Testing Computer Software
Second Edition

APPENDIX: COMMON SOFTWARE ERRORS

THE REASON FOR THIS APPENDIX

This Appendix describes over 400 bugs. The descriptions are short, including only what we considered the most interesting information. It's worthwhile reading this list, even though you may find it boring. On first reading it provides a frame of referenceódetails, and background about problems you should look for while testing. Its greater value is as an organized list of program problems for future reference. A good list, built up with time and experience, can be a powerful tool.

A set of test materials is nothing more than a list of possible problems in a program and a set of procedures for determining whether the problems are actually there or not.

Generating reference lists of bugs on the fly is risky and inefficient. It takes too long to dredge up old insights, and you're too likely to get stuck on one or a few themes. We prefer to check a comprehensive "standard" list to get further ideas for testing a program. This Appendix is an example of a standard list.

Whenever you test a new program, you'll think of new entries for this list. Add them to it. We recommend that you enter this list onto your computer and manage it with an outline processor.

Here are examples of ways you should use this list:

1. Evaluate test materials developed for you by someone else. When you contract out the development of test materials you're likely to get back many thick binders of test cases. They look impressive, but they're incomplete. The author has blind spots, which you have to find. For example, in stacks of test materials that we've seen developed by contract testing agencies, we've yet to see an explicit race condition test. Unfortunately, it's easier to see what an author had in mind than to step back and see what's missing.

We recommend evaluating the coverage of test materials against a checklist. Include a few examples of each class of error that you can imagine existing in the program. It doesn't matter whether the program has these problems ó it only matters that they could be there. Once the list is complete, check the test materials to see which problems they would expose. If these tests wouldn't expose some errors in your list, look for tests for similar problems. This can quickly expose large classes of missing tests.

It's easy to make a checklist from a large error list like the one in this Appendix. Just look for classes of errors that are relevant to your testing project and find a few good examples of each. Add any other problems that you think of that seem important for the program under test.

2. Developing your own tests.

Write your own list of errors that you think the program could have. Then look here for more ideas. Consider each problem in this Appendix: if you're sure it's irrelevant to the program, forget it. Otherwise write a test case (or a few) to see if the program has the problem. When in doubt, ask the programmer whether a given type of error is possible in the program and how to test for it.

We also recommend this list for testing "on the fly." If you don't have time to plan and document a set of tests for a program, test from a list of candidate problems.

3. Irreproducible bugs.

A bug is hard to replicate because you don't know what triggered the failure, what the program did or displayed, or what could cause such a problem. You give up because you run out of hypotheses.

Use this Appendix before giving up.

Try to match reported symptoms with problems listed in this Appendix. Be creative: the matches won't be perfect because the reports are incomplete and incorrect. (If they weren't, you could reproduce the bug.) Even so, the list can help you break out of your current line of thinking. If you didn't need to refocus on other possibilities, you wouldn't be so close to giving up.

4. Unexpected Bugs.

When you discover a bug accidentally, or when one surfaces in a shipping product, look for others like it. You probably missed running a group of tests to detect the lot. This section can help you generate a list of possible related problems and tests.

OUTLINE OF COMMON SOFTWARE ERRORS

- Help text and error messages
  - Inappropriate reading level
  - Verbosity
  - Inappropriate emotional tone
  - Factual errors
  - Context errors
  - Failure to identify the source of an error
  - Hex dumps are not error messages
  - Forbidding a resource without saying why
  - Reporting non-errors
- Display bugs
  - Two cursors
  - Disappearing cursor
  - Cursor displayed in the wrong place
  - Cursor moves out of data entry area
  - Writing to the wrong screen segment
  - Failure to clear part of the screen
  - Failure to highlight part of the screen
  - Failure to clear highlighting
  - Wrong or partial string displayed
  - Messages displayed for too long or not long enough
- Display layout
  - Poor aesthetics in the screen layout
  - Menu layout errors
  - Dialog box layout errors
  - Obscured instructions
  - Misuse of flash
  - Misuse of color
  - Heavy reliance on color
  - Inconsistent with the style of the environment
  - Cannot get rid of onscreen information

COMMAND STRUCTURE AND ENTRY

- Inconsistencies
  - "Optimizations"
  - Inconsistent syntax
  - Inconsistent command entry style
  - Inconsistent abbreviations
  - Inconsistent termination rule
  - Inconsistent command options
  - Similarly named commands
  - Inconsistent capitalization
  - Inconsistent menu position
  - Inconsistent function key usage

- Inconsistent error handling rules
- Inconsistent editing rules
- Inconsistent data saving rules

Time-wasters

- Garden paths
- Choices that can't be taken
- Are you really, really sure?
- Obscurely or idiosyncratically named commands

Menus

- Excessively complex menu hierarchy
- Inadequate menu navigation options
- Too many paths to the same place
- You can't get there from here
- Related commands relegated to unrelated menus
- Unrelated commands tossed under the same menu

Command lines

- Forced distinction between uppercase and lowercase
- Reversed parameters
- Full command names not allowed
- Abbreviations not allowed
- Demands complex input on one line
- No batch input
- Can't edit commands

Inappropriate use of the keyboard

- Failure to use cursor, edit, or function keys
- Non-standard use of cursor and edit keys
- Non-standard use of function keys
- Failure to filter invalid keys
- Failure to indicate keyboard state changes
- Failure to scan for function or control keys

MISSING COMMANDS

State transitions

- Can't do nothing and leave
- Can't quit mid-program
- Can't stop mid-command
- Can't pause

Disaster prevention

- No backup facility
- No undo
- No Are you sure?
- No incremental saves

- Error handling by the user
  - No user-specifiable filters
  - Awkward error correction
  - Can't include comments
  - Can't display relationships between variables
- Miscellaneous nuisances
- Inadequate privacy or security
- Obsession with security
- Can't hide menus
- Doesn't support standard O/S features
- Doesn't allow long names

PROGRAM RIGIDITY

- User tailorability
  - Can't turn off the noise
  - Can't turn off case sensitivity
  - Can't tailor to hardware at hand
  - Can't change device initialization
  - Can't turn off automatic saves
  - Can't slow down (speed up) scrolling
  - Can't do what you did last time
  - Can't find out what you did last time
  - Failure to execute a customization command
  - Failure to save customization commands
  - Side-effects of feature changes
  - Infinite tailorability
- Who's in control
  - Unnecessary imposition of a conceptual style
  - Novice-friendly, experienced-hostile
  - Artificial intelligence and automated stupidity
  - Superfluous or redundant information required
  - Unnecessary repetition of steps
  - Unnecessary limits

PERFORMANCE

- Slow program
- Slow echoing
- How to reduce user throughput
- Poor responsiveness
- No type-ahead
- No warning that an operation will take a long time
- No progress reports
- Problems with time-outs
- Program pesters you
- Do you really want help and graphics at 300 baud?

OUTPUT

Can't output certain data
Can't redirect output
Format incompatible with a follow-up process
Must output too little or too much
Can't control output layout
Absurd printed level of precision
Can't control labeling of tables or figures
Can't control scaling of graphs

ERROR HANDLING

ERROR PREVENTION

Inadequate initial state validation
Inadequate tests of user input
Inadequate protection against corrupted data
Inadequate tests of passed parameters
Inadequate protection against operating system bugs
Inadequate version control
Inadequate protection against malicious use

ERROR DETECTION

Ignores overflow
Ignores impossible values
Ignores implausible values
Ignores error flag
Ignores hardware fault or error conditions
Data comparisons

ERROR RECOVERY

Automatic error correction
Failure to report an error
Failure to set an error flag
Where does the program go back to?
Aborting errors
Recovery from hardware problems
No escape from missing disk

BOUNDARY-RELATED ERRORS

NUMERIC BOUNDARIES

EQUALITY AS A BOUNDARY
BOUNDARIES ON NUMEROSITY
BOUNDARIES IN SPACE
BOUNDARIES IN TIME
BOUNDARIES IN LOOPS
BOUNDARIES IN MEMORY
BOUNDARIES WITHIN DATA STRUCTURES
HARDWARE-RELATED BOUNDARIES
INVISIBLE BOUNDARIES
CALCULATION ERRORS
OUTDATED CONSTANTS

CALCULATION ERRORS

IMPOSSIBLE PARENTHESES
WRONG ORDER OF OPERATORS
BAD UNDERLYING FUNCTION
OVERFLOW AND UNDERFLOW
TRUNCATION AND ROUNDOFF ERROR
CONFUSION ABOUT THE REPRESENTATION OF THE DATA
INCORRECT CONVERSION FROM ONE DATA REPRESENTATION TO ANOTHER
WRONG FORMULA
INCORRECT APPROXIMATION

INITIAL AND LATER STATES

FAILURE TO SET A DATA ITEM TO 0
FAILURE TO INITIALIZE A LOOP-CONTROL VARIABLE
FAILURE TO INITIALIZE (OR REINITIALIZE) A POINTER
FAILURE TO CLEAR A STRING
FAILURE TO INITIALIZE (OR REINITIALIZE) REGISTERS
FAILURE TO CLEAR A FLAG
DATA WERE SUPPOSED TO BE INITIALIZED ELSEWHERE
FAILURE TO REINITIALIZE
ASSUMPTION THAT DATA WERE NOT REINITIALIZED
CONFUSION BETWEEN STATIC AND DYNAMIC STORAGE
DATA MODIFICATION BY SIDE-EFFECT
INCORRECT INITIALIZATION
RELIANCE ON TOOLS THE CUSTOMER MAY NOT HAVE OR UNDERSTAND

