Testing interfaces

Alexander Belyshkin and Alexey Lukutin. First published in PC Week, No. 2 (320), 22 January 2002. Translated from Russian; the original is on itWeek.

There are no software developers left who do not test their products. As a rule, though, they check only how the product functions: they look for errors in the code. Errors that degrade a product in use do cost both customers and developers money. But how a product performs in use depends on many other things as well, and one of the most significant is the quality of the user interface.

Interface errors do the most damage to four properties of a product: user productivity (how many units of work a user can produce in an hour), the number of human errors (how often a user gets a particular operation wrong), the speed of learning the system, and the users' subjective satisfaction. Until recently developers treated all of these as secondary, and only in the last few years have some large software makers set up ergonomics departments of their own.

In Russia there are several reasons for software ergonomics to develop. The first and most important is competition, which keeps sharpening and keeps forcing quality up.

The second is that Russian software products are entering maturity. Any product starts out with limited functions and a comparatively simple interface. Over time the functions grow or change, and the interface has to be optimised in step, or no extension will be effective.

The third is the growing number of users with no computer skills. Personal computers are becoming more affordable, so far more people now deal with them, and these people are not inclined to study the fine points of configuring software.

All of this raises the cost of using software. Testing the user interface and then optimising it can help avoid that cost. To keep things simple and concrete, here is a real-life example of profit raised directly by optimising an interface.

A company has developed a home banking program. Technical support for end users is at present a considerable expense (makers of boxed software usually include it in the price). Suppose the product now has 1,500 users. The full cost of an hour of a support operator's work, including salary, equipment, taxes and communications, is $10, and one request takes 20 minutes, a third of an hour. Suppose that optimising the interface prevents at least three support calls per user per year. Multiplying the number of users by the number of calls, the time per call and the cost of the operator gives:

1,500 users × 3 calls × 1/3 hour × $10 = $15,000

So even a minor optimisation of the interface, and the reduction in the example is small, can save the developer a considerable sum in a year. This work almost always improves other properties of the system as well. An ergonomic interface brings sizeable savings

through higher user productivity, fewer human errors, lower training costs and so on.

How it is done

The study can take different forms depending on the task.

The most common form today is expert evaluation. It is economical and quick, and in the shortest time it identifies the main problems and suggests ways to solve them. It rests entirely on the experts' experience and does not involve end users. Because the average quality of software interfaces is low, and most are full of the same kinds of errors, expert evaluation quickly locates and corrects the most serious ones.

When the interface already meets a certain level of quality, more resource-intensive work can be done. One example is an interview, in which an interface designer discusses different aspects of the system with a user. The method needs no special equipment and the procedure itself is relatively cheap. Its weaknesses are limited applicability, since it works only with fairly skilled users, and a degree of subjectivity.

Laboratory testing is often used. Users are given a task, and their work on it is recorded on video, capturing both what they do and what they see on the screen. An interface designer, together with the developers if necessary, then analyses the recording and identifies the interface decisions that caused the most difficulty. Detailed discussion helps to see how the interface should change so that these difficulties do not arise. Such studies differ according to which property of the interface is being tested: speed of work, number of errors, ease of learning, subjective satisfaction and so on. If needed, the experimenter may ask the user to think aloud. To expose the weakest points of the interface, constraints may also be imposed: on time, on how friendly the environment is, and others.

Other interfaces

Most computer users associate the word "interface" with the windowed interface of Windows, but the concept is much broader. One of the clearest examples is the application programming interface, the API. The abbreviation is familiar to programmers who work with function libraries or with CORBA, COM and JavaBeans components. A programmer turning to a library or a component meets the same problems as a user learning a new application. It is hard, for instance, to tell what a function named NeverCallMe is for when it takes 25 parameters that are pointers to data of arbitrary type void*, named a, b, c and so on. Unfortunately such functions are not rare in libraries. An application programmer complicates the user interface over time; a systems programmer complicates the programming interface of libraries, tangling them and breaking the order and logical integrity of the API. As a result application programmers do not know how to use such libraries, make more and more errors and deliver low-quality software. All that was needed was a little more attention to testing how usable the API is.

There are other interfaces too, less hidden from the user than an API but used less often, though they matter a great deal. These are a program's facilities for installation and configuration. They are critically important, especially when there is no system administrator who can type dozens of command-line parameters from memory. Some programs have quite a good user interface in normal operation and then drive the user to distraction at the slightest failure or unusual situation. This is far from rare: recall fighting Windows after a driver was installed incorrectly, or the aftermath of installing a Unix clone by hand. Even so, developers often pay too little attention to how convenient installation is and to how informative the error messages are.

And how much misery configuring software can cause. Modern programs are often shipped not ready to use but as semi-finished products or construction kits. Configuring or adapting the program can then cost far more than the licence. The cause often lies in a poorly thought-out way of configuring and in the low ergonomics of the user interface. Both the customer and the developer bear substantial support costs (see the example above), while the salaries of specialists who can make sense of the puzzle of configuration files grow exponentially. Would it not be better to make the construction kit convenient from the start, or at least to choose the most suitable one on the market? Only independent testing of software interfaces can answer that question objectively.

One last thing. What does an ordinary user do when the program cannot be made to work properly? The user tries to fish something useful out of the documentation. This "interface" has to be of proper quality too. The language, the structure of the material, how easy it is to understand and how easily the needed information can be found all matter greatly to the user and directly shape the overall impression of the product. Well-written documentation often masks shortcomings in the programming. How is this achieved? With well-designed templates, talented technical writers, competent editors and, of course, checking and testing of the documentation.

Conclusions

We hope that this article, brief as it is, will prompt a fresh look at ergonomics in today's consumer software and an improvement in the quality of existing programs, and will make developers think about modernising the way software is produced.