TI-MIX 1983 International Symposium

April 5-8, 1983
New Orleans Hilton Hotel

Session Proceedings

Software & Systems Engineering

TABLE OF CONTENTS

TI-MIX
M/S 2200
P.O. Box 2909
Austin, Texas 78769
(512) 250-7151

AN ANALYSIS OF ERRORS

BY
JIM FISHER
I.C. SYSTEM, INC.
ST. PAUL, MN

PREPARED FOR THE 1983 TI-MIX SYMPOSIUM
SYSTEMS AND SOFTWARE ENGINEERING SESSION

Today's data processing professional is very well aware that hardware is getting cheaper while software is getting relatively more expensive. Although it is often overlooked, the major component of this software cost is incurred not during the development phase, as one might expect, but during the maintenance phase of a project. The cost of correcting mistakes and making changes to a system is too often from 2 to 4 times the initial development cost. Unfortunately, the average EDP organization spends 50 percent of its DP budget on ongoing debugging and modification of existing systems [1]. Software engineering techniques are aimed at reducing this cost and at increasing the quality of the final product.

This paper will take a look at software errors, where they are typically found, their causes, and the cost of correction at various stages in the development life cycle. If we take a good hard look at software errors and gain an understanding of their living habits, it becomes easier to develop standards and techniques aimed at reducing the overall life cycle costs of a project.

RESULTS OF VARIOUS STUDIES

A number of empirical studies have been made over the past 20 years investigating various aspects of software economics. Starting from a very broad view, we Americans spent about 40 billion dollars on software development and maintenance in 1980. That's about 2% of the Gross National Product [2]. And much of it appears to be spent on non-productive activities.

For example, one study indicates that of that money we spend on software, only about one-fourth of it is spent developing something new [3]. And when we do embark on a new development project, we risk a 1 in 7 chance of failure. That is, for every 7 projects started, one of them will never deliver anything.

Maintenance, as defined by most of these studies, involves both debugging and making modifications. Debugging, as we all know, means attempting to correct the cause of software failures. We are all aware of the cost of fixing mistakes. But we should also keep in mind that the need to add a feature or make a modification to a system is also a form of mistake. It is a mistake in the analysis of what the user needed. In order to understand the cause of these errors, we must begin by realizing that it can easily be 10 to 100 times more expensive to add a feature to a system than it would have been to incorporate that feature into the original design.

THE SOFTWARE DEVELOPMENT EFFORT

For most typical large scale projects, about 45% of the effort is spent on analysis and design, about 20% on coding and unit testing, and the remaining 35% is spent on integration and installation [3,4]. Ask a typical programming manager how these costs break down and you may get a surprisingly different answer. The typical manager sees the cost of ownership to be fairly low during the analysis and design phases. After all, only a few people are involved during this stage. He would expect the cost to peak during the coding phase where there are obviously more resources being used. He also will consider the project done when the system is installed and operational.

What the typical manager doesn't realize is that these costs are, in reality, only the tip of an iceberg. When he thinks the project is done, only one-third to one-fifth of the money has been spent [4]. To handle the maintenance costs of a typical system throughout its 5 to 10-year life-cycle, he should automatically budget 3 or 4 dollars for every dollar actually spent during its development. And in some companies, this is actually done as a matter of course. If you spend $20,000 developing a system, plan on spending at least another $5,000 a year for the rest of the system's lifetime.

A more realistic cost of ownership curve not only peaks higher than expected, but also later in the life cycle. More money is spent in the coding and debugging phase than anyone would expect. And the operational costs can be anywhere from 4 to 10 times more than what would be expected.

PROPOSED COST OF OWNERSHIP

The reason the actual curve has this shape is that we all have a tendency to rush into things. During the analysis and design phases, perhaps only 3 to 5 people are involved; whereas during the coding and debugging phases, 15 to 30 people may be involved.

Imagine the president of a company walking through one department and seeing 3 analysts quietly drawing models of a system. He looks at them and thinks, "Hmmmmmm". Then he walks into another department and sees 27 programmers pounding feverishly on their keyboards, and says "Ah, Haaaaaa". In his mind, there is <u>obviously</u> more work being accomplished in the second department and they are <u>obviously</u> much farther along, right? This should not be all that <u>obvious</u>. The analysts in the first department may indeed not only be accomplishing more but also be farther along than the programmers in the second department. We have to keep in mind that the analysts are working at a much higher level and that the quality of their work will have a much greater impact on the total cost of the system. One bad analysis of a user's needs will cost you much much more than one bad programming job.

Our proposed cost curve emphasizes an increase in expenditure during the initial phases of a project. Spend more time determining what the user really wants, and how his needs are to be solved. The primary reason for this emphasis is to catch and eliminate errors as early as possible.

WHAT IS AN ERROR?

The Ballistic Missile Early Warning System was designed to detect and monitor any object moving toward the United States, and if the object was unidentified, to initiate a sequence of defensive procedures starting with attempts to establish communication, establish the identity of the object, and so forth. An early version of this system mistook the rising moon as a missile heading over the northern hemisphere [3]. Is this an error?. In the eye of the analyst, it may not have been, because it followed the letter of the design. The moon is an object. It does, on occasion, move toward to the United States. The system did not however, follow the user's conception of what was intended, and this is important.

DEFINITION OF ERROR VS. FAILURE

A software error is present when the software will not do what the user reasonably expects it to do. Using this definition, then, a software failure is simply the manifestation of a software error. Software errors are always there, but you don't see them until the software fails to perform.

Note that the definition does not say "what the system was designed to do", rather it emphasizes what the user expects the system to do. With that in mind, let's take a look at what today's users are expecting of a well-designed system.

