the key in the ignition (a detection device). At few years ago, some cars were designed not to start until the passengers had buckled their seat belts (a prevention device); but this mechanism was too intrusive and was replaced by a warning beep (a detection device).

Characteristics of good poka-yoke devices

Good poka-yoke devices, regardless of their implementation, share many common characteristics [5]:

- they are simple and cheap. If they are too complicated or expensive, their use will not be cost-effective.
- they are part of the process, implementing what Shingo calls "100%" inspection.
- they are placed close to where the mistakes occur, providing quick feedback to the workers so that the mistakes can be corrected.

Judged by these criteria, the "small dish" solution to the missing-spring problem is an excellent poka-yoke device:

- It was simple.
- It was cheap, involving only the cost of a small dish.
- It provided immediate feedback about the quality of the work; corrections could be made on the spot.

Further reflections on the small dish solution

The small dish solution used at Yamada Electric is typical of many poka-yoke devices:

- It did not merely examine switches at the end of the operation; it changed the procedure for assembling switches. The additional step of putting the springs into the dish slowed down the individual operation, but the increased reliability of the assembly eliminated the need for rework and therefore sped up the overall process.
- It was designed to stop a particular mistake -- a worker forgetting to insert a spring. It did not stop all possible mistakes. It would not detect, for instance, a situation where the worker, after removing the springs from the dish, accidentally dropped one on the ground without noticing.
- Being a detection device, the small-dish solution was not completely error-proof. It could only warn of a problem, relying on the worker to correct the situation. Unlike the 3.5 inch diskette, this solution did not make it impossible to assemble a switch incorrectly. A worker wishing to ignore the warning could do so.
- The solution dealt with aspects of the assembly that were necessary, though not sufficient, for correct operation of the product. This poka-yoke ensured only that each push-button had a spring under it; it did not attempt to detect whether the springs were the right height or made of the proper materials.
- Finally, the quality check done was independent of the actual, eventual use of the switch. The poka-yoke device was oblivious to the overall goal of a properly assembled switch. Instead, one could argue that the small-dish solution was actually implementing a crude form of syntax checking enforcing the one-to-one correspondence between push-buttons and springs. I will return to this notion of syntax-checking later in this paper.

Poka-yoke and Software Quality

Being mainly a manufacturing technique, poka-yoke has only rarely been mentioned in connection with software development, but the philosophy behind poka-yoke has never been far from the heart of software quality. Gordon Schulmeyer [6] and James Tierney [7] refer to poka-yoke explicitly, but many software quality authors have championed detection and prevention methods in software.

In 1990, Boris Beizer wrote in Software Testing Techniques [8]:

We are human and there will be bugs. To the extent that quality assurance fails at its primary purpose -- bug prevention -- it must achieve a secondary goal of bug detection.

In 1993, Steve Maguire echoed a similar sentiment in Writing Solid Code [9]:

All of the techniques and guidelines presented in this book are the result of programmers asking themselves two questions ...

- How could I have automatically detected this bug?
- How could I have prevented this bug?

Prevention devices in software

From a poka-yoke perspective, the development of computer languages could be viewed as a prevention device, since one objective of these languages is to prevent us from creating code that can be error-prone. High level languages prevent self-modifying code. Structured programming rescues us from spaghetti code. Object-oriented programming keeps us from stepping on each other's data.

Detection devices in software

Software testing is a form of detection device, but traditional system testing occurs too late in the process to allow quick, corrective feedback on mistakes. Unit testing and "smoke testing" [7] come closer to the notion of poka-yoke, in that they are located close to the source of the potential mistakes and the quick feedback they provide can keep mistakes from moving further along in the process.

The tools in software that most closely resemble poka-yoke devices are the programs such as lint, printfck, cchk, clash [10] that examine the syntax of programs and alert the programmer to a possible mistake in need of correction. Static analysis utilities are simple and cheap to run; they aim to eliminate certain classes of common mistakes; and they concentrate on the syntax of the program rather than the program's function.

Some of Hewlett Packard's recent work in localizing software applications has illuminated areas that yield more readily to a poka-yoke approach than to traditional testing. The sections that follow describe our application of poka-yoke principles to solve a problem that defied a traditional software testing approach.