CONTROL FLOW ERRORS

PROGRAM RUNS AMOK

- GOTO somewhere
- Come-from logic errors
- Problems in table-driven programs
- Executing data
- Jumping to a routine that isn't resident
- Re-entrance
- Variables contain embedded command names
- Wrong returning state assumed
- Exception-handling based exits
- Return to wrong place
    - Corrupted stack
    - Stack under/overflow
    - GOTO rather than RETURN from a subroutine
- Interrupts
    - Wrong interrupt vector
    - Failure to restore or update interrupt vector
    - Failure to block or unblock interrupts
    - Invalid restart after an interrupt

PROGRAM STOPS

- Dead crash
- Syntax errors reported at run-time
- Waits for impossible condition, or combination of conditions
- Wrong user or process priority

LOOPS

- Infinite loop
- Wrong starting value for the loop control variable
- Accidental change of the loop control variable
- Wrong criterion for ending the loop
- Commands that do or don't belong inside the loop
- Improper loop nesting

IF, THEN, ELSE, OR MAYBE NOT

- Wrong inequalities (e.g., > instead of >=)
- Comparison sometimes yields wrong result
- Not equal versus equal when there are three cases
- Testing floating point values for equality
- Confusing inclusive and exclusive OR
- Incorrectly negating a logical expression
- Assignment-equal instead of test-equal

Commands belong inside the THEN or ELSE clause
Commands that don't belong inside either clause
Failure to test a flag
Failure to clear a flag

MULTIPLE CASES

Missing default
Wrong default
Missing cases
Case should be subdivided
Overlapping cases
Invalid or impossible cases

ERRORS IN HANDLING OR INTERPRETING DATA

PROBLEMS WHEN PASSING DATA BETWEEN ROUTINES

Parameter list variables out of order or missing
Data type errors
Aliases and shifting interpretations of the same area of memory
Misunderstood data values
Inadequate error information
Failure to clean up data on exception-handling exit
Outdated copies of data
Related variables get out of synch
Local setting of global data
Global use of local variables
Wrong mask in bit field
Wrong value from a table

DATA BOUNDARIES

Unterminated null terminated strings
Early end of string
Read/write past end of a data structure, or an element in it

READ OUTSIDE THE LIMITS OF A MESSAGE BUFFER

Compiler padding to word boundaries
Value stack under/overflow
Trampling another process' code or data

MESSAGING PROBLEMS

Messages sent to wrong process or port
Failure to validate an incoming message
Lost or out of synch messages
Message sent to only N of N+1 processes

DATA STORAGE CORRUPTION

- Overwritten changes
- Data entry not saved
- Too much data for receiving process to handle
- Overwriting a file after an error exit or user abort

RACE CONDITIONS

RACES IN UPDATING DATA
ASSUMPTION THAT ONE EVENT OR TASK HAS FINISHED BEFORE ANOTHER BEGINS
ASSUMPTION THAT INPUT WON'T OCCUR DURING A BRIEF PROCESSING INTERVAL
ASSUMPTION THAT INTERRUPTS WON'T OCCUR DURING A BRIEF INTERVAL
RESOURCE RACES: THE RESOURCE HAS JUST BECOME UNAVAILABLE
ASSUMPTION THAT A PERSON, DEVICE, OR PROCESS WILL RESPOND QUICKLY
OPTIONS OUT OF SYNCH DURING A DISPLAY CHANGE
TASK STARTS BEFORE ITS PREREQUISITES ARE MET
MESSAGES CROSS OR DON'T ARRIVE IN THE ORDER SENT

LOAD CONDITIONS

REQUIRED RESOURCE NOT AVAILABLE
DOESN'T RETURN A RESOURCE

- Doesn't indicate that it's done with a device
- Doesn't erase old files from mass storage
- Doesn't return unused memory
- Wastes computer time

NO AVAILABLE LARGE MEMORY AREAS
INPUT BUFFER OR QUEUE NOT DEEP ENOUGH
DOESN'T CLEAR ITEMS FROM QUEUE, BUFFER, OR STACK
LOST MESSAGES
PERFORMANCE COSTS
RACE CONDITION WINDOWS EXPAND
DOESN'T ABBREVIATE UNDER LOAD
DOESN'T RECOGNIZE THAT ANOTHER PROCESS ABBREVIATES OUTPUT UNDER LOAD
LOW PRIORITY TASKS NOT PUT OFF
LOW PRIORITY TASKS NEVER DONE

HARDWARE

WRONG DEVICE
WRONG DEVICE ADDRESS
DEVICE UNAVAILABLE
DEVICE RETURNED TO WRONG TYPE OF POOL
DEVICE USE FORBIDDEN TO CALLER
SPECIFIES WRONG PRIVILEGE LEVEL FOR A DEVICE
NOISY CHANNEL
CHANNEL GOES DOWN
TIME-OUT PROBLEMS
WRONG STORAGE DEVICE
DOESN'T CHECK DIRECTORY OF CURRENT DISK
DOESN'T CLOSE A FILE
UNEXPECTED END OF FILE
DISK SECTOR BUGS AND OTHER LENGTH-DEPENDENT ERRORS
WRONG OPERATION OR INSTRUCTION CODES
MISUNDERSTOOD STATUS OR RETURN CODE
DEVICE PROTOCOL ERROR
UNDERUTILIZES DEVICE INTELLIGENCE
PAGING MECHANISM IGNORED OR MISUNDERSTOOD
IGNORES CHANNEL THROUGHPUT LIMITS
ASSUMES DEVICE IS OR ISN'T, OR SHOULD BE OR SHOULDN'T BE INITIALIZED
ASSUMES PROGRAMMABLE FUNCTION KEYS ARE PROGRAMMED CORRECTLY

SOURCE, VERSION, AND ID CONTROL

OLD BUGS MYSTERIOUSLY REAPPEAR
FAILURE TO UPDATE MULTIPLE COPIES OF DATA OR PROGRAM FILES
NO TITLE
NO VERSION ID
WRONG VERSION NUMBER ON THE TITLE SCREEN
NO COPYRIGHT MESSAGE OR A BAD ONE
ARCHIVED SOURCE DOESN'T COMPILE INTO A MATCH FOR SHIPPING CODE
MANUFACTURED DISKS DON'T WORK OR CONTAIN WRONG CODE OR DATA

TESTING ERRORS

MISSING BUGS IN THE PROGRAM

Failure to notice a problem
Misreading the screen
Failure to report a problem
Failure to execute a planned test
Failure to use the most "promising" test cases
Ignoring programmers' suggestions

FINDING "BUGS" THAT AREN'T IN THE PROGRAM

Errors in testing programs
Corrupted data file
Misinterpreted specifications or documentation

POOR REPORTING

Illegible reports
Failure to make it clear how to reproduce the problem
Failure to say that you can't reproduce a problem
Failure to check your report
Failure to report timing dependencies
Failure to simplify conditions
Concentration on trivia
Abusive language

POOR TRACKING OR FOLLOW-UP

Failure to provide summary reports
Failure to re-report serious bugs
Failure to verify fixes
Failure to check for unresolved problems just before release

USER INTERFACE ERRORS

The user interface (UI) includes all aspects of the product that involve the user. The UI designer tries to strike a balance between

- functionality
- time to learn how to use the program
- how well the user remembers how to use the program
- speed of performance
- rate of user errors

- the user's satisfaction with the program

In seeking a good balance, the designer weighs the experience and needs of the people she expects to use the program against the capabilities of the equipment and available software technology. An error in the UI results in a suboptimal match between the user and the program.

Because tradeoffs are unavoidable in UI design, a good designer might deliberately make any of many of the "errors" listed below. Don't take this list as gospel. If you are at all unsure, listen to the designer's reasoning before condemning one of her choices. See Baecker & Buxton (1987), Helander (1991), Laurel (1990, 1991), Rubenstein & Hersh, (1984), Schneiderman (1987), and Smith and Mosier (1984) for excellent introductions to user interface design, including extended discussion of many of the issues raised in this Appendix.

Throughout this Appendix we write as if you were the user of the program. As a tester of it, you will certainly use it heavily. Realize that other people will also use the program and they will have different problems from you. Try to be empathetic.

FUNCTIONALITY
A program has a functionality error if something that you reasonably expect it to do is hard, awkward, confusing, or impossible.

Excessive functionality
This is an error (see Brooks, 1975). It is the hardest one to convince people not to make. Systems that try to do too much are hard to learn and easy to forget how to use. They lack conceptual unity. They require too much documentation, too many help screens, and too much information per topic. Performance is poor. User errors are likely but the error messages are too general. Here's our rule of thumb: A system's level of functionality is out of control if the presence of rarely used features significantly complicates the use of basic features.

Inflated impression of functionality
Manuals and marketing literature should never lead you to believe the program can do more than it can.

Inadequacy for the task at hand
Because a key feature isn't there, is too restricted or too slow, you can't use the program for real work. For example, a database management system that takes 8 hours to sort 1000 records can claim sorting ability but you wouldn't want to use it.

Missing function

A function was not implemented even though it was in the external specification or is "obviously" desirable.

Wrong function
A function that should do one thing (perhaps defined in a specification) does something else.

Functionality must be created by the user
"Systems that supply all the capabilities the user could want but that also require the user to assemble them to make the product work are kits, not finished products." (Rubenstein & Hersh, 1984, p. 45.)

Doesn't do what the user expects
For example, few people would expect a program written to sort a list of names to sort them in ASCII order. They wouldn't expect it to count leading blanks or distinguish between uppercase and lowercase letters. If the programmers insist that the function should work this way, get them to change its name or add the expected behavior as an option.