REASONABLE EXPECTATIONS

The first thing that a user may reasonably expect is that the system work. That it not lose track of data, that it perform calculations properly, and so forth. These expectations are so obvious that I have chosen to eliminate them from this discussion. Today's more sophisticated user has many additional expectations which may not be terribly obvious.

Today's user may reasonably expect a system to provide needed information in a most usable format at the proper time. A profit and loss statement that comes out 6 months late is not very usable. And if it's buried in the middle of a 100-page financial report, it's not very usable either. Also, if management information is only available on the terminal and no hard-copy can be printed, the information is useless to someone who might want to take it home and study it.

He will also expect a system to strain out data that is not usable to him. Any good analyst will make a distinction between what the user thinks we wants and what he really needs. There is a fundamental difference between raw data and information. Information is data which has been given a meaning, a usefulness to a particular user. A good system will provide that information while "hiding" the unnecessary detail data.

A system, nowadays, may also be expected to react to what the user wants to do when he wants to do it. The system should be responsive to user demands rather than forcing the user to fit the mold of a system. With command or menu-oriented systems, we are heading in that direction, and we must continue.

A system should also produce at least a "pound" of results for each "ounce" of effort. If we have an automated filing cabinet system where it take the user 10 seconds to describe that he wants a phone number, all we have done is totally waste the resources of the computer. The user would be better off with a Rolodex. If a user has to enter the name and address every time an invoice is generated, again, we are wasting our time.

It is very important that a system guard itself against, or at least be tolerant of mistakes made by the user. It should guard against accidental deletion of required information. It should always provide the user with a simple means of getting out of something he didn't mean to get into. The CMD key is a perfect example of this. We must also provide the user with natural and effortless ways of correcting his own mistakes. There is no excuse for not providing for an "up arrow" function. But we can go beyond that by providing "oops" keys to undo something, "go back" keys to backup to a previous step, and even "what if" keys to allow the user to try something without having his actions affect any "live" data.

We should also be able to predict what the user may want to do next. This may be as simple as the intelligent use of default responses to questions. It can, however, go way beyond that into the areas of artificial intelligence.

Lastly, the system must be modifiable at a reasonable cost. We should not only provide the user with the capability of changing his mind during execution, but also to provide him with the freedom to change his conception of what his needs are. His conceptions will grow and mature along with his use of the system and we should be ready to grow along with him.

DISTRIBUTION OF ERRORS

In a study reported by SofTech in 1976, they discovered that about two-thirds of all software errors are to be found in the analysis and design phases of a project [4]. Only about one-third are the result of coding errors. This may surprise many of you. But, if we stop and examine the root cause of these errors, this distribution takes on a great degree of credibility.

ROOT CAUSE OF ERRORS

Most errors are the result of errors in translation [3]. A translation occurs when one person attempts to communicate his conception of what the user wants to a second person. In any such translation, errors are likely to occur. Realizing that translation errors are the reasons for software errors is extremely important because it describes the underlying causes of unreliability. We should identify when a translation occurs and provide some sort of testing mechanism to ensure that the process was performed completely and accurately.

This translation model describes how the vast majority of software errors originate [3]. This model is often viewed with surprise by the uninitiated, for their view is often that software errors are errors made by a programmer while writing the source code. Although these errors do exist, they represent a small percentage of the errors in a system and are usually easily corrected.

COMPLEXITY

Complexity, being a principal underlying cause of translation errors, is one of the major causes of unreliable software [3]. The complexity of something is simply some measure of the mental effort required to understand it. Complexity varies somewhat from one individual to the next. What a system programmer might consider to be simple, may be incredibly complex to the average user. Since complexity is the root cause of translation errors which, in turn, is the root cause of software errors, we need to minimize complexity as much as possible. But how do we do this?

There several simple guidelines which may be followed. One is to maximize independence. One practical application of this concept involves illustrating just the inputs and outputs to a given function without concern for where the inputs come from or where the results are used. The technique is often called the "black box" approach to system design. A good example of a "black box" is a square root routine. If given a positive number, the routine will always produce a square root. It doesn't care where the orginal number came from or where the result will be used. It has a well-delineated job to perform which is simple to understand because of its high degree of independence. This concept can be applied to any function in a system.

Hierarchies in a system design are integral parts of most modern techniques. This, too, tends to eliminate complexity by leading the user from overall summary levels through successively finer levels of detail. Here, again, we want to show only how the current level is structured at the next level down. Illustrating three or four levels at a time is too complex.

Another technique is commonly found in database applications. If a record in a database contains 30 or 40 items, but only 2 are required to understand the current function, mention only those two. Give the function a "logical view" of the record such that it appears to contain only those items which it needs to function properly. The unnecessary levels of detail should remain "hidden" to avoid complexity.

We have seen that the translation of an object from one person to the next is a major source of errors. This is also true at the interface between two functions. When one routine passes data to another, a translation process occurs and this is, again, a major source of errors. The coupling between two modules should be described in detail and it should be kept as simple as possible. Consistency, here, is the key. The mechanism of the interface should always be the same. And if the format of the data is consistent, the less likely that errors will be introduced.

Keep in mind that any interface is much more difficult to grasp than the actual internal functioning of a module. Beginning programmers, regardless of the language, always have trouble understanding the CALL statement. They can generally grasp arithmetic statements and even compound conditional statements with ease relative to understanding how data is transfered from one module to another.

OTHER ASPECTS OF ERRORS

Historically, we find that analysis and design phase errors take longer to correct that coding errors. They are not only the most common errors, but also tend to be the most severe. If left undetected during the initial phases, they are also, by far, the most expensive.

They also tend to be missed by the developer. Our testing efforts are often concentrated on finding coding errors within a module rather than uncovering the design errors in the interfaces. That level of testing is too often left up to the user. An error left undetected until the validation phase can cost 5 times what it would have cost to fix it during the design phase [4]. If an error is discovered following installation, the difference is often an order of magnitude greater.

