Traditional software usability methods can help us design more understandable and more useful APIs. They also give us information we need to write good API reference documentation—before we invest in either programmers or writers and before evolving a large body of code or content.

Building More Usable APIs

Samuel G. McLellan, Alvin W. Roesler, and Joseph T. Tempest, Schlumberger
Clay I. Spinuzzi, Iowa State University

"Imagine hypothetically, just for a moment, that programmers are humans," writes Steven Pemberton in a July 1997 magazine devoted to human–computer interaction design and development. "Now suppose for a moment, also for the sake of the argument, that their chief method of communicating and interacting with computers was with programming languages. What would we, as HCI people, then do? Run screaming in the other direction...."[1] It is a good question and, unfortunately, an all too common response.

It's hard enough for us to ensure that product interfaces, like those for Excel or Word, are easy to use and learn. But programmers are users, too. They need application and system libraries that are just as easy to learn and use as the products they build from these libraries. Listen to this customer: "I think it would be worthwhile if all developers would spend maybe a couple of hours a year seeing how the[ir] product is used by…customers. Just watching them. And while they're watching … the customer would say, 'I don't like the way this works….' You need to see how they use it."[2] Now ask yourself: why is it easier to visualize the customer who's purchased a financial accounting package from a neighborhood computer outlet, rather than a programmer whose company has just purchased a new Java class library? Wouldn't the developer of this library find it worthwhile to watch programmers work with it?

Increasing productivity is every manager's dream. If we increased productivity substantially, we would cut cost and schedules or deliver more functionality in the next release. Many companies have experienced amazing success in focused initiatives to increase productivity, while others seem to accept their productivity level as a fact of life.

Capers Jones analyzed the results from hundreds of productivity improvement initiatives and found many ways to boost the output of development organizations; the single most effective one was reuse. In his sample, he found that reuse increased productivity by 70 percent (*Software Development Magazine*, Dec. 1997). As added bonuses, delivered defects fell by 85 percent and schedules improved by 50 percent. These are big numbers; it takes a well-defined development process to duplicate these results.

One prerequisite for effective reuse is the usability of the material we want to incorporate in a new system. This article looks at the usability aspects of one of the most frequently considered artifacts for reuse, the actual code libraries. Usability has become a key competitive element for software products: if you buy a piece of software and it has an ugly user interface, you won't use it. The same holds true for code. Developers are known for their tendency to write an application from scratch rather than try to use existing code. This is a natural barrier to reuse. The hurdle is significantly higher if material for potential reuse is cryptic and it takes a lot of effort to understand what it does. If we treat our code as a product needing an easy-to-understand "user interface"—that is, it is well commented and documented—we can reduce this hurdle.

—Wolfgang Strigel

Wolfgang B. Strigel is president of the Software Productivity Centre Inc., Vancouver, Canada; wstrigel@spc.ca; http://www.spc.ca.

In 1997, the human factors group at Schlumberger's Austin Product Center set up a methodology that our system developers could use to quickly test and offer improvements to the usability of a specific application programming interface, and which our writing staff could use as input to the documentation for how to use it.

By application programming interface, or API, we mean any set of higher-level library calls that have been built from lower-level system calls to hide complexity. They make it easier for application programmers to perform certain tasks. For example, one API may let programmers construct oilfield graphic displays, like crossplots or logs, without using low-level graphic primitives. Another API may enable programmers to manipulate and exchange data of a particular type between different oilfield data exchange standards like RODE and SEG-Y without knowing all the details of either.

Testing the usability of reference manuals is certainly not new.[3] However, our methodology offers one key difference and benefit: it tests the object of the reference manual—here, the API—early enough in the development cycle so that recommended changes to the API will more likely be made rather than simply accounted for later in the accompanying documentation.

Field Observations

Like those of many large companies, Schlumberger computer systems can run more than 30 million lines of code. Its worldwide oilfield organization can be thought of as a set of concentric circles. At the center is an engineering staff at the Austin Product Center in Texas, who design and develop oilfield systems and application libraries. Outside this systems center is a circle of other engineering groups. Located in different locations around the world and using the application libraries from its Austin group, these groups develop application products and integrate them into these systems. Outside of these groups lies another circle of field units and business offices that actually use the systems and the products running on them. These groups provide services to oil and gas clients in some 100 countries and feedback to the company's development centers regarding new features, modifications, and fixes.

