Discussed is a methodology of creating and using scenarios to assess completeness, correctness, consistency, and usability of the external design of computer software. Scenarios are paper tests of the specifications of software being designed. The approach is an outside-in, user-oriented evaluation of programs. The technique requires no machine time to perform the evaluation. As a result, defects are identified and changes are recommended early in the design phase of software development, at the time when defect removal costs are lowest.

Technique for assessing external design of software

by R. J. Pearsall

Software development has primarily focused on advancing the state of the computer programming art by adding new functions and performing existing ones at faster rates. These accomplishments have, for the most part, been attributed to technological innovation. Now, more attention is being given to the usability of software.

Substantial progress is being made in removing defects from software during the software engineering process. Indeed, specific steps in most software engineering processes allow for the removal of various types of defects from the design and resulting code. The scenario technique described in this paper complements this effort by identifying defects in a program's external design, thereby allowing for the removal of these defects prior to program development.

Computer applications are changing significantly, placing new demands on programming. In the past, users of programs were primarily data processing professionals. Today, users are often persons who may not have a data processing background. These non-data-processing professionals relate to programs in terms of usability for handling their applications. This creates new requirements for easy-to-use external interfaces. Users of software see it collectively as tools to help them accomplish their jobs, and they may or may not decide to use a particular tool, depending on its usability.[1] Usability is thus a key consideration for developers of software.

This paper describes a technique that uses *scenarios* to evaluate the external design of large-scale, general-purpose programs. A scenario is an English language exposition of the interaction of specified programming and a particular application, expressed as what the user does via job streams and especially what the user sees as a result of the action. Functional specifications are used to create these scenarios, and the technique provides a means for evaluating completeness, correctness, consistency, and usability of the design as defined by the programming specifications. A scenario uses descriptions of macros, commands, and publications such as reference manuals to create job streams that perform the various tasks in a representative user environment. This approach gives a user-oriented, outside-in look at a program. Hence, the scenario technique provides paper tests of programs under development before coding begins.

Scenarios offer several advantages. Functional defects can be found when the scenarios are used to evaluate the external design. Also, finished scenarios help to provide the developers with feedback on the acceptability of programming externals. Scenarios can further aid in preparing for system-level function testing, and we have used them to assist in the production of program documentation.

Problems addressed by the scenario technique

We refer to the steps in going from application requirements to the finished program collectively as the *software engineering process.* We have added the scenario technique to this process[2] and applied it to the development of data base management systems and other programming for the evaluation of external program interfaces.

Among the existing variations of the software engineering process, the method with which we are most familiar calls for the external design of software to be described in a document known as the Final Programming Functional Specifications (FPFS). The software requirements are contained in a document known as the Programming Objectives (PO). These two documents are primary inputs to the creation of the scenarios.

command completeness

An FPFS is normally organized and described by program function or component. Also, different people are often assigned to design each program function. Consequently, it is difficult to determine the completeness of the commands provided to perform the designed functions or the completeness of the set of functions offered to perform the tasks. The scenario technique provides for identification of user tasks and evaluation of design in terms of those tasks, thereby testing the completeness of commands and functions.

usability of external languages

Another problem the technique addresses is that of the difficulty of assessing the usability of proposed external languages by merely

studying syntax descriptions in reference manuals. A syntax may look usable at first glance, but a program that uses mixtures of blanks, parentheses, hyphens, and slashes as delimiters, for example, makes it difficult to write command streams without syntax errors. Lengthy keywords and nonstandard keyword abbreviations do not appear to be difficult to use until one tries to spell them correctly and enter the data on lines limited to eighty characters. One does not realize such difficulties until he starts using the system.

command consistency

Still another necessity addressed by the scenario technique is that of consistency of commands in terms of content and format. This need pertains not only to commands within the program being designed but also to commands between that program and related programs that a user must apply to accomplish desired tasks. For example, a program that defines DROP as a command against objects of type A and ERASE against objects of type B embodies an inherent difficulty for the user. For the same reason, one who is using MVS and who is also using TSO interactive facilities may find a program difficult to use correctly where two commands mean the same thing. For example, the user of the DELETE command for getting rid of objects managed by TSO may be confused by an ERASE command that performs the same function in a program that runs under control of a TSO session.