It's imperative that we facilitate the testing of the design as well as the coding and integration of a system. Designs should be kept on easily correctable media such as pencil designs on paper. If a system design chart is cast in concrete, or put on a blueprint, it becomes difficult and expensive to accomodate changes and corrections.

Here are some more interesting things to consider. A programmer's chance of fixing a bug has been found to be 50-50 if 5 to 10 statements are modified in a program. If 40 or 50 statements need modification, the odds drop to an incredibly low 1 in 5 [1]. In practical terms, this means if major alterations are required to a module, your chances of success will be much greater if the entire module is thrown out and rewritten. Instead of playing nursemaid to an ailing program, take out your scalpel and surgically remove and replace any major problems.

Also, if more than one module is affected, there is a translation process occurring and the odds of fixing the problem drop dramatically. This is a simple rule of probability. If any 4 interconnected events each have a 90% chance of working perfectly, the system as a whole only has a 65% chance of working perfectly. In addition, if ten such events are involved, the odds drop dramatically to a slim 35% chance. These results are obtained simply by multiplying the probabilities together. Here again, we can minimize this effect by maximizing the independence of one module upon another. And this can be done simply by having each module double check its inputs and prevent any errors from propogating throughout the system.

Regardless of how carefully we design systems, errors are still likely to occur. Given that fact, let's look at some approaches to handling these errors.

FOUR APPROACHES TO RELIABILITY

There are four basic approaches to software reliability: fault avoidance, fault detection, fault correction, and fault tolerance.

The obvious best choice is fault avoidance. If we have procedures and methodologies that tend to prevent errors before they occur, we are obviously much better off. Again, we want to manage and minimize the complexity of a system to avoid the root cause of translation errors.

We also should emphasize the completeness and accuracy of any translation process. This can effectively be accomplished through various testing procedures. Through the use of design reviews and walkthroughs involving people from both ends of the translation process, we improve communication which leads directly to the reduction of errors. Analysts should be actively involved in the design and designers should be actively involved in the programming activities of a system. Testing personel should be involved at each step. To reduce the overall cost of development, we should be testing the analysis, testing the design, testing the code and even testing the documentation. Any methodology which facilitates this will result in an overall lower cost of ownership.

The second best method is fault detection. In the movie 2001, there is a scene where HAL, the computer, detected an error in an AE-35 communication unit. His error message not only included the source of the error but also predicted its moment of manifestation.

In more down-to-earth applications, like ours, we can minimize the effect of an error by detecting it as early as possible, reporting it in a standard fashion and isolating it from the rest of the system.

Fault detection methods should be standardized. All incoming data to a module should be passed through a validation "sieve", to guard that module against errors created in other parts of the system. The actual format of the error message should be considered. It should indicate the module involved, the data received, the file involved and a process history if possible. Quite often, the quality of an error message will have a great impact on the time required to identify the cause of an error. And keep in mind that these considerations should be an integral part of the design! Don't leave it up to the programmer.

Fault correction is the next approach to reliability. This involves repairing any damage that may have been caused by a failure. Generally, this method is only cost-effective if it can be generalized to handle a wide variety of errors.

Examples of this method include use of checkpoint/recovery and prelogging techniques that are common to most operating systems. Application systems can also make use of this method through the use of detail redundancy. This simply involves a mechanism whereby summary figures are recalculated from the detail information.

Of course, its often better to design the system in such a way that the need for fault correction is minimized.

Our last approach is fault tolerance. Here there are three basic methods: Dynamic redundancy, fallback, and error isolation.

Dynamic redundancy is basicaly a "voting" technique which is sometimes used in complex hardware-oriented systems such as the space shuttle. It is generally not practical to implement in application software.

Fallback involves the return to a previous release which was known to work. This is often done at a customer site manually, and is, again not practical to automate.

Error isolation, however, is very practical. It is easy and cheap to isolate errors to a minimal part of a system once they are detected. Modules should be designed to either work or fail in some graceful fashion no matter what the input. The concept of garbage-in-garbage-out, is not a very cost-effective way of handling errors. Rather, if you detect some garbage, take it to the dump. And try to stress that the redundant validation of module inputs is cheap. Do not rely on some other module to provide the validation for your own input. If your module will only work with positive numbers, then fail gracefully if it just happens to be negative. If you find yourself thinking, "Well that just can't happen", then insert the code necessary to ensure that it won't.

CONCLUSION

Many books have been written discussing various software engineering techniques and how effective they are at reducing overall life cycle costs. What I'd like to leave you with is an understanding of the cause of software errors and some background that will enable you to choose a methodology that attacks these causes. Whatever technique you choose, keep in mind these four things: It should be aimed at reducing maintenance costs, since this is the major expenditure in the life cycle of any piece of software; it should emphasize ease of modification, with a minimum of complexity; it should facilitate better designs as it relates to the user's expectations of the system; and it should provide for testing at each step along the way.

REFERENCES

1. Edward Yourdon/Larry L. Constantine, Structured Design, Prentice-Hall, 1979.

2. Barry W. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

3. Glenford J. Meyers, Software Reliability, John Wiley & Sons, 1976.

4. SADT Structured Analysis and Design Technique, SofTech, 1976.

PRODUCTIVITY ORIENTED SOFTWARE

Software and Systems Engineering

Fred Hoette
Productive Computer Systems, Inc.
Middle Haddam, Conn.

PRODUCTIVITY ORIENTED SOFTWARE

Assume that I am a user requiring an application to be developed on a computer. I go to my local software house where I am told that the hardware will cost me $20,000 and can be operational in two months. The software will take three times as long to develop and cost twice as much. The hardware will come with a one year warrantee and the manufacturer guarantees a four hour response time on service. Given the mean time between failure of computer hardware, I am not likely to require service very often. On the other hand, the contract for the software hardly reflects the software house's confidence in the product, boasting such inspiring phrases as ".. no warrantee..", "..accepts no liability..", "..as is..", etc. As a customer I am clearly impressed.