To check how others' findings (see the boxed text "The Literature" on page 83) scale to large system and application development such as that done at Schlumberger, we spent a few days observing some 20 programmers as they went about their scheduled programming tasks. The programmers were working at one of three different work sites. Observations lasted between 20 minutes and an hour, were recorded in field notes, and checked in subsequent, audiotaped interviews.

A representative dialogue with one application programmer, shown in Figure 1, confirmed extensive use of examples. As we watched, she searched system baselines with personal grep scripts to find application examples using specific calls. She found in-

Question: Am I right to say I notice examples are fairly important to you?

Answer: Yeah. Examples are important because basically they're…they're kind of a parallel proof of concept. Or they show what the boundary conditions [are], what you can do and can't do. Because just the definitions are not appropriate alone. When you want a certain function to do something, definitions are great. If you're scanning through this here, you want to know the constraints of it. But if you want to know its usage, the usage scenarios, you've got to have examples.

Question: So an example helps you see context, different ways you might use some function or functions?

Answer: Oh, it does, it does. You need usage scenarios. Programmers need usage scenarios. Some of the best books are ones that have rich sets of examples. Because there may be cases, usage scenarios that there are no examples for, and you have to experiment to find out what the boundaries of it are. And if there are examples of it supplied, they're worth their weight in gold.

Figure 1. Sample dialogue from field observations.

adequate the documentation for a library she was using. Given her tight schedule, she scanned comments for pieces of code similar to ones she had to write, copied the code into the correct spot, edited the code to use the data structures she wanted to use, then debugged the code. She did this, she said, without really understanding the code but—and this is important—as a mechanism for getting started.

Testing APIs for Usability and Information Requirements

From both the literature and our own field observations, we saw clearly that commented code examples, properly constructed, could help programmers learn very quickly the ins and outs of new API calls and their relationships in context. Too, they offered an opportunity to test and measure the usability of API designs early on.

Usability, as we're using it, measures these product attributes:

- ♦ how easy the API is to learn,
- ♦ how efficiently the API can be used for specified tasks,
- ♦ how easy the API calls are to remember,
- ♦ what misconceptions or errors programmers make using the API, and
- ♦ how programmers perceive the API.[4]

In our case, some programmers would not have access to the code underneath the API. For instance, outside customers may purchase our APIs for use with their own programs. They use library calls but cannot access the uncompiled library code itself, and their own programs might not contain examples of proper calls to all library functions. Thus, we turned to a contrived code example for usability testing purposes.

In the course of one day, we conducted a think-aloud protocol study with four Schlumberger application programmers. A think-aloud protocol requires that a test subject (here, an application programmer) think aloud as he or she works with a particular software interface—here, an example program using an API to move oilfield data from one data storage standard to another.

Subjects

The programmers had profiles matching those for which the API was ultimately targeted. All were familiar with C and C++. Their general programming experience ranged from several to 20 years. Two were familiar with at least one of the data standards and two were unfamiliar with either, but all were familiar with general data exchange concepts. While it was unclear whether programmers experienced with an API would expect or need different kinds of information than those wholly new to the API, the RODE API addresses first-time oilfield company programming groups looking to buy and use a package for one-time use, as well as occasional users who need to reacquaint themselves quickly with the API.

Materials

We constructed the code example so it would perform one of the typical job scenarios identified by multiple oil company clients: moving a specific set of data from one standard to another. We chose this particular job because it would exercise a broad range of library functions—in all, about 75 percent of the API. Using the empirical evidence available, the C program example ran 2,327 lines with descriptive comments, including pseudocode at the beginning of the file to outline the sample program's organization, as shown in Figure 2. The example had not been broken into functions, meaning that the dataflow was contained entirely within `main()`. We reasoned that a program with no functions might be easier to follow.