English and programming languages

Command verbs within a single program are sometimes so much alike in their English language definitions that a user has trouble remembering the verb to use in a given computing situation. Consider a programming language that offers the similar English language commands DISPLAY and SHOW to operate on the same set of objects, but defines the programming language command verbs as performing very different operations. This may lead to quite different and erroneous results. Complementary to this is the case of similar English and programming language verbs that operate on different objects. An example of such a design flaw is that of defining SHOW as valid for objects of type A only and DISPLAY for objects of type B only. With either design, the user faces unnecessary confusion between the English and the programming language.

reasonableness of operands

Reasonableness of command operands in terms of the maxima, minima, and granularity that they provide cannot be evaluated without understanding other programs that are available to the user in his environment. An example of reasonableness is that of a buffer manager that allows buffers to be specified in integral multiples of the page size only. Such buffer specifications can be used by the storage manager and by access methods to minimize the number of physical I/O operations. An example of an unreasonable maximum is that of a buffer manager that limits a user to writing 1K-byte records when the underlying access method (or another program) currently in use allows 4K-byte records to be written.

abbreviations and defaults

An analysis of abbreviations and defaults should also be performed as a part of the evaluation of software externals. The usefulness of choosing to abbreviate or default and the value or term chosen deserve careful attention. For example, the number of Write To Operator (WTO) buffers provided by MVS was at one time an order of magnitude too small to operate efficiently. As a result, many MVS users were forced to override that default. An understanding of related programs available to the user in his environment is therefore key to evaluating this aspect of the design.

system action messages

Another important consideration is to ensure that system messages go to the right persons. For all commands, precise messages should be provided to indicate the actions performed by the software at various stages of a job execution. These messages should report the successful and unsuccessful execution of commands. Unanticipated events should also be reported. Systems problems are difficult for many software users to determine and often require the attention of individuals with specialized skills for resolution. Good external design can provide data necessary to minimize the expense and lost time associated with problem determination. Scenarios attempt to identify everything that can go wrong with resources managed by software so that good diagnostic messages can be built into the product and sent to the proper user.

effect of program internals

Command verbs, keywords, and operands often coincide with internal objects of programs. This tends to make program externals appear to be internally oriented (i.e., reflecting implementation algorithms and structures), which may generate resistance to the use of the program.[1] Terms that are familiar to non-data-processing professionals and convey the same information ought to be used in system messages, contrary to the tendency for the externals designer to use the same names as the internal supporting construct. Because the software designer does not have the same perspective as the intended user, an outside group can help to highlight areas where internal program objects are presented to the end user.

One analysis of a product using the scenario technique found that users were presented with names given to collections of components within the program via commands and messages. The names had no meaning and provided no additional useful information to the user. These names, which had been assigned to groups of components for the purpose of partitioning the code by department and functional area for development, were identified and subsequently removed from the externals.

command ambiguity

The scenario process also evaluates a program design to identify externals that at times are not indicative of their corresponding function. For example, the CREATE verb was once used to bring objects into existence. The CREATE verb was also used where it added

information to an existing object but did not result in the creation of a new object. This example illustrates an ambiguous use of a command verb.

human factors

Often, insufficient attention is given to seeing commands and the resulting response messages from the viewpoint of an intended user. Just as publications can be difficult to read when they are not written for the intended audience, program externals can be similarly misdirected. The scenarios segregate program externals by user task, which allows an analysis of the acceptability of the external design as viewed through the experience of the users associated with each of the defined tasks.

Few programs developed today run independently of other software. This complicates the design evaluation in that one must look beyond the bounds of the program being evaluated in order to obtain a system-level perspective. One must consider the interaction of a given program with other programs. A user must deal with compilers, data base managers, operating systems, access methods, session control managers, and so forth, to accomplish certain tasks. The better these programs work together, the better they serve the user.

For example, consider an MVS system where the user applies TSO editing facilities to create batch job utilities for later execution. Suppose that for an MVS utility a hyphen were chosen as the last character on a line to indicate continuation of the input data stream to the next physical record. With TSO, the hyphen indicates to the editor that the user wishes to skip to a new physical line on his terminal, but the new line is to be concatenated to the record currently being entered. As a result, the task of creating input data becomes overly complex because the two programs do not work well together. Users have developed techniques to avoid this problem, but it might have been obviated by the use of scenarios during the development of the system.