What impresses me is what a remarkable job computer manufacturers have done in lowering the cost of their product while increasing reliability and what a remarkably ineffective job we in the software development sector have done in either improving cost, quality or productivity of software systems.

The raison d'etre of software development staff, be it at a software house or at a user site, is to support somebody's business process. Our focus, terefore, should be on developing solution oriented system products. The only way that we can produce better software is to use a better methodology and to use better tools to implement that methodology.

It currently costs an amazing $ 8 per line of code put into production. Our response to this has been to develop "better" languages, the current vogue being PASCAL. Ten years ago the answer was PL/1. I do not believe that language syntax or implementation addresses the problem, which is improper methodology.

One of the first books on systems development methodology was written by Charles Dodgson, a vicar in Oxford, England (he may not have fully aware of his books global application at that time). In 1865, under the penname Lewis Carroll, he published the book "Alice In Wonderland". Bear with Alice as she discusses systems development with the Chesire Cat. "Would you tell me, please, which way I ought to go from here ?". "That depends a good deal on where you want to get to" the Cheshire Cat replied, "I don't much care.." said the Alice. "Then it doesn't matter which way you go" said the philosopher Cat, "..so long as I get **somewhere**..". "Oh, you're sure to do that if you only walk long enough" replied the Cat.

Traditional system development relies on **the long walk**. It makes the assumption that the user knows where he's going, that he will recognize his destination when he gets there and that, at least in part, he will even know **how** to get there. In practice none of these is usually completely true.

The result is a so called phased systems development approach whereby one or several users are expected to sign off on systems specifications which often compete with New York City Yellow pages for sheer bulk. To encourage speedy sign-off, no further development allegedly takes place until the specifications have been approved. If the specs are based on some fundamental error in communication, which the user signing off may not notice, since he's really too busy too read **all** the specifications, then the resulting system will be fundamentally incorrect. The maintenance impact of a basic error in design is usually dramatic.

The basic problems with a one pass implementation are:

1. A user cannot place a complex system in context or consider its full ramifications. This problem compounds itself exponentially as the number of different users increases.

2. Typically none or few deliverables reach the user until the time when when the cost and impact of change is at its maximum, systems testing at the earliest.

To this entire process we apply terms like "computer science", "software engineering", "information systems", "software technology". There is, in fact, no theory of computing algorithms or of programming: systems development is a **craft**. Witness an error rate of between 1 and 2 errors per 100 lines of code put into production. With a development cost of $ 8 per line of code, the cost per error line is many, many times higher. OS 360, IBMs first flagship operating system, which for that reason is perhaps a uniquely unfair example to use, had over **100,000 different** errors in its first seven years of operation. While hardly in the same class, contemporary operating system patch file listings frequently run thousands of lines some of which, incidentally, often make fascinating reading.

So let's leave Wonderland and examine current productivity oriented offerings:

- Special purpose languages. GPSS is an example of a simulation language.

- Problem specification languages. Languages which set up or interpret decision tables, for example.

- Customerizers. This is a general term to describe application packages which have all options predefined. The user generates his own version by describing his environment.

- Program generators. There are quite a few of these products on the market. They generally offer a shorthand way of generating a high-level program. The program is then used in the same manner as any other program. Unfortunately, the output from a program generator looks precisely like what would be expected from a machine. Furthermore, program generators usually compound the maintenance problem which comprises a major cost over the life of the system.

All the previously mentioned productivity aids emphasize only the "how to get there faster" aspect of systems development. But if you don't know where you're going, it doesn't matter how fast you travel.

The final entry in the list is a facility which I will call a systems modelling or prototyping application development facility. I believe that this type of product represents the way of the future in systems development.

A modelling system is a facility to quickly, easily and relatively painlessly develop a basic system which may then be iteratively built upon to produce the final system. This approach answers not only the question "how will I get there ?", but also "where am I going ?". The technique assumes that the most effective way of designing and building a system is to involve the user in the complete specification process by letting him or helping him to build the prototype and constantly refining that prototype until the final product has been generated by iteratively asking the question "is this what you mean ?".

Such a deveopment system should have the following characteristics:

- It should be easy to use. The user of such a facility should be able to address any entity in the system using meaningful terms. All functions should be menu or help driven, but an experienced user should be able to bypass any menu of help text to get directly to the desired function.

The user should not need to define anything to the system more than once and therefore should not need to tell the system anything which it should be able to derive.

- It should include a screen generator whose output, the screen definition, should be should be completely independent of the procedural code.

- It should include a report generation facility.

- Data integrity and validation definition should be an off-line non-procedural process, data dictionary driven.

- The procedural language should be very high level and should include extensive, inherent consistency checks.

- Each system component should generate its own documentation, so that the prototype and eventually the full system is as self-documented as possible.

- Lastly, and perhaps most importantly, the I/O and data attributes should be dynamically alterable without the need to make changes to or recompile the procedural code. Screens, report formats and, to a large degree, data attributes should be dynamically modifyable by simply making the appropriate change in the off-line maintenance utility or data dictionary entry.

Therefore the products of a modelling system, in other words the application system, will have the following qualities:

- Correctness. There should simply be fewer opportunities to make mistakes.

- Responsiveness. Call it user friendliness. The system should communicate with English instructions, error messages, etc.

- Adaptability. The system should be easily extendable in function. Since the iterative development process is really a form of maintenance, the resulting system will be easily changed.

- Self documented.

The advantages of using this approach are :

- Communication pitfals are avoided. The user gets what he wants the first time.