Poka-yoke and Localization

Some background on message catalogs and localization

To create POSIX-compliant software that runs in multiple locales, developers store locale-specific strings in files called message catalogs. [11] Rather than hard-code a text string into the application, a developer stores the text string in the message catalog and references it by its message set and message number. A message set and message number uniquely identify any message string in the catalog.

Localization is the process of creating a message catalog for a particular language. Hewlett-Packard currently localizes its Common Desktop Environment software for 11 locales: French, German, Italian, Korean, Spanish, Swedish, plus 2 Japanese locales and 3 Chinese locales.

Localization is typically done after the development of the software has stabilized, and it is typically done by people external to the core development organization. These people, called localizers, receive the application's message catalog from the development organization. They then translate each message string into its equivalent expression in the target language.

Localizing a software application is a difficult job. The localizers may be unfamiliar with the application. They may be located halfway around the world from the development organization. They may not even be familiar with programming. Usually, therefore, the localizer performs translations based almost exclusively on the contents of the message catalogs and the information provided in the localization documentation.

Testing localized software

Testing localized software poses a unique set of challenges. The localizers know what the translated messages say, but since they may never have seen the application run, they cannot know if their translation is correct. The development team knows what the translated messages are supposed to say, but since they are not familiar with the target languages, they cannot know if that is what the translated message actually says. Given the constraints of time and distance, it is difficult for localizers and developers to work together, especially when localization is being done in 11 languages.

The usual testing approaches do not offer much relief. Running the tests manually can become tedious when there are 11 foreign locales to test. Testers become fatigued running the same test in multiple locales.

The traditional way to test the message catalogs is as follows:

- the test team receives the translated message catalog from the localizer
- the test team installs the new message catalog and executes the test plan in the target locales
- obvious mistakes are referred back to the localizer and incorporated into a later release of the catalog.

This approach has several drawbacks. It is difficult to execute a test automatically in multiple locales. For one thing, image comparisons are not portable across locales, since by the definition of localization something should change on the screen when moving to a new locale. The alternative -- recording golden images in a dozen locales -- would be a maintenance nightmare.

Yet some sort of testing of localized software is necessary because there are many opportunities for a localizer to make mistakes when creating a message catalog. A localizer could

- specify an application menu incorrectly
- inadvertently delete a message string
- specify an invalid data format
- specify an invalid conversion format
- neglect to translate a message
- translate a phrase incorrectly due to lack of context

All of these mistakes have different causes and different effects on the software that is delivered to the customer. It is, however, possible to construct poka-yokes to counteract each of these mistakes. As an example, we will go into some depth about the poka-yoke we created to mistake-proof the localized application menus.

Of mice and menus

Many software users navigate through menus using only their mouse, clicking on the selections they want. But users can invoke menu actions without a mouse, by using menu mnemonics. Mnemonics are single characters (usually underlined) in a menu label. If the mnemonic is typed at the keyboard while the menu is displayed, the associated action is invoked, just as if the user had selected the action with a mouse. For instance, in the English locale menu shown in Figure 1, the selection that will close the application window has the label "Close" and the mnemonic "C".

Figure 1: Text Editor File menu in English and French locales

Application menus are prime candidates for translation into various languages. How else can users unfamiliar with English know what options are being offered? And since we are translating each menu label, we must also translate the mnemonic associated with each label. In the French locale menu shown in Figure 1, for instance, the selection that closes the window has the label "Fermer", and the associated mnemonic "F". In their respective message catalogs, the English and French menus appear as follows:

$set 11	$set 11
17 N	17 N
18 New	18 Nouveau
19 O	19 O
20 Open ...	20 Ouvrir ...
21 I	21 I
22 Include ...	22 Inclure ...
23 S	23 S
24 Save	24 Sauvegarder
25 A	25 a
26 Save As ...	26 Sauvegarder sous ...
27 P	27 p
28 Print ...	28 Imprimer ...
29 C	29 F
30 Close	30 Fermer

Table 1: English and French locale menus in message catalogs

Using Poka-Yoke to Detect Menu Defects

We first decided to break the menu testing problem down into parts that we could solve.

Our first advance on the problem was to understand that there were two separate aspects to the message catalogs. There was the content aspect: the simple text translations, such as changing "Close" to "Fermer". Since the test team was not fluent in the 11 target languages, we had to leave this aspect to the language experts.