Procedure

In each session, a programmer was seated at a computer screen and shown, in a text editor, a sample application written earlier using calls from the particular API. We asked the programmer to read through the code, thinking aloud and asking for help wherever and whenever he or she needed further explanation or help. We asked, what would you need to know or understand about this new API to have produced this example? The programmer was free to read the example in any order, to loop through or reread any part of the code, and to take as much time as needed to understand the example. A camera pointing at the screen recorded the programmer's navigation and associated comments and questions. An API expert, sometimes called a wizard in such protocol studies, acted as a kind of help system to answer questions. Each session lasted 45 minutes to an hour.

Finally, at the end of each session, we tested the example's support of analogic reasoning by asking what other general features the programmer might expect this library to provide, based on the single example. Figure 3 shows a quick overview of this process.

Results

It took us several hours to construct the example, an average of 45 minutes to test each programmer, and a half day to transcribe and analyze the data. In addition, we gave each programmer a short questionnaire to help gauge their perception of overall API functional completeness, overall satisfaction with the API based on what they could see from the example, and so on—similar to questionnaires about end-user interfaces like the University of Maryland's Questionnaire for User Interaction Satisfaction.

The results of our study provided insights about two distinct things:

♦ the nature of examples themselves and some of the guidelines with which they should be constructed, and

♦ the usability of the particular API and the nature of its accompanying documentation.

Using examples

We found that an extended example supported various activities but hindered others.

At the end of the tests, all four programmers could describe reasonable features that the library might provide—and, in fact, the API did provide. The code example thus successfully supported a deep understanding of the library's general use. Part of the credit might go to the lengthy pseudocode: all four programmers spent a disproportionate amount of time examining it and asking questions about it. Although the pseudocode's focus was on the code example, it functioned as a way to "show off" the library's capabilities, allowing the programmers to form hypotheses about the library itself as well as the code example. That is, this pseudocode functioned as a "minimalist" description,[5] giving programmers a simple picture of the code's interaction with the library and encouraging them to recall pro-

```
/*
//========================================
// This example tests:
//  1) loading of EFLR attributes with RODE_PutCommonAttr
//  2) encapsulation
//  3) retrieval of EFLR attributes with RODE_GetCommonAttr
//  4) unencapsulation
//  5) multiple instances
//
// This example creates 2 instances of the RODE Object.
//  Instance1 encapsulates a SEGY file into RODE.
//  Instance2 reads the encapsulated file and produces a second
//    SEGY file from it.
//
// Instance 1:
//  1) Opens the RODE session (RODE_OpenSession).
//  2) Populates EFLR values
//     (RODE_PutCommonAttr or RODE_GetTemplate).
//  3) Establishes channels and writes EFLR set data
//     (RODE_WriteInit).
//  4) Loops to read SEGY file and write IFLR data records
//     (RODE_Write).
//  5) Writes eof then eot (RODE_Write).
//  6) Closes the RODE Object (RODE_CloseSession)
//
// Instance 2:
//  1) Opens the RODE session (RODE_OpenSession).
//  2) Establishes channels and loads EFLR values (RODE_ReadInit).
//  3) Displays EFLR values (RODE_GetCommonAttr)
//  4) Loops to read IFLR data records and writes them to file
//     (RODE_Read)
//  5) Closes the RODE Object (RODE_CloseSession)
//
// Input Arguments:
//  This example requires 3 command line arguments.
//  argv[1]  =  The original SEGY file.
//  argv[2]  =  The created RODE IMAGE.
//  argv[3]  =  The SEGY file created from RODE IMAGE.
//========================================
*/
```

Figure 2. The opening pseudocode in the code example outlines the program's organization.