- User resistance to the system will be minimal since he effectively designed it.

- It will not be necessary to publish version 2 specifications before version 1 goes live. This is a fairly common characteristic of contemporary systems.

- Overall development cost is lower. More resources are consumed in the design pahse and far fewer in programming and testing.

- The maintenance costs are much lower since the system will require less maintenance and that which is needed will be much easier to apply.

These remarks apply not only to custom systems development, but to tailoring of canned packages as well. Most customers resist adapting their environment to suit the limitations, however minor, of a software package. This is especially true where the business function to which the software applies has been well developed. Most software developers, on the other hand, resist modifying their packages to suit a particular customer.

A package which has been built using a prototyping development facility affords both standardization and flexibility. If you develop software, consider the impact on your sales or on your effectiveness if you were able to offer each customer or user a tailored off-the-shelf package without having to make any programming changes or recompilatons.

Such packages are available now on a number of computer systems including Texas Instruments. I believe that these products, after some initial resistance from the traditionalists in d.p., will become the common development approach. I look forward to the day when the last print program is relegated to a well deserved retirement and to the time when we will never again have to worry about whether a programmer decided to practice his art form to its fullest level of creativity and used some obscure date encoding scheme to save a byte of disk space.

A Methodology for testing the completeness and functional correctness of a unit of software.

Software and Systems Engineering Session

Gene Korienek

Johnson Controls
Milwaukee Wisconsin

ABSTRACT

At some time in the development cycle of a unit of software it becomes necessary to determine that the software is complete in its prescribed functions and that these functions execute correctly as specified. This paper describes a structured testing methodology that is designed to evaluate software completeness and functional correctness.

There are four basic processes in this methodology: 1) the identification of the functions of the software, 2) principles for the selection of test cases, 3) a format for the structured execution of the test cases, and 4) a validation process by which the effectness of the testing methodology may be evaluated.

Each process within the methodology is structured and explicitly described in both text and example. Data collected from a case study application of the methodology is also discussed.

With software gaining an ever more responsible role in our society the issue of software quality has generated considerable interest. In the past five years software testing has become a viable and very important discipline within the computing sciences. An area that has been given particular attention is the area of structured testing. Much of the published work in this area has been theoretical or academic in nature and has tended not to present many case studies in which implementation level details can be discussed. The current paper is an account of a project undertaken to develop a structured software testing methodology designed to govern the following aspects of the software testing process:

1) The selection of test cases
2) The writing of the test plans
3) The execution of the test plans
4) The evaluation of the results

A black box approach to the testing was used and test cases were derived directly from the documentation. The general goal of this particular testing methodology was to verify that the software functioned as specified. The remainder of this paper will proceed to discuss the implementation of this methodology.

SELECTION OF TEST CASES

The testing principles discussed in this paper currently guide the construction of a set of test plans. These test plans are designed to evaluate the functional characteristics of the system.

The actual test case structure of these test plans is determined by the following principles of test case selection:

1) Equivalence Class Partitioning
2) Boundary Value Analysis
3) Error Guessing

Equivalence Class Partitioning

Equivalence class partitioning is a technique that systematically reduces a very large number of possible inputs into a manageable, but representative set of inputs. By using this technique an input domain can be partitioned into a finite number of equivalence classes such that a test of any given value of a class is equivalent to a test of any other value of that class.

From a testing point of view this means that if one test case in an equivalence class detects an error then all other cases in the same class would have detected the same error. In an extension of this logic, if one test in an equivalence class detects no error then all other cases in the same class would detect no error.

Guidelines for test case selection by equivalence class partitioning have been discussed in Meyers (1976,1979). The guidelines that were used in this project are:

A. Equivalence classes can be identified by taking each input condition and defining its set of valid input values. Select one of these values to be the representative value for that set of inputs.

Repeat this process for any invalid values that may be associated with the input being analyzed. The purpose of this procedure is to develop a minimal set of meaningful test cases. For example, if an input condition has a valid range of $1 < \text{input} < 999$, the set of equivalent classes for this input would consist of:

Valid Data	$1 < \text{test data} < 999$
Invalid Data	$1 \geq \text{test data}$
Invalid data	$999 \leq \text{test data}$

B. If an input condition specifies a numerical range of values (e.g., 1 through 28 are valid day values for the month of February) identify one valid equivalence class input (14) and two invalid equivalence class inputs (0,29). In this case the input, 29, happens to be a member of an invalid class and a valid class (leap year).

C. If an input condition specifies a set of values and there is reason to believe that each is handled differently by the software (e.g., peripherals) identify one valid equivalence class for each and one invalid equivalence class (e.g., peripheral: xyz).

D. If an input condition specifies a "must be" situation (e.g., valid entry must be any non-blank character) identify one equivalence class (i.e. any non-blank character) and one invalid equvalence class (i.e. a blank).

E. If there is any reason to believe that elements in an equivalent class are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.

There may exist situations in which equivalent classes identified in the documentation resources (e.g. Feature Descriptions, Functional Specificatons) are not implemented as eqvalent classes in the software. Situations such as these **require** that the documentation level equivalent class be subdivided into the appropriate implementaiton level equivalent classes.

By following these guidelines, the set of all possible inputs to a program may be reduced to a set of all possible equivalence classes of inputs. This results in a considerable reduction of tests cases.During the reality of testing, input values are selected from these equivalence classes (one from each). A further technique, boundary value analysis, facilitates the efficient selection of input values from the identified equivalence classes.

Boundary Value Analysis

The literature indicates that software boundaries are particularly fruitful areas when seeking errors (Meyers, 1979; Howden, 1981). By including a sub-set of test cases containing boundary tests in the set of test cases derived from each equivalence class, the test plan can take advantage of, and test for, high error probability boundary conditions. The following list of test cases extends the previous example to include boundary condition test cases.