Creating the scenarios

The programming areas just discussed have motivated us to develop a scenario methodology that we divide into five sequential steps: (1) establishing objectives; (2) defining representative environments; (3) developing scenarios; (4) reviewing scenarios; and (5) identifying problems.

To develop scenarios we have formed a group made up of members from areas responsible for program testing, manual publications, programming assurance, world-wide applicability, programming education, system-level environment testing, and software planning. Designers of programs being evaluated are excluded because their inside knowledge, rather than the written specifications, may

influence their contributions to the scenarios. They may, however, be consulted for clarification of specifications.

Our primary objective is to identify and remove external design defects by constructing an outside-in evaluation of proposed software, keeping in mind the user's perspective of the external design. We also believe that scenario development can increase the participants' knowledge of the software. Thus we believe that if our work is successful the resulting scenario document will prove beneficial in analyzing the usefulness of a proposed program, in preparing support documentation, in preparing system-level function testing plans, and in fostering an understanding of external designs by upper-level executives.

scenario document

As a first step, the potential defects previously described are explained to the team members to sensitize them to the types of problems to look for. Also explained is the job of translating programming externals that are specified in an expository format, together with a sample user environment, into job streams to be run against a proposed program. The job streams are to be set forth in a document called the *scenario document*.

task identification

The next step is to partition the work of writing scenarios by analyzing program contents and the intended users' tasks. Teams of two persons are assigned to write scenarios for each of the defined tasks. In the scenario document that we produced for a data base system we identified the following tasks: system installation, system operations, system recovery, application development, data base administration, system security, and system optimization.

task outlines

The third step is to outline each task by extracting functions (line items) from the Programming Objectives (PO) and from the Final Programming Functional Specifications (FPFS) and to bring them together as task outlines. The resulting outlines bound and define the functions of each task. For example, the recovery task outline contains five major items: (1) failure of system code; (2) system data set errors; (3) data base manager log errors; (4) user data base errors; and (5) recovery utility errors. Each major item is further subordinated. From the data base system example just given, failure of system code is further refined to address: (1) operating system abend (abnormal ending), wait, or loop problems; (2) data base manager loop, wait, or abend; (3) failure of a data-base-manager-dependent program (such as the resource lock manager; (4) failure of a data communication control region; (5) failure of a dependent message processing region; and (6) failure to respond to an end user. The completed outlines are reviewed by fellow study group members who focus on identifying items omitted from any of the task outlines and items that logically belong to other tasks. Outlines are updated to reflect such changes.

task narratives

The next step is to expand task outlines into what we term the "narrative form." This step involves the expansion of each topic in each task outline into an English language narrative of what must be done to perform that aspect of the task. In the recovery task, for example, the outlined item entitled Data Base Manager Failure (loop, wait, abend) is expanded to discuss symptoms indicating which problem had occurred and the action to take to correct the problem. Depending on the error situation, the symptoms are described in either informational or error messages that appear in one or more of the following places: user terminals, operating system master consoles, records written to the data base manager's log, or in an operating system log.

The functional descriptions of these symptoms and corrective actions are extracted from FPFS. In the event that symptoms and/or corrective actions are not stated in the FPFS, the writer of the narrative proposes a problem log statement derived from the external design of the program. Consider, for example, an event in which no indication of the data base manager's stopping or restarting is sent to the master terminal operator responsible for data communication and message processing. In this case, the master terminal operator first sees queues of work building; then he sees the system go into recovery mode. He may eventually determine what has happened, but he may never learn via messages when the data base manager is ready for processing. As a result of analysis, a design change was incorporated to inform the master terminal operator of stopping and restarting events.