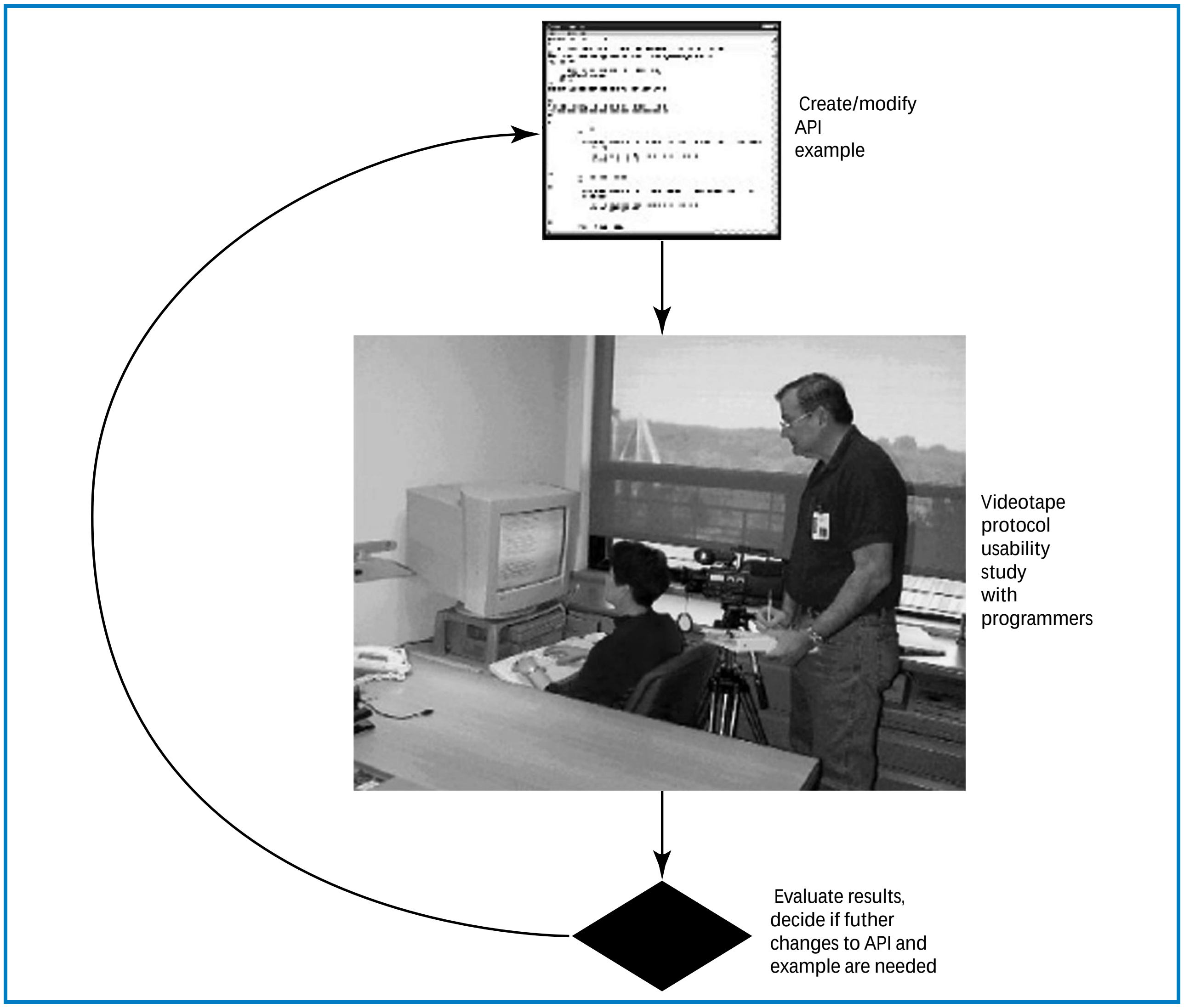


Figure 3. Overview of the API usability test process.

grams and libraries that do similar things. Given the foundational hypotheses they formed using the pseudocode, programmers were able to form deeper understandings of the library than they might have otherwise.

On the other hand, the programmers sometimes seemed uncertain about the usage protocols they were to employ. One of the advantages of code examples, according to the available research, is that programmers can infer a function's usage protocol by examining calls to that function. However, this code's function calls often included identifiers to which the programmers had no access. For instance, the function

```
stat=RODE_GetCommonType(instance2,
FILEHEADER_ID);
```

uses the identifier FILEHEADER_ID, which is defined in the library rather than the code example. The participants were not able to determine what data type FILEHEADER_ID was, or what other information might go into that argument if they were to use this function in their own code. Two of them suggested dealing with this problem by providing a list of function signatures or prototypes.