COMMUNICATION
This section describes errors that occur in communication from the program to the user. Our model is an interactive program, with the person sitting at a computer or terminal. However, batch programs also give information, such as error messages.

Missing information
Anything you must know should be available onscreen. Onscreen access to any other information that the average user would find useful is also desirable.

No onscreen instructions
How do you find out the name of the program, how to exit it, and what key(s) to press for Help? If it uses a command language, how do you find the list of commands? The program might display this information only when it starts. However it does it, you should not have to look in a manual to find the answers to questions like these.

Assuming printed documentation is readily available
Can you use the program after losing your manual? An experienced user should not have to rely on printed documentation.

Undocumented features
If most features or commands are documented onscreen, all should be. Skipping only a few causes much confusion. Similarly, if the program describes "special case" behavior for many commands, it should document them all.

States that appear impossible to exit

How do you cancel a command or back up in a deep menu tree? Programs should allow you to escape from undesired states. Failure to tell you how to escape is almost as bad as not providing an escape path.

No cursor
People rely on the cursor. It points to the place on the screen where they should focus attention. It can also show that the computer is still active and "listening." Every interactive program should show the cursor and display a salient message when turning the cursor off.

Failure to acknowledge input
An interactive program should acknowledge every keystroke by echoing it immediately on the screen. A few exceptions are not errors:

- When choosing a menu item, you are not confused if your keystroke is not echoed, as long as the next screen appears immediately and the words in the screen's title are identical to those of the menu choice.
- If the program ignores erroneous commands or keystrokes, it should not echo them.
- The program should honor your choice if you can tell it not to echo input.
- When you input your password or security code, the program should not echo it on the screen.

Failure to show activity during long delays
When a program is doing a long task (two seconds), it must show you that it's still working, it's not in an infinite loop, and you don't have to reset the computer.

Failure to advise when a change will take effect
A program may execute a command much sooner or later than you expect. For example, it may continue to display erased data until you exit. If it's unclear when the program will do something, customers will perceive it as having bugs and will make many errors.

Failure to check for the same document being opened more than once
Program that allows user to open multiple documents must check for the same document being opened more than once. Otherwise, the user will not be able to keep track of the changes made to the documents since they all have the same name. For example, the file My_Doc is open, if the user attempts to open My_Doc again, there must be the way for users to identify the first My_Doc versus the second one. A typical method for keeping track is to append a number after the file name such as My_Doc:1 and My_Doc:2 for the first and second file respectively. An alternative method is not to allow the same file to be opened twice.

Wrong, misleading, or confusing information
Every error trains you to mistrust everything else displayed by the program. Subtle errors

that lead readers to make false generalizations, such as missing qualifications and bad analogies, annoy testers more than clear factual errors because it's harder to get them fixed.

Simple factual errors
After a program changes, updating screen displays is a low priority task. The result is that much onscreen information becomes obsolete. Whenever the program changes visibly, check every message that might say anything about that aspect of the program.

Spelling errors
Programers dont woory much abuot they're speling misteaks but customers do. Get them fixed.

Inaccurate simplifications
In the desire to keep a feature description as simple as possible, the author of a message may cover only the simplest aspects of the feature's behavior, omitting important qualifications. When she tries to eliminate jargon, she may paraphrase technical terminology inaccurately. Look for these errors. As the tester, you may be the only technically knowledgeable person who carefully reviews the screens.

Invalid metaphors
Metaphors make the computer system seem similar to something you already know. They are good if they help you predict the behavior of the computer and bad if they lead you to incorrect predictions. For example, the trash can icon can be a bad metaphor (Heckel, 1984). If you can take a discarded file out of the trash can, the metaphor is correct. If the file is gone forever once moved to trash, a paper shredder is a better icon.

Confusing feature names
A command named SAVE shouldn't erase a file; nor should it sort one. If a command name has a standard meaning, in the computer community or in the English language, the command must be compatible with its name.

More than one name for the same feature
The program shouldn't refer to the same feature by different names. Customers will waste much time trying to figure out the difference between shadow and drop shadow when the programmer uses both to mean the same thing.

Information overload
Some documents and help screens swamp you with technical detail, to the point of hiding the information or confusing the answers you're looking for. If you think these details are useful, ask whether they are more appropriately located in an appendix in the manual.

Some (not all) detailed discussions of program functioning are disguised apologies from the

programmer or complaints from a writer over a poor design. Do users need this information? Also, is there a solution to the problem the programmer is claiming to be insoluble?

When are data saved?
Suppose you enter information that the program will save. Does it save data as you type, when you exit, when you ask for a save, every few minutes, when? You should always be able to find this out. If you get confused answers, look for bugs immediately. Two modules probably make different assumptions about when the same data will be saved. You can probably get one of them to claim that outdated data are up to date. You may find one module erasing or overwriting data that another has just saved.

Poor external modularity
External modularity refers to how modular the product appears from the outside. How easily can you understand any one piece of it? Poor external modularity increases learning time and scares away new users. Information should be presented as independently as possible. The less you need to know in order to do any particular task the better.

Help text and error messages
Help text and error messages are often considered minor pieces of the product. They may be written by junior programmers or writers. Updating them may be given low priority.

You ask for help or run into error handlers when confused or in trouble.You may be upset or impatient. You will not suffer bad messages gladly. The product will build credibility with you as you use it. If some messages mislead you, the rest may as well not be there.

Inappropriate reading level
People don't read as well at computer terminals (see Schneiderman, 1987). When given their choice of reading levels for onscreen tutorials, experimental subjects preferred Grade 5 (Roehmer and Chapanis, 1982). People reading help or error messages may be distressed: these messages should never be more complicated than tutorial text. Messages should be phrased simply, in short, active voice sentences, using few technical terms even if the readers are computer-experienced.

Verbosity
Messages must be short and simple. Harried readers are infuriated by chatty technobabble. When some users need much more information than others, it's common to give access to further information by a menu. Let people choose where and how much further they want to investigate (Houghton, 1984).

Inappropriate emotional tone
People feel bad enough when they make an error or have to ask for help. They don't need their noses rubbed in it. Look for messages that might make some people feel bad.

Exclamation marks can be interpreted as scolding, so they should not be in error messages. Violent words like "abort," "crash," and "kill" may be frightening or distasteful. Even the word "error" is suspect: many "errors" wouldn't be if the program (or the programmer) were more intelligent. As a final note, many actions may be invalid or unexpected at a computer, but few are truly "illegal."

Factual errors
Help and error messages often give incorrect examples of how to do something "correctly." Some are outdated. Others were never right. Every message should be (re)checked in one of the last testing cycles.

Context errors
Context-sensitive help and error handlers check what you've been doing. They base their messages (recommendations, menu lists, etc.) on that context. This is excellent when it works, but the message makes no sense when they get the context wrong.

Failure to identify the source of an error
At a minimum, an error message should say what's wrong, and unless it's immediately obvious, repeat or point to the erroneous input (data, program line, whatever). A good error message will also say why something is wrong and what to do about it.

Hex dumps are not error messages
An error message that merely says Error 010 or dumps one or more lines of hexadecimal (octal, even decimal) data is acceptable only if the cost of printing a message is outrageous, the computer doesn't have enough time to print a real message before it crashes, or the programmer is the only person who will ever read it.

Forbidding a resource without saying why
If a program tries to use a printer, modem, more memory, or other resource, and can't, the error message should not only announce the failure but should also say why. You need this information because you'll respond differently to Printer already in use versus Printer not connected.

Reporting non-errors
Error messages should only be triggered by error states. You will ignore all error messages if most are normal-case debug messages or reports of events that are rare but not necessarily due to a bug.

Display bugs
Display bugs are visible. If you see many, you are less likely to buy or trust the program. Unfortunately, display bugs are often considered minor and not even investigated. This is risky. They may be symptoms of more serious underlying errors.

Two cursors
It is a nuisance if the programmer forgets to erase the old cursor when he jumps to another part of the screen. Worse, a second cursor might reflect confusion in the code about which area of the screen is active. The program may misinterpret inputs even it echoes them correctly. If the cursor misbehaves during a test, save and examine any data that you enter.

Disappearing cursor
The cursor usually disappears because the programmer displayed a character on top of it or moved it and forgot to redisplay it. However, a program's pointer to the cursor can be corrupted. If it is, then when it points to a memory location used for data or program storage rather than to screen memory, you won't see the cursor. The program will overwrite information in memory whenever it tries to display the cursor.

Cursor displayed in the wrong place
The program shows the cursor in one place but echoes inputs, etc., in another. This is annoying because it leads you to focus on the wrong part of the screen. A slightly misplaced cursor may warn that the program will truncate entered character strings or pad them with garbage. As with dual cursors, entered text may be echoed correctly but saved incorrectly.

Cursor moves out of data entry area
The cursor should never move out of data entry areas. This is usually a coding error but some programmers deliberately let you move the cursor anywhere on the screen, then beep and display an error message that says you can't enter anything here. This is a design error.

Writing to the wrong screen segment
The cursor is in the right location, but data are displayed in the wrong place on the screen.

Failure to clear part of the screen
A message is displayed for a few seconds, then only partially erased. Or your response to a previous question is left onscreen. It is confusing and annoying to have to type over prompts or irrelevant responses in order to enter something new.

Failure to highlight part of the screen
If a program usually highlights a particular class of items, such as prompts or all text in the active window, it must always do so.