The creation of the narrative level documentation is a rather complex job, requiring a great deal of interpretation to show the program control flow. This interpretation is necessary because messages in the FPFS generally are not tied to specific commands. The writers of the narratives are forced to make informed guesses in describing the flow. For this reason, the narratives are reviewed by the program designers to verify that the interpretations conform to the designers' intentions. Questions related to incompleteness of the FPFS are included in the narratives so that they may be reviewed by the designers. The designers either provide the answers to questions in the FPFS or give the answers when they are not contained in the FPFS. Lacking such information, the designers state that they cannot answer the questions because the design work is incomplete. Whenever a question is answered by a designer or the design work is found to be incomplete, the FPFS is amended and the narratives are updated to reflect the comments from the designers' review.

user environment

Before the narratives can be considered as complete, they are compared with a representative user environment that contains descriptions of user data bases, applications, transactions, user IDs, and associated names that are to be used in and be common to the scenarios produced in the next step. In the case of our data base

system scenario, the IMS/VS primer[3] contains a sample problem that we used with modifications to study applications, data bases, and so forth to exercise those aspects of the system that we were evaluating.

scenario creation and review

To create the actual scenarios, study group members write such programming as the user-initiated instructions, commands, and JCL for each narrative. This programming is required to perform the work described by the narrative for the sample environment and for all of the system responses to the user. The scenario writers extract the command syntax and messages from the FPFS, and parameter names and values from the user environment definition appendix of the scenario document in order to create job streams. The resulting job streams are the solutions to the defined user tasks for the sample environment. When commands and messages necessary to perform a task are not contained in the FPFS, design change requests are submitted. Design change requests are also submitted when usability concerns are recognized in the provided externals. For example, in using commands to accomplish a particular task, we recognized that a particular set of commands would be more usable as subcommands of another specific command. This eliminated the requirement that a user temporarily interrupt an existing terminal session to enter a command that logically belonged to the session.

The scenarios are subjected to a second review by the system designers that focuses on verifying the externals used to accomplish the tasks. Again, questions are placed in the scenario document when the writers cannot find the appropriate external in the FPFS. The scenarios are finally updated to reflect approved design change requests that affect the externals of the program, plus changes attributed to incorrect interpretation of the external design as defined by the FPFS. The format of and actions resulting from the scenario review by the system designers are the same as those described previously in the narrative review.

Concluding remarks

The final version of a scenario gives an accurate view of the program through the eyes of a user. The scenario also gives management a useful view of the system. Thus the scenario becomes the first vehicle to give flow to the program externals, allowing an examination of the externals as they apply in a user environment. Anyone should be able to use a scenario to evaluate a proposed program for usability, compatibility, consistency, and completeness of functional content. The difficult job of transforming the contents of reference manuals and sample environments into task-oriented job streams need be performed and validated only once. Thus program testers, publication writers, and others can use the same scenario, rather than each group's having to generate similar information for its specific requirements.

We have found that scenarios may also be applied in the following situations:

- Educating program development support groups.
- Creating system-level function test procedures.
- Deriving publication objectives and plans.
- Producing marketing review board inputs.
- Obtaining users' opinions, approval, and feedback prior to the development of running code.

In our first application of the scenario technique—a data base system—a number of problems were identified in the program functional specifications and were removed. The defects found involved usability, completeness, correctness, and consistency of system externals. The scenario process provides an outside-in look at the system prior to the actual creation of the running code to support the design. The evaluation of the system required no machine time, and the technique was applied sufficiently early in the program development cycle to remove defects at minimum cost.

In programming projects where scenarios have been written, the scenario writers have found a number of defects that had previously been overlooked. The scenario technique has thus proved to be an effective discipline for exposing external design defects. Initial planning work for system-level functional testing has been developed using scenarios as primary inputs. Publications and planning groups have also made use of scenarios as primary inputs. Programming executives have increased their understanding of programs under development, and marketing executives have used the scenarios in making their evaluations of programs. We believe that the scenario concept is suitable for incorporation in all software engineering processes, particularly in the development of large and/or complex interactive systems.

CITED REFERENCES

1. J. L. Bennett, "Incorporating usability into system design," *IBM GPD Design '79 Symposium Proceedings* **1,** San Jose, CA (April 1979); a copy of the paper may be obtained from the author of the present paper.
2. H. Remus, *Planning and Measuring Program Implementation,* Technical Report TR03.095, IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, CA 95150 (June 1980).
3. *IMS/VS Primer,* World Trade System Center Bulletin No. S320-5757-2 (September 1977); available through IBM branch offices.

The author is located at the IBM General Products Division Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, CA 95150.

Reprint Order No. G321-5167.