They also had a hard time building hypotheses at other levels. We found that the example's code occasionally violated what our programmers regarded as discourse rules. As described by Elliot Soloway and Kate Ehrlich, programming is governed by "rules of programming discourse," or "rules that specify the conventions in programming."[6] In our case, programmers tended to guess from function names whether those functions had counterparts. One programmer was confused because `GetCommonType()` did not have a corresponding `PutCommonType()` and `DumpEFLR()` did not have a corresponding `DumpIFLR()`. Another was confused because the specific-sounding function name `GetCommonAttr()` did not

The Literature

We have reviewed some 100 research studies conducted in the last 15 years dealing with program comprehension or production. One commonality concerns using examples. For instance, ViewMatcher, a documentation tool designed to facilitate code reuse, has been used to effectively teach programmers how to use Smalltalk/V functions through short, contrived examples.[1] Lisa Neal describes a similar system based on Pascal.[2] David Redmiles describes a system called Explainer, which uses contrived examples to document Lisp routines and boasts a better learning rate than with a conventional help system.[3] Robert Burow and Gerhard Weber describe a similar project in which they include examples in an intelligent tutoring system for Lisp.[4] Patricia Schank et al. have investigated Pascal template libraries and case studies as an aid for Pascal programmers of various experience levels.[5] Willemien Visser's four-week study of a professional programmer indicated that examples were an essential part of the programmer's work.[6] In a similar one-week study, Beth Lange and Thomas Mohler found that the programmer they studied had an established strategy of code reuse, based on using existing code as a starting template; this strategy allowed her to avoid techniques that require deep understanding of code details or symbolic execution whenever possible.[7] Whereas Bonnie Nardi's users employed examples to gain a deeper understanding of the spreadsheet application,[8] Lange and Mohler's programmer used examples to reduce the complexity of the code and to avoid deeper understanding of it. The latter behavior is exhibited by the programmers in Mary Beth Rosson and John Carroll's study,[1] suggesting that this pattern comes from the resource demands on industry workers.

The prognosis from these studies is consistent and positive. Programmers who use examples gain a clearer idea of a desired product than with descriptive materials alone. They modify and run their programs substantially fewer times before producing correct answers. They opportunistically borrow usage protocol, often without a deep understanding of what an example is doing. The studies also show that, when programmers build plans for new programs or ideas about existing programs, they do so faster and more accurately with resources such as examples.

But we must be careful when constructing program examples in the first place. For instance, Ted Tenny found that comments had a significant effect in program comprehension for 148 students studying a procedureless 73-line PL/I program, but did not significantly affect comprehension of a similar program with procedures.[9] Barbee Teasley provides a thorough review of the research into how naming style can affect program comprehension.[10] Susan Wiedenbeck finds that beacons—elements like comments that help to highlight a critical aspect of code and make initial program comprehension easier—can, when placed inappropriately or erroneously in a program, reduce experts' comprehension to the same level as that of novices.[11]

Extended examples of using an API, then, must be constructed with caution; be careful not to introduce problems that will unduly delay programmers or focus their attention on some aspect of the example rather than on the API itself.

References

1. M.B. Rosson and J.M. Carroll, "The Reuse of Uses in Smalltalk Programming," *ACM Trans. Computer-Human Interaction*, Vol. 3, No. 3, 1996, pp. 219-253.
2. L.R. Neal, "A System for Example-Based Programming," *Human Factors in Computing Systems: CHI '89 Proc.*, ACM Press, New York, 1989, pp. 63-68.
3. D.F. Redmiles, "Reducing the Variability of Programmers' Performance through Explained Examples," *Human Factors in Computing Systems: CHI '93 Proc.*, ACM, New York, 1993, pp. 67-73.
4. R. Burow and G. Weber, "Example Explanation in Learning Environments," *Empirical Studies of Programmers: Sixth Workshop*, W.D. Gray and D.A. Boehm-Davis, eds., Ablex, Norwood, N.J., 1996, pp. 457-465.
5. P. Schank, M.C. Linn, and M.J. Clancy, "Supporting Pascal Programming with an On-Line Template Library and Case Studies," *Int'l J. Man-Machine Studies*, Vol. 38, 1993, pp. 1031-1048.
6. W. Visser, "Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer," *Empirical Studies of Programmers: Second Workshop*, G.M. Olson, S. Sheppard, and E. Soloway, eds., Ablex, Norwood, N.J., 1987, pp. 217-230.
7. B.M. Lange and T.G. Moher, "Some Strategies of Reuse in an Object-Oriented Programming Environment," *Human Factors in Computing Systems: CHI '89 Proc.*, ACM Press, New York, 1989, pp. 69-73.
8. B. Nardi and J.R. Miller, "Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development," *Int'l. J. Man-Machine Studies*, Vol. 34, 1991, pp.161-184.
9. T. Tenny, "Program Readability: Procedures vs. Comments," *IEEE Trans. Software Eng.*, Vol. 4, No. 9, 1988, pp. 1271-1279.
10. B.E. Teasley, "The Effects of Naming Style and Expertise on Program Comprehension," *Int'l. J. Human-Computer Studies*, Vol. 40, 1994, pp. 757-770.
11. S. Wiedenbeck, "The Initial Stage of Program Comprehension," *Int'l. J. Man-Machine Studies*, Vol. 35, 1991, pp. 517-540.