Failure to clear highlighting
This is common when attributes of screen positions are stored separately from displayed text. The programmer removes highlighted text but forgets to clear highlighting from that area of the screen. The error is most confusing when the program highlights with double intensity or boldfacing (as opposed to inverse video, for example). The blank screen area looks fine. The

problem only becomes evident when new text is displayed: it is always highlighted.

Wrong or partial string displayed
The displayed message might be garbage text, a segment of a longer message, or a complete message that should appear some other time. Any of these might reflect errors in the program logic, the values of the pointers used to find message text, or the stored copies of the text. They might indicate minor problems or severe ones.

Messages displayed for too long or not long enough
Many messages are displayed for a fixed time, then erased automatically. The message should be onscreen long enough to be noticed and then read. Unimportant messages can be cleared sooner than critical ones. Short messages can go sooner than long ones. Messages that come up frequently and are easily recognized can be displayed for less time than rare ones.

Be suspicious when the same message sometimes displays briefly and sometimes lasts longer. This may reflect unanticipated race conditions. Try to figure out how to obtain which delays, then get this looked into carefully by the programmer.

Display layout
The screen should look organized. It should not be cluttered. Different classes of objects should be displayed separately, in predictable areas. There are many guidelines for display layout, but they boil down to this: it should be easy to find what you want on the screen.

Poor aesthetics in the screen layout
The screen may be unbalanced, rows or columns may not be aligned, or it might just look "bad." Use your good taste. If you're not confident, get a second opinion. If it looks badly laid out to you, something is probably wrong even if you can't articulate the problem yet.

Menu layout errors
Schneiderman (1987) and Smith and Mosier (1984) cover this area well. Their discussions run to many more pages than are available here. Here are a few points that we want to emphasize:

- Similar or conceptually related menu choices should be grouped. Groups should be clearly separated.
- The action required to select a menu item should be obvious or should be stated onscreen.
- Menu selections should generally be independent. To achieve a single result, the customer shouldn't have to make two or more different selections on different menus.
- Selecting a menu item by typing its first letter is usually better than selecting it by a number. However, all items have to start with different letters for this to work well.

Watch out for assignments of odd names to menu items.

Dialog box layout errors

For further reference in this area, we recommend IBM's SAA Advanced Interface Design Guide (1989), and SAA Basic Interface Design Guide (1989), and Apple's Human Interface Guidelines (1987).

- Dialog boxes should operate consistently. For example, they should use consistent capitalization, spelling, and text justification rules.Dialog box titles should occupy a consistent place and match the name of the command used to call up the dialog. The same shortcut keys should work from dialog to dialogó shouldn't cancel out of some dialogs (no changes made) but complete (all changes accepted) others.
- Controls in the dialog box must be arranged logically. Group related controls together, and separate groups by using proper spacing.
- Selection and entry fields should be aligned vertically and horizontally so users can navigate the cursor movement in a straight-line pattern.
- Watch out for interdependencies across dialogs. It is confusing when a selection in one dialog box determines which options are available in another box.

Obscured instructions

You should always know where to look to find out what to do next. If the screen is at all crowded, an area should be reserved for commands and messages. Once you understand this convention, you'll know where to focus attention. It may also be good to blaze critical information across the center of the screen, no matter what used to be there.

Misuse of flash

Flashing pictures or text are noticeable; lots of flashing is confusing and intimidating. You should be able to tell immediately why an object is flashing. Excessive or ambiguous flashing is an eyesore, not a good alert.

Misuse of color

Too much color can be distracting, make the text harder to read, and increase eye strain. Color shouldn't make the screen look busy or distracting. Programs like word processors, spreadsheets, and databases should use highlighting colors sparingly. Most text should be one color. You should also complain if the program's combinations of colors looks ugly.

Heavy reliance on color

Programs limit their audience severely if they use color as the only differentiator between items. What happens to a colorblind person or someone with a monochrome monitor? A few applications may not be worth running in monochrome, but many others (including drawing programs and many games) don't need color.

Inconsistent with the style of the environment
If the style associated with a computer offers certain consistencies and conveniences, you'll notice their absence from any one program. Even if the programmer thinks he can replace them with "better" ones, many people will resent having to learn a new series of conventions. For example, if the operating system and most applications are mouse and icon-based, an application that requires typed command words will feel inappropriate. New programs should also follow the lead when most others display error messages in a certain way, on a certain place onscreen.

Cannot get rid of onscreen information
It's great (often essential) to have a menu of command choices available on part of the screen. However, once you become proficient with the program, the menu is a waste of screen space. You should be able to issue a command to get rid of it, and another to call it back as needed.

COMMAND STRUCTURE AND ENTRY
This section deals with the way the program organizes commands and presents them to you, and with how you enter those commands. Schneiderman (1987) considers how to choose among the different command entry styles. There are many choices. This section assumes that the programmer's choice of style was reasonable. It deals only with flaws in the implementation.

Inconsistencies
Increasing the number of always-true rules shortens learning time and documentation and makes the program more professional-looking. Inconsistencies are so common because it takes planning and agony to choose a rule of operation that can always be followed. It is so tempting to do things differently now and again. Each minor inconsistency seems insignificant, but together they quickly make an otherwise well conceived product hard to use. It is good testing practice to flag all inconsistencies, no matter how minor.

"Optimizations"
Programmers deliberately introduce inconsistencies to optimize a program. Optimizations are tempting since they tailor the program to your most likely present need. But each new inconsistency brings complexity with it. Make the programmer aware of the tradeoff in each case. Is saving a keystroke or two worth the increase in learning time or the decrease in trust? Usually not.

Inconsistent syntax
Syntactic details should be easily learned. You should be able to stop thinking about them. Syntax of all commands should be consistent throughout the program. Syntax includes such things as:

- the order in which you specify source and destination locations (copy from source to destination or copy to destination from source)
- the type of separators used (spaces, commas, semicolons, slashes, etc.)
- the location of operators (infix (A+B), prefix (+AB), postfix (AB+)).

Inconsistent command entry style
You can select a command by pointing to it, pressing a function key, or typing its name, abbreviation, or number. A program should use one command style. If the program offers alternative styles for doing the same task, it should offer the same alternatives everywhere. If the program must switch styles across different parts of the program, it must make it clear when to use which.

Inconsistent abbreviations
Without clear-cut abbreviation rules, abbreviations can't be easily remembered. Abbreviating delete to del but list to ls and grep to grep makes no sense. Each choice is fine individually, but the collection is an ill-conceived mess of special cases.

Inconsistent termination rule
Fill-in-the-blanks forms only allow so much room for a command name or data item. Suppose an entry can be eight characters long. When you enter seven characters or less, you have to say you're done by pressing or some other terminator (, , , etc.). If you enter an eight-character name, some programs act on it without waiting for the terminator. This is confusing.

The program should require terminators for multi-key entries. People rarely remember commands (or data) in terms of how many letters they are. They will habitually enter the eighth character then press . If the program has already supplied its own , the extra one typed by the user is an annoying input "error."

Inconsistent command options
If an option makes sense for two commands, it should be available with both (or neither), should have the same name, and should be invoked in the same sequence in both cases.

Similarly named commands
It is easier to confuse two different commands if their names are similar.

Inconsistent capitalization
If command entry is case sensitive, first letters of all commands should all be capitalized or none should be. First letters of embedded words in commands should always or never be capitalized.

Inconsistent menu position

It's hard to keep the same command in the same position on different menus if it occurs in many submenus, but with work and care the programmer can frequently achieve this.

Inconsistent function key usage
The meaning of function keys should remain constant across the program. Reversals (sometimes saves data and deletes, other times deletes and saves) are unacceptable.

Inconsistent error handling rules
When the program detects an error, it may announce it or attempt to correct it. After handling the error, the program may stop, restart, or return to its last state. The error handler may change data on disk or save new information. Error handlers can vary a great deal. The behavior of any one program's should be completely predictable.

Inconsistent editing rules
The same keys and commands should be available to change any datum as you entered it or examined it later.

Inconsistent data saving rules
The program should save data in the same way everywhere, with the same timing and scope. It shouldn't sometimes save data as each field is entered, but other times save at the end of a record, a group of records, or just before exit.

Time-wasters
Programs that seem designed to waste your time infuriate people.

Garden paths
A program leads you down the garden path if you must make choice after choice to get to a desired command, only finding at the end that it's not there, wasn't implemented, or can't be used unless you do something else (down a different path) first. Look for these problems in complex menu trees.

Choices that can't be taken
There is no excuse for including choices in a menu that cannot be made. How can you view, save or erase the data if there are no data? How can you print the document if there is no printer? How dare the programmer say, Press for Help, then when you press it say, Sorry, Help Not Available at this Level (whatever that means)?

Are you really, really sure?
Programs should ask you to confirm critically destructive commands. You should have to tell the computer twice to reformat a data-filled disk. The program should not pester you for confirmation of every little deletion. You become annoyed; you'll learn to answer Yes automatically, even in the critical cases when you should think first.

Obscurely or idiosyncratically named commands
Command names should be informative. You should not have to constantly look up the definition of a command name in the manual, until you eventually memorize it. There is no excuse for names like grep, finger, and timehog in products released to the public.

Menus
Menus should be simple, but they become complex when there are poor icons or command names and when choices hide under nonobvious topic headings. The more commands a menu covers, the more complex it will be no matter how well planned it is. But without planning, complex menus can become disasters.