The second aspect of the message catalogs was the structure, the syntax rules that a properly constructed target catalog must obey. Unlike content, it would be possible for the test team to verify the structural aspects of the catalogs.

As an example of what is meant by structure, consider the labels and mnemonics of an application menu. A menu is made up of labels and associated mnemonics. Each menu, regardless of its contents or its locale, must obey the following rules listed in the Motif Style Guide [12]:

- Each mnemonic must be contained in its associated label
- Each mnemonic must be unique within the menu
- Each mnemonic must be a single character
- Each mnemonic must be in ASCII

These rules are invariant across locales, and can be used to verify that a menu is constructed correctly in the target locale.

Design decisions

There were several possibilities for how to mistake-proof the menu mnemonics:

- (Prevention device) We could write a program to generate mnemonics automatically, given a list of the labels in each menu. This approach would prevent mistakes, but the problem of choosing a good mnemonic is difficult and the effort required to write the program would not be justified by the benefit gained.
- (Prevention device) We could write a program that would prevent the localizer choosing mnemonics that did not meet the criteria. This approach would also prevent mistakes, but the benefit gained would be minimal; incorrect mnemonics are easy enough to detect and correct after they occur.
- (Detection device) We could provide a program to verify that the chosen menu labels and mnemonics meet the criteria above. Our localizers could

run the programs on their translated message catalogs before sending the catalogs to us. This approach would provide very quick feedback on mistakes, and it is likely as a future step. For the moment however, since we are still developing such scripts, it would be difficult to support them at multiple remote sites.

- (Detection device) We could write a program to verify the menu labels and mnemonics, and run the program on message catalogs after they are returned to us by the localizers. This approach is the path we are currently taking. It is not as efficient as some of the above methods, and it can require communication back and forth with the localizers, but the detected errors are still easy to correct at this point.

The poka-yoke scripts

Several small poka-yoke scripts were used to validate the structural aspects of the menus. The first step was to construct a small table for each menu, showing the locations in the message catalogs of each mnemonic and label in the menu. A typical layout is shown in Table 2.

A small poka-yoke script would read the table, retrieve the mnemonics and labels from the message catalog, and compare the retrieved strings against the established criteria noted above. For example, consider the script that checked whether each mnemonic was contained in its label. The script would read the mnemonic location (e.g., 11, 17) and fetch the string stored at that location in the message catalog ("N"); the script would then get the label location (11, 18) and fetch the string stored there ("New"); finally the script would determine if the mnemonic "N" was contained in the label "New".

mnemonic location	label location	C mnemonic	C label
11, 17	11, 18	N	New
11, 19	11, 20	O	Open
11, 21	11, 22	I	Include
11, 23	11, 24	S	Save
11, 25	11, 26	A	Save As ...
11, 27	11, 28	P	Print ...
11, 29	11, 30	C	Close

Table 2: Locations of mnemonics and labels for Text Editor File menu

All such rules are invariant across locales, so it was a simple matter to run the script against a message catalog in a different locale. An error message is printed for any mnemonic that fails to meet one of the criteria.

We eventually wrote a half-dozen such poka-yoke scripts to verify the rules of the application menus

Results

The poka-yoke scripts were small (roughly 100 lines of Korn shell each).

They were easy to write; some were written in under an hour.

They were easy to run. Because the scripts concerned themselves only with the message catalogs, we did not need to set up a system to execute the application. We also did not need to worry about fonts, synchronizations, hard-to-reach menus, or image comparisons.

We ran our poka-yoke scripts against 16 applications in the default English locale plus 11 foreign locales. Each locale contained 100 menus, for a total of 1200 menus.

The menu utilities found 311 mistakes in menus and mnemonics. Few of the problems we uncovered were earth-shattering, but in total they amounted to a large annoyance in testing and running our localized applications. And though 311 seems like a large number, I encourage you to run similar checks on some of your own localized applications. The results are likely to be eye-opening.

Lessons Learned

Implied rules

By creating and running these scripts, we discovered a rule for menu mnemonics that was not explicitly stated in the Motif Style Guide. One developer used the same mnemonic in the message catalog for items on two different menus. The mnemonic at location "20,100" served as the mnemonic for "Rename" on one menu and for "Reference" on another menu. This arrangement worked fine as long as both labels kept the same mnemonic, "R". But when the German localizer translated "Rename" to "Umbenennen" and changed the mnemonic from "R" to "U", it broke the mnemonic-label relationship on the other menu. We

will eventually refine our poka-yoke scripts to include an inter-menu dependency rule:

- No mnemonic location can be reused within an application.