Valid Data	1 < Test Data < 999
Valid Data	2 = Test Data
Valid Data	Test Data = 998
Invalid Data	1 = Test Data
Invalid Data	Test Data = 999

Error Guessing

The previously mentioned techniques form the structured portion of the test plan. The strengths of methodologies like equivalence class partitioning, and boundary value analysis lie in thier ability to structure a reasonably comprehensive test of input conditions. There are, however, test cases that are not amenable to selection by these methodologies. In the current project, these types of test cases have been identified through a technique we call "error guessing". Error guessing is an unstructured testing technique testing technique used by intelligent, creative, and experienced testing personnel when flying by the seat of thier pants. Fortunately it works and will be used to provide additional test cases.

DEVELOPMENT OF THE TEST PLAN

With the test case selection methodologies chosen it became necessary to decribe a process by which the documentation could be translated into test cases and an eventual test plan. Listed below are the steps that were defined in the test case selection process. It is very much a generic process and should apply to most all software documentation.

1. **Read the documentation** that describes the functional characteristics of the software being tested. A productive way of using the documentation is to highlight all assertions, inputs, actions, limits, and output statements.

2. **List all highlighted statements in input and expected outcome form.** If the statement is an input, list it along with the expected outcome of its execution. If it is an action, then list the input and output associated with the action. If the statement is an output, list it along with the input that caused it to happen. At this point a basic test plan outline has been constructed.

3. **Combine any inputs that are members of the same equivalence class.** This step should reduce the size of the outline and subsequently reduce the number of test cases.

4. **Translate each line of the outline** into one or more test cases. The inputs should be in the command syntax of the system wheneverpossible, and the expected outcomes in English words. Add all test cases necessary to conduct a complete boundary value analysis on each of the inputs listed in the outline. Then add any additional test cases that are felt necessary to test the sortware but do not fall within the auspices of the structured test case selection methodologies.

Prior to the to the completion of the project, a syntax standard was developed that facilitated the readibility, executability, and overall documentation capabilities of the test plans. The standard also described the test case selection methodologies and a procedure for the execution of the test plan.

Execution of the test plans constructed and written to this standard is a relatively quick and routine operation that can easily be accomplished by almost anyone with a minimal understanding of the system. In addition, it became apparent that when a completed test plan was at hand during testing there tended to be more testing done. This was most probably due to the guidance provided by the test plan.

EVALUATION OF THE METHODOLOGY

The primary motivation behind the development of any software testing methodology is to enhance the quality of the software. The manner in which "software quality" is measured varies from project to project. Within the scope of this testing project it was decided the evaluation of the error detecting effectiveness of the test plans would provide an indirect measure of the quality of the software. The evaluation of the test plans consisted of the following tests:

Reliability

Reliability in the context of a test plan refers to its ability to reproduce the same results over many executions. It is important that potential confounding variables(e.g. non-significant hardware, tester personality / ability, etc.) have no effect on the results of the test. The test plans must be structured such that their results reflect the status of the software and nothing else. In the current project reliability was evaluated by using three people to simultaneously execute several of the test plans. A comparison of the results of these executions was then conducted. As it turned out, the level of detail and procedural rigor inherent in the test plans assured a high level of reliability. Of the 150 test cases executed in the test plans involved in the reliability test, 147 test cases revealed identical results across executions. The response generated by the software in the remaining 3 test cases was ambiguously defined in the specification and, consequently, differently interpreted by the various testing personnel.

Similiar tests were also conducted with a variety of hardware configurations. Results indicated that the test plans reliably revealed the same errors independent of hardware configuration. The hardware configurations tests were specified to be functionally equivalent.

Validity

Evaluating the validity of this methodology has been difficult. Because of the number of variables encountered during the software development cycle the impact on software quality that can be attributed to the test plans cannot be isolated. Long range trends in error number and type represent the only data available for analysis at the time. The results to date do not reveal any differences in error rates that could be attributed only to the introduction of this testing methodology.

Error seeding experiments are being planned in an effort to explicity evaluate the validity of the testing in an isolated environment.

Completeness

Completeness refers to the ability of the test plan to test all functions of the software as documentated in the software specificaiton. In this testing project the writing of the test plans was almost entirely driven from statements in the documentation. As a result there is a very tight relationship between what functions the software was specified to do and what functions were tested. This type of relationship implies a high level of completeness in the testing.

A somewhat more explicit check for completeness was conducted by determining how many command words were specified for use in a particular unit of software and how many of these command words were actually tested in the test plan. The results indicate that all command words specified were tested with an average of 25 test cases written to test each command word.

In addition to the test plan effectiveness and completeness evaluations, it was also of interest to determine the amount of time taken to write and execute the test plans. Initial data indicate that time utilization was as follows:

Process	% of total time spent on testplan
Outlining documentation	25
Writing testplan	69
Executing the testplan	5
Reporting the errors	<1

In summary, this methodology has been implemented on a large software system. Preliminary reliability and completeness evaluations have been made and indicate that the methodology is reliably testing the software in all of its prescribed functions. The effectiveness of the methodology in revealing errors has not yet been explicity evaluated.

Casual observations of the process reveal that the writing of the test plans is a very time consuming process but that it contributes to the functional definition and correctness of the software. Enhancements to the testing methodology are in the design phase and involve techniques for increasing the effectiveness of the test plan, automating various aspects of the test plan process, and further attempts at evaluating the completeness and validity of the test plans.

THE UNSTRUCTURED APPROACH TO SOFTWARE TESTING

Software and Systems Engineering Session

Ronald W. Reinert

Johnson Controls, Inc.