Excessively complex menu hierarchy
If you have to wade through menu after menu before finally reaching the command you want, you'll probably want to use another program. Programmers who create deep menu trees cite the design rule that says no menu should have more than seven choices. This may be best for novices. Experienced users prefer many more choices per menu level, make fewer errors and respond more quickly, so long as the choices are well organized, neatly formatted, and not ridiculously crowded or abbreviated. Paap & Roske-Hostrand (1986) and MacGregor, Lee, & Lam (1986) provide some interesting discussion.

Inadequate menu navigation options
In even a modestly deep menu structure, you must be able to move back to the previous menu, move to the top of the menu structure, and leave the program at any time. If there are hundreds of topics, you should also be able to jump directly to any topic by entering its name or number.

Too many paths to the same place
The program needs reorganization if many commands reappear in many menus. It can be handy to have a command repeated in different places, but there are limits. You should worry about the program's internal structure and reliability if it feels like you can get anywhere from anywhere.

You can't get there from here
Some programs lock you out of one set of commands once you've taken a different path. You have to restart to regain access to them. This is usually unnecessary.

Related commands relegated to unrelated menus
Grouping commands or topics in a complex menu is not easy. It is easy to overlook what should be an obvious relationship between two items, and to arbitrarily assign them to separate menus. When you report these, explain the relationship between the two items, and suggest which menu both belong in.

Unrelated commands tossed under the same menu
Some commands are dumped under a totally unrelated heading because someone thought it would take too much work to put them where they belong, which may involve adding a new higher level heading and reorganizing.

Command lines
It is harder to type the command name without a hint than to recognize it in a menu, but experienced users prefer command line entry when there are many commands and many options. The menu system feels too bulky to them. Anything that makes it easier to remember the command names and options correctly is good. Anything that makes errors more likely is bad.

Forced distinction between uppercase and lowercase
Some programs won't recognize a correctly spelled command name if it isn't capitalized "correctly." This is more often a nuisance than a feature.

Reversed parameters
The most common example is the distinction between source and destination files. Does COPY FILE1 FILE2 mean copy from FILE1 to FILE2 or from FILE2 to FILE1? The order doesn't matter (people can get used to anything) as long as it stays consistent across all commands that use a source and a destination file. Application programs must follow the operating system's ordering conventions.

Full command names not allowed
Abbreviations are fine, but you should always be able to type delete, not just del. The full name of a command is much more reliably remembered than the abbreviation, especially if there are no consistent abbreviating rules.

Abbreviations not allowed
You should be able to enter del instead of having to type delete in full. Enough systems don't allow abbreviations that we can't class their absence a design error, but, implemented properly, it sure is a nice feature.

Demands complex input on one line
Some programs require complicated command specifications (Do X for all cases in which A or B and C is true unless D is false). You will make many mistakes if you have to specify compound logical operators as part of a one-line command. Fill-in-the-blank choices, sequential prompting, and query by example, are all more appropriate than command line entry with compound logical scope definition.

No batch input
You should be able to type and correct a list of commands using an editor, then tell the

computer to treat this list as if you had typed each command freshly at the keyboard.

Can't edit commands
You should be able to backspace while typing a command. If you try to execute an incorrectly typed command, you should be able to call it back, change the erroneous piece, and re-execute it.

Inappropriate use of the keyboard
If a computer comes with a standard keyboard, with labeled function keys that have standard meanings, new programs should meet that standard.

Failure to use cursor, edit, or function keys
It doesn't matter if a program was ported from some other machine that doesn't have these keys. Users of this machine will expect their keys to work.

Non-standard use of cursor and edit keys
The keys should work the way they usually work on this machine, not the way they usually work on some other machine, and definitely not in some totally new way.

Non-standard use of function keys
If most other programs use as the Help key, defining it as Delete-File-And-Exit in this program is crazy or vicious.

Failure to filter invalid keys
The program should trap and discard invalid characters, such as letters if it adds numbers. It should not echo or acknowledge them. Ignoring them is less distracting than error messages.

Failure to indicate keyboard state changes
Lights on the keyboard or messages on the screen should tell you when Caps Lock and other such state changing features are on.

Failure to scan for function or control keys
You should be able to tell the computer to quit what it's doing (, for example). The program should also always recognize any other system-specific keys, such as , that programs on this machine usually recognize quickly.

MISSING COMMANDS
This section discusses commands or features that some programs don't, but should, include.

State transitions
Most programs move from state to state. The program is in one state before you choose a menu item or issue a command. It moves into another state in response to your choice.

Programmers usually test their code well enough to confirm that you can reach any state that you should be able to reach. They don't always let you change your mind, once you've chosen a state.

Can't do nothing and leave
You should be able to tell an interactive program that you made your last choice by mistake, and go back to its previous state.

Can't quit mid-program
You should be able to quit while using a program without adversely affecting stored data. You should be able to stop editing or sorting a file, and revert to the version that was on disk when you started.

Can't stop mid-command
It should be easy to tell the program to stop executing a command. It shouldn't be hard to return to your starting point, to make a correction or choose a different command.

Can't pause
Some programs limit the time you have to enter data. When the time is up, the program changes state. It might display help text or accept a displayed "default" value, or it may log you off. Although time limits can be useful, people do get interrupted. You should be able to tell it that you are taking a break, and when you get back you'll want it in the same state it's in now.

Disaster prevention
System failures and user errors happen. Programs should minimize the consequences of them.

No backup facility
It should be easy to make an extra copy of a file. If you're changing a file, the computer should keep a copy of the original (or make it easy for you to tell it to keep it) so you have a known good version to return to if your changes go awry.

No undo
Undo lets you retract a command, typically any command, or a group of them. Undelete is a restricted case of undo that lets you recover data deleted in error. Undo is desirable. Undelete is essential.

No "Are you sure?"
If you issue a command that will wipe out a lot of work, or that wipes out less but is easy to issue in error, the program should stop you and ask whether you want the command executed.

No incremental saves
When entering large amounts of text or data, you should be able to tell the program to save your work at regular intervals. This ensures that most of your work will have been saved if the power fails. Several programs automatically save your work in progress at regular intervals. This is an excellent feature, so long as the customer who is bothered by the delay during saving can turn it off.

Error handling by the user
People can catch their own errors and recognize from experience that they are prone to others. They should be able to fix their work and, as much as possible, build in their own error checks.

No user-specifiable filters
When designing data entry forms, spreadsheet templates, etc., you should be able to specify, for each field, what types of data are valid and what the program should ignore or reject. As examples, you might have the program reject anything that isn't a digit, a letter, a number within a certain range, a valid date, or an entry that matches an item in a list stored on disk.

Awkward error correction
It should be easy to fix a mistake. You should never have to stop and restart a program just to return to a data entry screen where you made an error. You should always be able to back up the cursor to a field on the same screen in which you entered, or could have entered, data. When entering a list of numbers, you should be able to correct one without redoing the rest.

Can't include comments
When designing data entry forms, spreadsheet templates, expert systemsóanything in which you are, in effect, writing a programóyou should be able to enter notes for future reference and debugging.

Can't display relationships between variables
Variables in entry forms, spreadsheet templates, etc., are related. It should be easy to examine the dependence of any variable on the values of others.

Miscellaneous nuisances

Inadequate privacy or security
How much security is needed for a program or its data varies with the application and the market. On multi-user systems you should be able to hide your files so no one else can see them, and encrypt them so no one elseónot even the system administratorócan read them. You should also be able to lock files so no one else can change (or delete) them. Beizer (1984) discusses security in more detail.

Obsession with security
The security controls of a program should be as unobtrusive as possible. If you are working at your own personal computer at home, you should be able to stop a program from pestering you for passwords.

Can't hide menus
Many programs display a menu at the top, bottom, or side of the screen. They use the rest of the screen for data entry and manipulation. The menus are memory aids. Once you know all the commands you need, you should be able to remove the menus and use the full screen for entry and editing.

Doesn't support standard O/S features
For example, if the operating system uses subdirectories, program commands should be able to reference files in other subdirectories. If the O/S defines "wildcard" characters (such as * to match any group of characters), the program should recognize them.

Doesn't allow long names
Years ago, when memory was scarce and compilers were sluggish, it was necessary to limit the length of file and variable names to six or eight characters. We're past those days. Meaningful names are among the best possible forms of documentation. They should be allowed.

PROGRAM RIGIDITY
Some programs are very flexible. You can change minor aspects of their functioning easily. You can do tasks in any order you want. Other programs are utterly inflexible. Rigidity isn't always bad. The fewer choices and the more structured the task, the more easily (usually) you'll learn the program. And you won't be confused by aspects of a program's operation that can't be changed without affecting the others. On the other hand, different people will like different aspects of the program and dislike others. If you change these to suit your taste, you'll like the program more.

User tailorability
You should be able to change minor and arbitrary aspects of the program's user interface with a minimum of fuss and bother.

Can't turn off the noise
Many programs beep when you make errors and provide a loud key click that sounds every time you touch the keyboard. Auditory feedback is useful but in shared work areas, computer noises can be annoying. There must be a way to turn them off.

Can't turn off case sensitivity
A system that can distinguish between uppercase and lowercase should allow you to tell it to

ignore cases.

Can't tailor to hardware at hand
Some programs are locked to input/output devices that have specific, limited capabilities. People who upgrade their equipment either can't use these programs or can't take advantage of the new devices' features. Experienced users should be able to tailor a program to the hardware. You should be able to change control codes sent to a printer and copy a program onto any mass storage device. It should not be impossible to use a mouse with any interactive program.