Handling exception cases

We found that our utilities needed to allow for exceptions. For example, in most locales, the Text Editor Edit menu has a selection to check spelling. This option does not appear on the menu in several of the Asian locales; therefore it is permissible in those locales for that mnemonic and label to violate the usual criteria for that menu. After discussing whether we should artificially enforce the criteria in those locales (which would inconvenience the localizers) or change the code in the utility (which would complicate a rather simple script), we decided to pass the utility's output through a script that would explicitly filter out error messages about that option in the affected locales.

Handling variant menus

Several menus had alternative forms. For instance, the fourth option in the Text Editor File menu in the could have either "Save" or "Save (needed)" as its label, depending on whether changes had been made to the file. We chose to treat these alternatives as two distinct menus to guard against the possibility of a localizer choosing a mnemonic for one variant that would not work for the other.

Conclusions

Poka-yoke scripts like the ones described here can eliminate entire classes of errors. And once the scripts are in place they can run automatically without human intervention, raising an alarm only when a problem is discovered. The scripts we used provided quick feedback early in the process, detecting localization mistakes before the application ever reached the formal testing phase.

The poka-yoke approach proved very flexible, allowing us to validate aspects of the menus early in the development, even without the associated applications. However, it is useful to keep in mind that verifying the menus syntactically is not the same as testing the menus in the application. The poka-yoke scripts did not verify that the menus actually worked; that would require running the application.

The poka-yoke approach provided a simple and robust way for us to detect and correct localization mistakes that would have been difficult to detect through traditional system testing.

Recommendations for Creating Good Software Poka-Yokes

Think ***simple***. It is better to have several simple poka-yokes, each with a single purpose, than to have one large complicated script.

Think ***specific***. Look at your process; identify a mistake that occurs frequently, and design a poka-yoke to prevent or detect that particular mistake.

Think ***attributes***. Rather than wait for the entire software application to become available, look for aspects of the software that can be verified independently.

Think ***early***. Try to detect and eliminate defects as early as possible so that they do not pollute processes downstream.

Think ***responsive***. Once a defect is detected, correct the mistake as soon as possible.

Think ***re-use***. Successful poka-yokes can be modified to serve new purposes.

Acknowledgments

I would like to thank Arne Thormodsen and Sankar Chakrabarti of HP Workstation Technology Center for their helpful discussions of many of the techniques mentioned in this paper.

I would also like to acknowledge John Grout's help in locating many resources relating to poka-yoke techniques. Anyone interested in further information about poka-yoke would do well to visit John Grout's Poka-Yoke Page.

References

[1] John Grout, Mistake-Proofing Production. Cox School of Business, Southern Methodist University, page 2

[2] Shigeo Shingo, Zero Quality Control: Source Inspection and the Poka-yoke System. Productivity Press, page 45

[3] Shigeo Shingo, The Sayings of Shigeo Shingo: Key Strategies for Plant Improvement. Productivity Press, page 145

[4] NKS/Factory Magazine, Poka-yoke: Improving Product Quality by Preventing Defects. Productivity Press, page vii

[5] Richard Chase and Douglas M. Stewart, Mistake-Proofing: Designing Errors Out, Productivity Press, page 29

[6] G. Gordon Schulmeyer, Zero Defect Software. McGraw-Hill, Inc.

[7] James Tierney, Eradicating mistakes in your software through poka yoke. MBC Video

[8] Boris Beizer, Software Testing Techniques, 2nd ed. Van Nostrand Reinhold, page 3

[9] Steve Maguire, Writing Solid Code. Microsoft Press, page xxii

[10] Ian Darwin, Checking C Programs with Lint, O'Reilly & Associates, pages 55-63

[11] Thomas C. McFarland: X Windows on the World. Prentice-Hall, Inc.

[12] OSF Motif Style Guide, Open Software Foundation

Harry Robinson was a software engineer for Hewlett Packard's Workstation Technology Center in Corvallis, Oregon from 1996 through 1998.