Milwaukee, Wisconsin

Abstract

Unstructured software testing (testing without a written plan prepared in advance) is discussed and contrasted with Structured software testing, which utilizes a formal test plan. Definitions and a few words on the content of the paper are followed by a very brief history of software testing approaches and a description of the software framework assumed in the discussion. We then consider some advantages and disadvantages of Unstructured testing and whether the Structured and Unstructured approaches tend to find different types of software errors. A discussion of the extent to which Unstructured techniques may yield to analytical definition follows. Some ideas are then presented which may help us to avoid duplication of effort when both the Structured and Unstructured methodologies are applied to the same piece of software. The paper ends with a brief summary.

Definitions

This paper discusses an approach to software testing which may be called Unstructured (or Seat-of-the-Pants) testing. Unstructured testing is defined here simply as a method of software appraisal which does not use a formal test plan. The software is tested against a specification of its intended functions without deciding in advance which test cases are to be performed. The test cases are created by the test technician as the test progresses, by studying the specification and by using his or her experience and instincts to suggest specific test cases and test data.

Unstructured testing can be contrasted to Structured testing methodologies, in which test cases are defined and committed to paper in a formal test plan prior to test execution. A Structured test then consists of performing only the tests described in the test plan and recording the results. A person performing the Structured test may "think of something he'd like to try" as the test is being carried out. If he does, he is venturing into the area of Unstructured testing.

Content

The purpose of this paper is to provide a starting point for determining the proper place of Unstructured testing in the total software testing process. We should not expect to find that either the Structured or the Unstructured approach is at this time sufficient to stand on its own as a truly thorough means of assuring a quality end-product. Rather, we will compare the relative merits of the two methodologies, and speculate on how they can best work together to achieve the desired results. We will ask ourselves questions and provide answers - sofe firm, some speculative. The intent is not to give the last word on the subject, but to provide a mental springboard which will encourage further investigation.

History of Testing Approaches

In the early days of software testing, programs were written and tested by the programmer or engineer who developed them. No test plan was perpared, and the programs were tested against a specification (if one existed) or against the programmer's knowledge of what the software was "supposed to do."

Later on, there came a trend toward putting the software appraisal function into the hands of a separate group, which could more objectively evaluate the program quality. For the most part, testing was still done without a written test plan. To this point, the Unstructured technique still dominated.

More recently, the trend has been to develop Structured testing methodologies which allow test cases and test data to be derived in advance, and committed to paper to create a formal test plan which rigidly defines the steps to be followed in executing the test. The advantages of the structured approach have been widely discussed.

What remains to be seen is whether Structured techniques can ever complete displace Unstructured ones in a thorough software quality approach.

Framework of the Discussion

A brief description of the software structure to be discussed is necessary. We will be talking about a software "system" which is composed of "features". A "feature" is a set of program modules which work together to accomplish a single function. In a building automation system, for example, one feature may control the time programmed operation of heating and cooling units, lights, door locks, etc. Another feature may calculate the optimum amount of outside air necessary to cool an area in the most energy-efficient manner, while yet another is responsible for the output of hard and soft-copy logs and summaries. A host operating system to schedule and run the features, along with the collection of features themselves, are put together to form the software "system" under discussion.

Some features may operate independently of others, while some may interact with each other in ways which are not always apparent in advance. As an example, two features which each have the ability to control the same heating unit, but for different purposes, may require a prioritization scheme to avoid the problem of fighting for control of the unit. All of the features, whether independent or not, may have an effect on total system performance and throughput.

In addition, we are talking about a fairly large system - one composed of many features working wogether in such a way that all of their possible interactions are nearly impossible to predict. The testing of the total system is made manageable by testing the operation of the features individually, followed by a system integration test to evaluate feature interactions. For the individual tests, a test plan may be prepared for each feature - if the Structured approach is being used.

It is further assumed that the features have already been debugged and subjected to an ititial Unstructured test by the people who wrote them. In other words, the developers have declared the features to be "working", and are submitting them to a separate test group for final qualification.

Advantages of Unstructured Testing

The motivation for employing Unstructured testing lies in the fact that it does have some advantages over Structured testing, as seen from experience. There are also disadvantages, to be discussed further on. What are some of the advantages?

The main point in favor of the Unstructured approach is that it does find problems in software, even after Structured tests have already been applied. This does not imply any difference in the expertise of the person who writes and executes the Structured test plan and the person who executes the Unstructured test. It results simply from the fact that the two methodologies do not use the same set of test cases.

Another advantage is that Unstructured testing does not require time to be spent preparing a test plan. This can result in significant savings in time and money.

Unstructured techniques usually extend beyond the confines of the software specification. The tester is allowed the flexibility of using his creativity and imagination in constructing test cases as the test progresses. This free-wheeling approach results in the investigation of operational aspects that might otherwise be overlooked. As a result, the tester is more likely to uncover feature interaction problems, specification deficiencies, throughput problems, and cases in which the software is responsible for various other unexpected effects.

Finally, Unstructured testing appears to be the method which makes the greatest use of the tester's experience, intuition and creativity - all valuable resources which should be fully exploited.

Disadvantages of Unstructured Testing

The move toward Structured testing methodologies has come about because of the fact that Unstructured techniques also have their disadvantages. We can consider some of these here.

The most apparent deficiency of Unstructured testing is its lack of repeatability. This is not to say that a particular test case can not usually be repeated from memory in the short term for the purpose of reproducing a problem symptom, but rather that the entire set of test cases can not be reproduced later on. Since there is no written plan, the same person testing the same software six months later will have to reinvent all of the test cases. With the Structured approach, the creative process of preparing the test plan is required only once. A different person can easily perform the identical test at any time in the future simply by following the steps of the plan.

It follows from the above that repetitions of Structured tests on the same software (or even initial tests from new test plans) can be executed by less experienced personnel, whereas an Unstructured test, to be productive, must be performed by someone with ample experience and insight into the software testing process. This is another disadvantage of Unstructured methods.