Can't change device initialization
An application program should either be able to send user-defined initializers or it should leave well enough alone. Suppose you want to send control codes to a printer to switch to condensed characters. If the program that prints the data doesn't let you initialize the printer, you have to change the printer mode from the device, then run the program. Some programs, however, defeat your printer setup by always sending the printer their own, inflexible, set of control codes. This is a design error.

Can't turn off automatic saves
Some programs protect you against power failures by automatically saving entered data to disk periodically. In principle this is great but in practice the pauses while it saves the data can be disruptive. Also, the program assumes that you always want to save your data. This assumption might not be true. You should be able to turn this off.

Can't slow down (speed up) scrolling
You should be able to slow down the screen display rate so you can read text as it scrolls by.

Can't do what you did last time

Can't find out what you did last time
You should be able to re-issue a command, examine it, or edit it.

Failure to execute a customization command
If the program lets you change how it interacts with you, your changes should take effect immediately. If a restart is unavoidable, the program should say so. You should not have to wonder why a command wasn't executed.

Failure to save customization commands
You should not only be able to tell the computer to turn off its beeps and clicks now, but should also be able to tell it to turn them off and keep them off forever.

Side-effects of feature changes

Changing how one feature operates should not affect another. When there are side effects, they should be well documented when you change the feature setting, in the manual and onscreen.

Infinite tailorability
You can change virtually all aspects of some programs. This flexibility can be good, but you have to step back from the program to figure out how it should work. To make the decisions intelligently you have to develop an expert user's view of the program itself along with learning the command language.

Programs this flexible usually have horrid user interfaces. The developers spent their energy making the program adjustable, and didn't bother making the uncustomized product any good. Since everyone will change it, they reason, its initial command set doesn't mean anything anyway. Such a program is terrible for novices and occasional users. It isn't worth their time (sometimes weeks or months) to figure out how to tune the program to their needs, but without the tuning, the program is only marginally usable.

The user interface of customizable products should be fully usable without modification. You should subject it to the same rigorous criticism applied to less flexible ones. Many people will use the uncustomized version for a long time.

Who's in control
Some programs are high-handed. Their error and help messages are condescending. Their style is unforgivingóyou can't abort commands or change data after entering them. None of this is acceptable. Programs should make it easier and more pleasant for you to get a task done as quickly as possible. They should not second-guess you, force a style on you, or waste your time.

Unnecessary imposition of a conceptual style
Some programs demand that you enter data in a certain order, that you complete each task before moving to the next, that you make decisions before looking at their potential consequences. Examples:

- When designing a data entry form, why must you specify a field's name, type, width, or calculation order, before drawing it onscreen? As you see how the different fields look together, won't you change some fields, move them around, even get rid of a few? You may have to enter field specifications before using the form, but subject to that restriction, you should decide when to fill in the details.
- When describing tasks to a project management system, why must you list all tasks first, all available people second, then completely map the work assigned to one individual before entering any data for the next? Since you're probably trying to figure out what to assign to whom, won't you want to change these data as you see

their consequences?

A surprising number of limits exist because some programmer decided that people should organize their work in a certain way. For "their own good" he won't let them deviate from this "optimal" approach. He is typically wrong.

Novice-friendly, experienced-hostile
Programs optimized for novices break tasks into many small, easily understood steps. This can be good for the newcomer, but anyone experienced with the system will be frustrated if they can't get around it.

Artificial intelligence and automated stupidity
In the names of "artificial intelligence" and "convenience" some programs guess what you want next and execute those guesses as if they were user-issued commands. This is fine unless you don't want them done. Similarly, the program that automatically corrects errors is great until it "corrects" correct data. People make enough of their own mistakes without having to put up with ones made by a program that's trying to second-guess them. Better than automatic execution, especially of something that takes noticeable time or changes data, the program should give you a choice. You should be able to set it to wait until you type Y (yes) before executing its suggestions. If you say No the program should abandon its suggestion and ask for new input.

Superfluous or redundant information required
Some programs ask for information they'll never use, or that they'll only use to display onscreen once, or ask you to re-enter data you've already enteredónot to check it against the old copy, just to get it again. This is a surprisingly common waste of time.

Unnecessary repetition of steps
Some programs make you re-enter the works if you make one mistake in a long sequence of command steps or data. Others force you to re-enter or confirm any command that might be in error. To do something "unusual" you may have to confirm every step. Repetitions or confirmations that are not essential are a waste of your time.

Unnecessary limits
Why restrict a database to so many fields or records, a spreadsheet cell to digits only, a project manager to so many tasks, a word processor to so many characters? Limits that aren't essential for performance or reliability shouldn't be limits.

PERFORMANCE
Many experienced users consider performance the most important aspect of usability: with a fast program, they feel more able to concentrate and more in control. Errors are less

important because they can be dealt with quickly. With few exceptions, reviewed by Schneiderman (1987), the faster the better.

Performance has different definitions, such as:

- Program Speed: how quickly the program does standard tasks. For example, how quickly does a word processor move to the end of the file?
- User Throughput: how quickly you can do standard tasks with the program. These are larger scale tasks. For example, how long does it take to enter and print a letter?
- Perceived Performance: How quick does the program seem to you?

Program speed is a big factor no matter how you define performance, but a fast program with a poorly designed user interface will seem much slower than it should.

Slow program
Many design and code errors can slow a program. The program might do unnecessary work, such as initializing an area of memory that will be overwritten before being read. It might repeat work unnecessarily, such as doing something inside a loop that could be done outside of it. Design decisions also slow the program, often more than the obvious errors.

Whatever the reason for the program being slow, if it is, it's a problem. Delays as short as a quarter of a second can break your concentration, and substantially increase your time to finish a task.

Slow echoing
The program should display inputs immediately. If you notice a lag between the time you type a letter and the time you see it, the program is too slow. You will be much more likely to make mistakes. Fast feedback is essential for any input event, including moving mice, trackballs, and light pens.

How to reduce user throughput
A lightning quick program will be molasses for getting things done if it slows the person working with it. This includes:

- anything that makes user errors more likely.
- slow error recovery, such as making you re-enter everything if you make a mistake when entering a long series of numbers or a complicated command.
- anything that gets you so confused that you have to ask for help or look in the manual.
- making you type too much to do too little: no abbreviations, breaking a task into tiny subtasks, requiring confirmation of everything, and so on.

Other sections of the Appendix describe specific errors along these lines. One tactic for applying pressure to fix user interface errors involves a comparative test of user throughput. Compare the product under development with a few competitors. If people take longer to do things with your program, and if much of the delay can be attributed to user interface design errors, these errors take on a new significance.

Poor responsiveness
A responsive program doesn't force you to wait before issuing your next command. It constantly scans for keyboard (or other) input, acknowledges commands quickly, and assigns them high priority. For example, type a few lines of text while your word processor is reformatting the screen. It should stop formatting, echo the input, format the display of these lines as you enter them, and execute your editing commands. It should keep the area of the screen near the cursor up to date. The rest is lower priority since you aren't working with it at this instant. The program can update the rest of the display when you stop typing. Responsive programs feel faster.

No type-ahead
A program that allows type-ahead lets you keep typing while it goes about other business. It remembers what you typed and displays and executes it later. You should not have to wait to enter the next command.

No warning that an operation will take a long time
The program should tell you if it needs more than a few seconds to do something. You should be able to cancel the command. For long jobs, it should tell you how long so you can use the time rather than waste it waiting.

No progress reports
For long tasks or delays, it is very desirable to indicate how much has been done and how much longer the machine will be tied up (Myers, 1985).

Problems with time-outs
Some programs limit the time you have to enter data. When the time's up, the program changes state. Except for arcade games, you should not have to race against a short time-out to stop the program from executing an undesired command.

Time-outs can also be too long. For example, you might have a short interval before a program does something reasonable but time consuming. If you respond during the interval, you shortcut the task (e.g., a menu isn't displayed unnecessarily). Program speed will suffer if this isn't kept short.

Time-out intervals can simultaneously be too long for people who wait them out and too

short for others who try to enter data during them.

Program pesters you
BEEP! Are you sure?
BEEP! Your disk is now 85% full. Please swap disks shortly.
BEEP! Are you really sure?
BEEP! You haven't saved your text for the last hour.
BEEP! Your disk is now 86% full. Please swap disks shortly.
BEEP! Please re-enter your security code.
BEEP! You haven't entered anything for 10 minutes. Please log off.
BEEP! Your disk is now 86% full. Please swap disks shortly.
BEEP! 14 messages still unanswered.

Reminders, alerts, and queries can be useful but if they are frequent they can be annoying.

Do you really want help and graphics at 300 baud?
On a slow terminal, help text, long menus, and pretty pictures are usually irritating. You should be able to use a terse command language instead. Similarly, programs that format output in a beautiful but time-consuming way on a printer should have a "fast and ugly" mode for drafts.

OUTPUT
Program output should be as complete as desired and intelligible to the human or program that must read it. It should include whatever you want, in whatever format you want. It should go to whatever device you want it to go. These requirements are real, but too general to (usually) achieve. You'll probably add many other annoyances and limitations to the following list.

Can't output certain data
You should be able to print any information you enter, including technical entries like formulas in spreadsheets and field definitions. If it's important enough to enter, it's important enough for you to be able to print and desk check it.

Can't redirect output
You should be able to redirect output. In particular, you should be able to send a long "printout" to the disk and print this disk file later. This gives you the opportunity to polish the output file with a word processor or print it with a program that can print files more quickly or as a background task.

The program should not stop you from sending output to unexpected devices, such as plotters, laser printers, and cassette tapes.