appear to have a counterpart—she even asked about a `GetUncommonAttr()` call.

Similarly, programmers were confused by the code sample's violation of another discourse rule: that numbers should not be hard-coded. Several numbers had been hard-coded into function calls. Although the code ran as it would had the functions been passed symbolic constants, three programmers remarked that the numbers were sloppy or confusing.

When programmers became confused in three tests, the wizard suggested a documentation solution, but the programmers instead suggested a solution within the code. Whereas the wizard wanted the code to be explained, the programmers wanted the code to explain itself.

The code example supported several different learning activities, such as understanding the pur-

Subject: I assume this is a [RODE standard] attribute name, FILEHEADER_ID? [gestures with mouse pointer].

Wizard: Yes.

Subject: And it's defined in the RODE?

Wizard: Correct, in …

Subject: [interrupting] dot H file. So the RODE functions return status, which you should check. [scrolls down and up past several calls writing other attributes]. How do I know, then, the attributes that are needed? Are they all, at least 10 of them, here?

Wizard: That's right, there are 10 that would be discussed [in the documentation]. There are many, many attributes. Some of them can be determined on the fly, like CREATION_TIME. Other ones can be derived from these 10. But these are the 10 units of information that the user HAS to supply.

Figure 4. Questions about needed API information, as derived from the study.

poses of the library, its usage protocols, and its usage contexts. If these activities are not supported directly by contrived examples, then they must be supported elsewhere, such as in reference documentation.

API usability and accompanying documentation

The recorded questions that the participants asked during the sessions revealed general areas and specific instances where redesign of the API or its documentation was warranted.

The particular API under study consisted of some 45 calls related to reading, selecting, and writing data from one oilfield data standard, called SEG-Y, to another, called RODE. At a project design review of the API, consisting of representatives from marketing as well as other program and user groups, very different answers were offered to questions raised about the API and its accompanying reference manual. How much explanation of the data standards and data exchange concepts is required to make sense of the API? Is there any missing functionality suggested by the set of calls in the current API? Are there some calls that will likely cause programmers more problems than others? How can we be sure that clients will be able to use this API easily? These are exactly the kinds of questions the study helped to answer.

Some calls received no questions about usage in at least the context presented. Others caused confusion and raised questions, as earlier shown, for various reasons, which the programmers themselves promptly helped to clarify. Examples include

♦ What does `RODE_CheckUserInputs()` do?

♦ What is `MAXBYTES`?

♦ Where does `DumpEFLR` dump the information?

♦ Do you have an instance where I could see what the `RODE_GetTemplate` template physically looks like?

In other cases, understanding calls such as `DumpEFLR()` and `DumpIFLR()` required only that RODE bulk data storage concepts like Explicit Logical File Record and Implicit Logical File Record be explained in the context of the other data being read or written, so that programmers knew when and where these calls were to be used.

In another instance, a routine was used to write a particular set of data from one standard to another. In some confusion, all four programmers asked what was minimally required versus what was optional for moving data between standards, what other data might be available, and whether the routine could be used with these data types (see Figure 4). Taken together, the list of questions about data standards showed what programmers needed to know about data storage to use the API effectively:

♦ how data from one standard was related to data from the other,

♦ what functions were related to what data of interest, and

♦ any required or recommended order for these functions.