An additional objection to an Unstructured test is that it is the product of a single mind (assuming that one person is performing the test). When the first draft of a Structured test plan has been written, it can be walked through by a group of people familiar with the software. This additional input adds to the effectiveness of the Structured plan. Unstructured testing generally employs only those test cases created by the person who is actually executing the test.

Types of Software Problems Discovered

Since the Structured and Unstructured approaches employ different techniques and use different sets of test cases (with some overlap), we may wonder whether they tend to uncover different types of software problems. Having frequently seen both methods used to test the same software, I can say that this does appear to be true.

One simple (though subjective) way of categorizing software errors is to place them along a continuous scale which ranges from "conspicuous" to "subtle". Errors which would lie toward the "conspicuous" end of the scale would be, for example, a clearly specified function which simply does not work. Toward the "subtle" end, we may have such errors as feature interaction problems, specification oversights, intermittent timing problems, etc.

It is certainly true that either testing approach is capable of finding problems anywhere along the scale. The Structured method, however, does seem to find a high percentage of "conspicuous" problems. This is not surprising, since a large amount of effort and the ideas of more than one person go into preparing the test plan from a thorough study of the specification. Errors in clearly specified aspects of operation are very likely to be caught. On the other hand, an Unstructured test usually seems to reveal a higher percentage of "subtle" errors. Again, this may be expected due to the free-handed nature of Unstructured testing.

A study is now under way at Johnson Controls which may shed light on whether the above suppositions are true or not. A number of software features are being tested first by Structured methods and then again by Unstructured ones. Error categories more suitable than "conspicuous" and "subtle" will be defined, and the problems found by the different methodologies will be categorized.

Analysis of Unstructured Techniques

Since Structured and Unstructured techniques each have distinct advantages, it is tempting to ask whether we could combine these advantages into a single testing methodology. The first approach that comes to mind is to try to analytically define Unstructured methods. If they can be defined and written down, they can become repeatable and reusable. In other words, the Unstructured becomes the Structured. Unfortunately, however, the analysis of techniques which rely heavily on insight and creativity is no easy task, as can be seen by considering some of the things we do know about Unstructured testing.

Many Unstructured test cases are suggested by occurrences which happen during the test. The person performing the test sees something happen which makes him wonder how another feature will react under similar circumstances, or it may make him wonder how the observed operation will interact with another feature. Since the test case suggested by the observed occurrence requires the observation in the first place, it is difficult to see how the necessity for the second test could have been predicted before the test was begun.

Some of the problems found during Unstructured testing are specification problems. For example, the tester may feel that an observed aspect of operation which is not well defined in the specification may be confusing to the end user. Since a Structured test plan tends to follow the specification closely, this type of question seems to arise more frequently in Unstructured testing. Again, the feeling that a problem may exist is triggered by an observation during the test - and the "feeling" that something may be wrong seems to be more of an intuitive process than a logical one.

In some cases, we may even have to question the value of adding structure to an Unstructured test. It often happens that problems found during Unstructured testing rely on highly specific test conditions. Once the specific problem is corrected, the likelihood of that particular test case ever finding another problem becomes very low. Adding the test, and many others like it, to a formal test plan may well be counter-productive.

One approach to integrating Structured and Unstructured techniques is to make "error guessing" a part of the Structured test plan preparation process. The person writing the plan first prepares the list of basic tests suggested by the specification, and then spends some time "guessing" at other test cases which may be useful but are not obvious. If one seems promising, it is added to the plan. The process of guessing is an Unstructured activity, but making the test case a part of the test plan makes it repeatable.

For now, though, it does not appear that we are prepared to make very much progress in adding structure to Unstructured testing techniques. This is not to say that further work may not enable us to do so, but the use of words such as "wondering", "feeling" and "guessing" in the above discussion indicates that we are dealing with the mental processes of intuition and creativity, which are not nearly as well understood as logical thought processes, and therefore defy analysis.

Duplication of Effort

If we are to take advantage of the strong points of both the Structured and Unstructured techniques, we may have to test each feature twice. If the quality of the final software product is the only consideration, there is no objection to this dual activity. It does, however, lead to duplication of effort which may be costly. This is because the "obvious" tests are likely to be a part of the written test plan and are just as likely to be repeated by the person performing the Unstructured test. A cost/benefit analysis could be attempted, but the cost of releasing software with undetected problems is very difficult to evaluate. What can be done to minimize duplication of effort?

One approach is to perform the Structured test first, and to then forward the test plan with its results to another person who will perform the Unstructured test. He can then review the test plan to determine those aspects of the feature which have already been found to work properly, and concentrate his attention on other tests as suggested by his experience and intuition - tests which are not a part of the test plan.

Another idea is to combine the Structured and Unstructured techniques into a single test. A technician with sufficient experience to perform fruitful Unstructured testing can be provided with the unexecuted Structured test plan. He can then perform the Structured test followed by a period of Unstructured testing, or intersperse Unstructured test cases with the Structured test as it is being executed.

Summary

Both Structured and Unstructured methodologies offer advantages which must be exploited if the total testing process is to result in a software product of the highest possible quality. Since they appear to find different types of problems due to the use of different sets of test cases, neither approach can by itself be expected to find as many software errors as will be found by employing both methods. To minimize the cost of duplicated effort, we need to find ways of integrating the two techniques into a testing approach which uses each test case only once, and which is repeatable. One way of doing this is by attempting to add structure to Unstructured techniques. The heavy reliance of Unstructured testing on experience, creativity, intuition and insight makes this a difficult task, but it appears to be a task which is worthy of further investigation.

TEXAS INSTRUMENTS COMPUTER USERS GROUP

TI-MIX