Format incompatible with a follow-up process
If one program is supposed to be able to save data in a format that a second can understand, you must test that it does. This means buying or borrowing a copy of the second program, saving the data with the first, reading it with the second, and looking at what the second program tells you it got. This test is all too often forgotten, especially when the second program was not made by the company that developed the first.

Must output too little or too much
You should be able to modify reports to present only the information you need. Having to dig through pages of printouts that contain just a few lines of useful information is nearly as bad as not getting the information.

Can't control output layout
You should be able to emphasize information by changing fonts, boldfacing, underlining, etc. You should also be able to control the spacing of information; that is, you should be able to group some sections of information and keep others separate. At a minimum, the program should be able to print reports to a disk file, in a format suitable for touchups by a word processor.

Absurd printed level of precision
It is silly to say that 4.2 + 3.9 is 8.100000 or that the product of 4.234 and 3.987 is 16.880958. In final printouts, the program should round results to the precision of the original data, unless you tell it to do otherwise.

Can't control labeling of tables or figures
You should be able to change the typeface, wording, and position of any caption, heading, or other text included in a table, graph or chart.

Can't control scaling of graphs
Graphing programs should provide default vertical and horizontal scales, but you should be able to override the defaults.

ERROR HANDLING
Errors in dealing with errors are among the most common bugs. Error handling errors include failure to anticipate the possibility of errors and protect against them, failure to notice error conditions, and failure to deal with detected errors in a reasonable way. Note that error messages were discussed above.

ERROR PREVENTION
Yourdon's (1975) chapter on Antibugging is a good introduction to defensive programming.

The program should defend itself against bad input and bad treatment by other parts of the system. If the program might be working with bad data, it should check them before it does something terrible.

Inadequate initial state validation
If a region of memory must start with all zeros in it, maybe the program should run a spot check rather than assuming that zeros are there.

Inadequate tests of user input
It is not enough to tell people only to enter one- to three-digit numbers. Some will enter letters or ten-digit numbers and others will press five times to see what happen. If you can enter it, the program must be able to cope with it.

Inadequate protection against corrupted data
There's no guarantee that data stored on disk are any good. Maybe someone edited the file, or there was a hardware failure. Even if the programmer is sure that the file was validated before it was saved, he should include checks (like a checksum) that this is the same file coming back.

Inadequate tests of passed parameters
A subroutine should not assume that it was called correctly. It should make sure that data passed to it are within its operating range.

Inadequate protection against operating system bugs
The operating system has bugs. Application programs can trigger some of them. If the application programmer knows, for example, that the system will crash if he sends data to the printer too soon after sending it to the disk drive, he should make sure that his program can't do that under any circumstances.

Inadequate version control
If the executable code is in more than one file, someone will try to use a new version of one file with an old version of another. Customers upgrading their software make this mistake frequently enough, then don't understand what's wrong unless the program tells them. The new version should include code that checks that all code files are up to date.

Inadequate protection against malicious use
People will deliberately feed a program bad input or try to trigger error conditions. Some will do it out of anger, others because they think it's fun. Saying that "no reasonable person would do this" provides no defense against the unreasonable person.

ERROR DETECTION
Programs often have ample information available to detect an error in the data or in their

operation. For the information to be useful, they have to read and act on it. A few commonly ignored symptoms or pieces of diagnostic information are described below. There are many others.

Ignores overflow
An overflow condition occurs when the result of a numerical calculation is too big for the program to handle. Overflows arise from adding and multiplying large numbers and from dividing by zero or by tiny fractions. Overflows are easy to detect, but the program does have to check for them, and some don't.

Ignores impossible values
The program should check its variables to make sure that they are within reasonable limits. It should catch and reject a date like February 31. If the program does one thing when a variable is 0, something else when it is 1, and expects that all other values are "impossible," it must make sure that the variable's value is 0 or 1. Old assumptions are unsafe after a few years of maintenance programming.

Ignores implausible values
Someone might withdraw $10,000,000 from their savings account but the program should probably ask a few different humans for confirmation before letting the transaction go through.

Ignores error flag
The program calls a subroutine, which fails. It reports its failure in a special variable called an error flag. The program can either check the flag or, as often happens, ignore it and treat the garbage data coming back from the routine as if it was a real result.

Ignores hardware fault or error conditions
The program should assume that devices it can connect to will fail. Many devices can send back messages (set bits) that warn that something is wrong. If one does, the program should stop trying to interact with it and should report the problem to a human or to a higher level control program.

Data comparisons
When you try to balance your checkbook, you have the number you think is your balance and the number the bank tells you is your balance. If they don't agree after you allow for service charges, recent checks, and so forth, there is an error in your records, the bank's, or both. Similar opportunities frequently arise to check two sets of data or two sets of calculations against each other. The program should take advantage of them.

ERROR RECOVERY
There is an error, the program has detected it, and is now trying to deal with it. Much error

recovery code is lightly tested, or not tested at all. Bugs in error recovery routines may be much more serious than the original problems.

Automatic error correction
Sometimes the program can not only detect an error but correct it, without having to bother anyone about it, by checking other data or a set of rules. This is desirable, but only if the "correction" is correct.

Failure to report an error
The program should report any detected internal error even if it can automatically correct the error's consequences. It might not detect the same error under slightly different circumstances. The program might report the error to the user, to the operator of a multi-user system, to an error log file on disk, or any combination of these, but it must be reported.

Failure to set an error flag
A subroutine is called and fails. It is supposed to set an error flag when it does fail. It returns control to the calling routine without setting the flag. The caller will treat the garbage data passed back as if they were valid.

Where does the program go back to?
A section of code fails. It logs the problem, sets an error flag, then what? Especially if the failing code can be reached from several GOTO statements, how does it know where in the program to return control to?

Aborting errors
You stop the program, or it stops itself when it detects an error. Does it close any open output files? Does it log the cause of the exit on its way down? In the most general terms, does it tidy up before dying or does it just die and maybe leave a big mess?

Recovery from hardware problems
The program should deal with hardware failures gracefully. If the disk or its directory is full, you should be able to put in a new one, not just lose all your data. If a device is unready for input for a long time, the program should assume that it's off line or disconnected. It shouldn't sit waiting forever.

No escape from missing disk
Suppose your program asks you to insert a disk that has files it needs. If the inserted disk is not the correct one, it will prompt you again until the correct disk is inserted. However, if the correct disk is not available, there is no way you can escape unless you reboot your system.

BOUNDARY-RELATED ERRORS
A boundary describes a change-point for a program. The program is supposed to work one

way for anything on one side of the boundary. It does something different for anything on the other side.

The classic "things" on opposite sides of boundaries are data values. There are three standard boundary bugs:

- Mishandling of the boundary case: If a program adds any two numbers that are less than 100, and rejects any greater than 100, what does it do when you enter exactly 100? What is it supposed to do?
- Wrong boundary: The specification says the program should add any two numbers less than 100 but it rejects anything greater than 95.
- Mishandling of cases outside the boundary: Values on one side of the boundary are impossible, unlikely, unacceptable, unwanted. No code was written for them. Does the program successfully reject values greater than 100 or does it crash when it gets one?

We treat the concept of boundaries more broadly. Boundaries describe a way of thinking about a program and its behavior around its limits. There are many types of limits: largest, oldest, latest, longest, most recent, first time, etc. The same types of bugs can happen with any of them so why not think of them in the same terms?

NUMERIC BOUNDARIES
Some numeric boundaries are arbitrary (bigger or less than 100) while others represent natural limits. A triangle has exactly three sides (not more, not less). Its angles sum to 180 degrees. A byte can store a (nonnegative) number between 0 and 255. If a character is a letter, its ASCII code is between 65 and 90 (capitals) or between 97 and 122 (lowercase).

EQUALITY AS A BOUNDARY
Every element in a list might be the same. Every element might be different. What happens if you try to sort either list? If the list is made of numbers, what happens if you try to compute their mean, standard deviation, coefficient of symmetry or kurtosis? (All four are summary statistics. The last two would either compute to 0 or cause a divide by 0 error depending on the calculation algorithm.)

BOUNDARIES ON NUMEROSITY
An input string can be up to 80 characters long? What if you type 79, 80 or 81? What does the program receiving your input do in each case? Can a list have one element? No elements? What is the standard deviation of a one-number list of numbers? (Answer: undefined or zero)

BOUNDARIES IN SPACE

For example, if a graphing program draws a graph and a box around it, what to do with a dot that should properly be displayed outside the box?

BOUNDARIES IN TIME

Suppose the program displays a prompt, waits 60 seconds for you to respond, then displays a menu if you haven't typed anything. What happens if you start typing just as it's starting to display the menu?

Suppose you have 30 seconds to answer a ringing telephone. After that, your phone stops ringing and the call forwards to the operator. Do you lose the call if you reach it at the 30th second? What if you reach it after the 30th second but before the operator has answered it?

Suppose you press the Space Bar while the computer's still loading the program from the disk. What happens? Is <Space> sent to the operating system (which is loading the program), saved for the program being loaded, or is this just so unexpected that it crashes the computer?

Race conditions reflect temporal boundaries.

BOUNDARIES IN LOOPS

Here's an example of a loop:

```
10  IF COUNT_VARIABLE is less than 45
     THEN   PRINT "This is a loop"
         SET COUNT_VARIABLE to COUNT_VARIABLE + 1
         GOTO 10
     ELSE quit
```

The program keeps printing and adding 1 to COUNT_VARIABLE until the counter finally reaches 45. Then the program quits. 45 bounds the loop. Loops can have lower as well as upper bounds (IF COUNT_VARIABLE is less than 45 and greater than 10). Beizer (1990) discusses tests of loop boundaries.