We also found semantic overloading between concepts in C++ and concepts in one or both of the data standards, either in the comments provided in the example or in the answers provided by the wizard to programmers' questions. In both situations, programmers needed the distinctions made clearer. For example, the term "template" is used in RODE library calls but means something very different for C++ programmers.

In some cases, a violation of discourse rules seemed to dictate redesign. For instance, call twins like `GetTemplate()` and `PutTemplate()` or `GetAnyAttr()` and `PutAnyAttr()`, as opposed to single calls like `SetTapeSummaryData()`, raised questions about reasons for such mismatches. If these types of questions are not resolved by direct changes in the API, they must be addressed in the accompanying documentation.

The participants also questioned the API's design. For instance, three of the subjects didn't like a low-level general function that was used 10 times, each time with a different required input. They instead preferred a function with all 10 required inputs, as shown by one dialogue exchange in Figure 5.

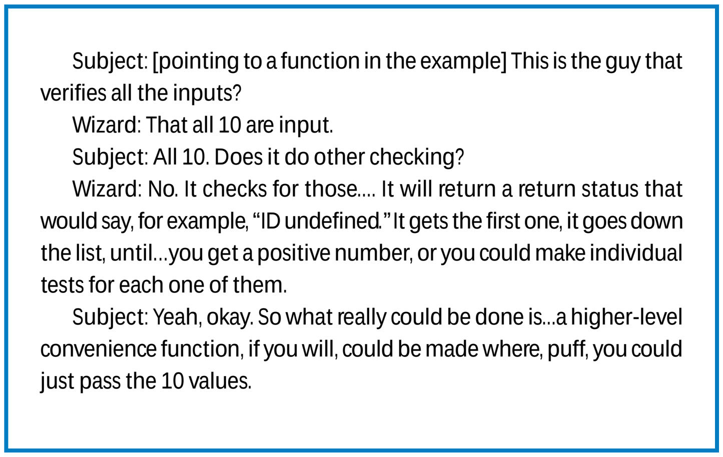
In some cases, the example lacked anticipated functionality present in the API: for example, most of the subjects expected positional functionality, namely the ability to locate and position at particular data points, something especially useful for selecting data for writing. On the other hand, the programmers provided several justifications for other calls—such as the ability to filter data based on some initial criteria—which were not available in that API.

All in all, these and other questions showed what users, namely application programmers, would need from documentation, examples, or manuals when they first encounter this particular API.

Our final report enumerated recommendations for changes to the API and the code example, as well as observations about elements that appeared to help test subjects understand the API or the example. In particular, we looked for patterns, such as

- questions or issues that at least three participants raised as they thought aloud,
- comparatively long timespans spent perusing certain code segments, and
- the number of questions related to particular calls or code segments.

Based on this report, the product team responsible to the API's clients made some decisions regarding changes. As with end-user interface development, we recommend one or more iterations to test the revised API with the revised example; this may demonstrate early on the overall effect of changes to user performance or perception. With time constraints for commercial release of the API, we decided that we could retest some individual changes quickly such as renaming calls to help clarify their purpose, eliminating mismatches, or satisfying programmer preferences by iteratively adding or changing calls and running these by other programmers until meaning and use were clear. Just prior to commercial release later the same month, we field-tested the implemented code along with the reference documentation and example. This went smoothly, eliciting only one change to the API: the addition of a call to handle a particular function from one oil company client.



Subject: [pointing to a function in the example] This is the guy that verifies all the inputs?

Wizard: That all 10 are input.

Subject: All 10. Does it do other checking?

Wizard: No. It checks for those.... It will return a return status that would say, for example, "ID undefined." It gets the first one, it goes down the list, until...you get a positive number, or you could make individual tests for each one of them.

Subject: Yeah, okay. So what really could be done is...a higher-level convenience function, if you will, could be made where, puff, you could just pass the 10 values.

Figure 5. Study participants questioned the API's design.

Testing API designs using contrived examples should not be considered the only or best method of doing so. Available resources, the evaluation's purpose and context, what research says about the yield of multimethod strategies, and the like must all be accounted for.

In addition, our results based on four programmers must be combined with those from additional studies with other APIs. We suppose that some general claims about API testing and documentation can be made independent of the particular programming language used for the API. As more case studies are performed, related but currently open questions will likewise become easier to answer. Does the kind of API make a difference? That is, will testing a graphics API reveal different types or presentations of information that programmers need to know about? Is it possible to begin to categorize the information programmers need or, from this, to develop principles that might guide developers or documenters in the best ways to design an API or represent its information? These are legitimate questions that warrant further study.

What we can say is this: with examples, testing APIs up front can offer early indicators about usability to both API developers and API documentation specialists. In effect, they will see exactly if and how their application programmers will understand and use their APIs, in the same way that usability testing of user interface designs shows if and how users will work with these products. In particular, the results of such usability tests should lead to iterative API redesign and testing, where justified, so that API libraries are validated by the kind of programmers who will use them, and API reference manuals do not become repositories just for the deficiencies in API design.

The programmers in our study raised additional questions beyond the scope of this work. For example, Schlumberger is automatically generating some reference information from the library code itself using commercially available compilers. Here's the way it works: the compiler extracts call and comment information, such as call prototypes. Then it produces, according to the predefined document template, a hypertext-indexed, online reference manual of library information. This information, available online to the company's programmers via an intranet and any Web browser, is always accurate and in sync with the code because it is derived from the code itself.

But, of the information that programmers need to understand and use an API, how much can be reasonably placed in the code—for example, in comments—so that it can be automatically generated as part of the reference manual? How can we best link the information between these examples and reference manuals so that programmers can efficiently get answers to the questions they have? Is it possible to categorize, and thus anticipate, the kinds of questions that programmers ask about APIs? These are a few questions that our usability group is looking at now.

We have found usability testing of APIs to be an effective method for developing APIs and determining useful information for reference documentation. As Gary Perlman has said, "It's difficult to develop good user interfaces. We know this because there is no shortage of bad user interfaces, even in products where developers tried to incorporate 'user friendliness.'"[7] The same goes for APIs. ❖

References

1. S. Pemberton, "Programmers are Humans Too, 2," *SIGCHI Bull.*, Vol. 29, No. 3, 1997, p. 64.
2. S.E. Poltrock and J. Grudin, "Organizational Obstacles to Interface Design and Development: Two Participant Observer Studies," *Human-Computer Interface Design: Success Stories, Emerging Methods, and Real-World Context*, M. Rudisill et al., eds., Morgan Kaufmann, San Francisco, 1996, pp. 303-337.
3. S. Rosenbaum and R.D. Walters, "Design Requirements for Reference Documentation Usability Testing," *Proc. 1987 IEEE Int'l Professional Communication Conf.*, IEEE, Piscataway, N.J., 1988, pp. 151-155.
4. J. Nielsen, "Usability Metrics: Tracking Interface Improvements," *IEEE Software*, Nov. 1996, pp. 12-13.
5. J.M. Carroll, *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*, MIT Press, Cambridge, Mass., 1990.
6. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Eng.*, Vol. 10, 1984, pp. 595-609.
7. G. Perlman, "Software Tools for User Interface Development," *Handbook of Human-Computer Interaction*, M. Helander, ed., North-Holland, Amsterdam, 1988, pp. 819-833.

About the Authors

Samuel G. McLellan is a research scientist with Schlumberger in Austin, Texas, specializing in human-computer interaction design and usability.

McLellan received a BA in mathematics from Texas Western, an MA in English from the University of Exeter, UK, and a PhD in English from the University of Texas at Austin.

Alvin W. Roesler has been an engineering specialist with Schlumberger in Austin, Texas, for more than 20 years, with a specialty in human interface design standards and process usability.

Roesler received an MA in mathematics from the University of Texas at Austin.

Joseph T. Tempest is a project engineer with Schlumberger in Houston, specializing in systems and application product design and development.

Tempest received a BS from Purdue University.

Clay I. Spinuzzi is a PhD candidate at Iowa State University, focusing on human-computer interaction. In the summer of 1997, he worked at the Schlumberger Austin Product Center as an intern.

Spinuzzi received an MA in technical writing from the University of North Texas.

Address questions about this article to McLellan at Schlumberger, PO Box 200015, Austin, TX 78720-0015; smclellan@slb.com.