BOUNDARIES IN MEMORY

What are the largest and smallest amounts of memory that this program can cope with? (Yes, a few programs do crash if you give them access to too much memory.) Are data split across pages or segments of memory? Is the first or last byte of a segment lost or misread? (By the way, is that first byte numbered 0 or 1?) Are some of the data in RAM and some on disk, in virtual memory format? Suppose the program reads a value from RAM then a value from

virtual memory, then the next value from what used to be in RAM, then back to what used to be (still is? is again?) virtual memory, etc. How seriously will this back-and-forth affect performance?

BOUNDARIES WITHIN DATA STRUCTURES

Suppose the program keeps data in a record structure. Each record holds a person's name, followed by their employee number and salary. Then comes the next person's record (name, number, salary), etc. If it retrieves them from disk, does the program read the first record correctly? The last record? How does the program mark the end of a record or the beginning of the next? Does everything fit in this format? What if you have two employee numbers?

HARDWARE-RELATED BOUNDARIES

If a mainframe can handle up to 100 terminals, what happens when you plug in the 99th, 100th, and 101st? What if you get 100 people to log on at the same time?

What happens when the disk is full? If a directory can hold 128 files, what happens when you try to save the 127th, 128th, and 129th? If your printer has a large input buffer, what happens if your program fills it but has more data to send? What happens when the printer runs out of paper or the ribbon runs out?

INVISIBLE BOUNDARIES

Not all boundary conditions are visible from the outside. For example, a subroutine might approximate the value of a function, using one approximation formula when the function argument is less than 100 and a different approximation if the argument is 100 or greater. The first formula might be incalculable (e.g., divide by zero) when the function argument is 100 and its values might make no sense when arguments are greater than 100. 100 is clearly a boundary but you might never realize it.

CALCULATION ERRORS

The program calculates a number and gets the wrong result. This can happen for one of three types of reasons:

- Bad logic: There can be a typing error, like A-A instead of A+A. Or the programmer might break a complex expression into a set of simpler ones, but get the simplification wrong. Or he might use an incorrect formula, or one inapplicable to the data at hand. This third case is a design error. The code does what the programmer intendedóit's his conception of what the code should do that is wrong.
- Bad arithmetic: There might be an error in the coding of a basic function, such as addition, multiplication, or exponentiation. The error might show up whenever the function is used (2 + 2 = -5) or it might be restricted to rare special cases. In either case, any program that uses the function can fail.
- Imprecise calculation: If the program uses floating point arithmetic, it loses precision

as it calculates, because of roundoff and truncation errors. After many intermediate errors, it may claim that 2 + 2 works out to -5 even though none of the steps in the program contains a logical error.

This area is huge and this section only begins to scratch its surface. For an introduction to the larger area, read Conte and deBoor (1980) and Knuth (1981). For a second source on topics in Conte and deBoor, try Carnahan, Luther & Wilkes (1969).

OUTDATED CONSTANTS

Numbers are sometimes used directly in a program. The computer might be able to connect to a maximum of 64 terminals. The length of the configuration file might be 706 bytes. The first two digits in the year are 19 (as in 1987). When these values change, the program has to change too. Often, they are changed in a few places but not everywhere. Any calculations based on the old values are now out of date, and thus wrong.

CALCULATION ERRORS

Some errors are as simple as typing a minus sign instead of a plus, or subtracting B from A instead of A from B. These usually show up easily if you exercise reasonable care in testing. If the program asks for input data, then prints a number calculated from these, do the same calculation yourself. Does your number match the computer's?

IMPOSSIBLE PARENTHESES

(A + (B + C) * (D + (A / C - B E / (B + (F + 18 / (A - F))))))

Formulas with many parentheses are hard to understand. It's easy to get one wrong, when writing the code in the first place and when trying to change it later.

WRONG ORDER OF OPERATORS

The program will evaluate an expression in a certain order, but it might be a different order than the programmer expected. For example, if ** represents exponentiation, so 5 ** 3 is 5 cubed, is 2 * 5 ** 3 equal to 1000 (10 cubed) or 250 (twice 5 cubed)?

BAD UNDERLYING FUNCTION

Commercially supplied programs and languages usually do the most basic functions, such as adding and multiplying, correctly. Of course, if your development group has written their own, these functions are as suspect as everything else. Slightly fancier functions, like exponentiation, sine, cosine, and hyperbolics, are not necessarily as trustworthy. Errors of this class may be deliberate. Some programmers use inaccurate approximation formulas because they evaluate quickly or are compact and easy to code.

OVERFLOW AND UNDERFLOW

An overflow condition occurs when the result of a numerical calculation is too large for the program to handle. For example, suppose the program stores all numbers in fixed point format, with one byte per number. It works with numbers from 0 to 255. It can't add 255 + 255 because the result is too large to fit in one byte. Overflows also occur in floating point arithmetic, when the exponent is too large.

Underflows occur only in floating point calculations. In floating point, a number is represented by a pair of values, one for the exponent, the other for a fraction. For example, 255 is 0.255 times 103. 255,000 is 0.255 times 106. The exponent changes, but the fractional part (0.255) is the same in both cases. Now, suppose the program allocates a byte for the exponent, and stores values of 0 to 255. What happens if the exponent is -1 (0.255 * 10-1 is 0.0255)? This is too small to be stored (because the smallest exponent we can store in this scheme is 0), so we have an underflow. Underflows are usually converted to 0 (0.255 * 10-1 becomes 0), without an error message. This is usually appropriate, but it can lead to computational errors:

Is 100 * .255 * 10-1 zero or 2.55?

TRUNCATION AND ROUNDOFF ERROR

Suppose the program stores only two digits per number. The number 5.19 has three digits. If the program truncates (drops) the 9, it stores 5.1. Instead, it could round 5.19 upwards to 5.2, which is much closer than 5.1.

If a programming language keeps two digits per floating point number, it works in two-digit precision. Calculations in this language would not be accurate. For example, 2.056 is about 74. However, if you round 2.05 to 2.1, the calculated value of 2.056 is nearly 86. If you truncate 2.05 to 2.0, you get 64 instead.

We don't know of any language that uses two-digit precision, but many keep only six digits in floating point calculations. Six-digit calculations are fine for simple computations but they can cause surprisingly large errors in more complicated calculations.

CONFUSION ABOUT THE REPRESENTATION OF THE DATA

The same number can be represented in many different ways; they can be confused with each other. For example, suppose that the program asks you for a number between 0 and 9. You type 1. It might store this 1 in a byte, as fixed point number between 0 and 255. There are 8 bits in this byte: the bit pattern is 0000 0001. Or it might store the ASCII code of the character you typed. The ASCII code for 1 is 49, or 0011 0001 in binary. In both cases, the number fits in one byte. It is easy to get confused later and treat something stored in ASCII format as if it were as a fixed point integer, or vice versa. Your number might also be stored in some other format, such as floating point. Again, confusion between formats is possible. Some programming languages straighten this out themselves, others issue warnings during

compilation, and others just let you get the wrong answer.

INCORRECT CONVERSION FROM ONE DATA REPRESENTATION TO ANOTHER

The program asks for a number between 0 and 9. You type 1. This is a character, and the program receives the ASCII code 49. To convert your input to a number, it should subtract 48 from the code value. It subtracts 49 instead. You entered 1, but the program will treat your response as 0. Whenever a program does its own conversion from one data representation to another, it has plenty of opportunity to go wrong. ASCII to Integer is only one example of a conversion. Conversions are common between ASCII, floating point, integer, character (string), etc.

WRONG FORMULA

Some programs use complicated formulas. It's easy to miscopy one, to read the wrong one from the book, or to make an error when deriving one.

INCORRECT APPROXIMATION

Many formulas for approximating or estimating certain values were developed long before computers. They're great formulas in the sense that they don't require much computation, but lousy otherwise. Unfortunately, they are also traditional. They will keep appearing in text books and programs for years. As a common example, if you are graphing a set of data and want to fit a curve to them that has the form Y = a Xb, it is traditional to take logarithms of everything in sight, estimating a and b by fitting a line to the new function, log Y = log a + b log X. This is easy to program, quick to run, and inaccurate. When you return from logarithms and plot a Xb, the curve fits data on the left of the figure (small values of X) better than data toward the right.

Many programs use bad approximation methods and other incorrect mathematical procedures. They might print impressive output, but it is wrong output. You can't test for these types of problems unless you understand a fair bit of the mathematics yourself. If you are testing a statistical package or other mathematical package, it is essential that you or another tester have a detailed understanding of the functions being programmed.

INITIAL AND LATER STATES

Before you can use a function, the program may have to initialize it. Typical initialization steps include identifying the function's variables, defining their types, allocating memory for them, and setting them to default values (such as 0). The program may have to read a disk file that contains defaults and other configuration information. What happens when the file is not there? Initialization steps might be done when the program is loaded (data defaults can be loaded into memory along with the program), when it is started, when the function is first

called, or each time the function is called.

Initialization needs and strategy vary widely across languages. For example:

- In many languages, a function's local variables keep their values from one call to the next. If a variable is supposed to keep the same value, the function can set the variable's value once and leave it alone thereafter. The function usually has to reset other variables to their starting values.
- In other languages, local variables are erased from memory on exit from the function. Whenever the program calls a function, it must redefine its variables, allocate memory for them, and assign starting values.
- Some languages allow the programmer to specify whether a variable should stay in memory or be erased after each function call.
- Some compilers provide initialization support. The programmer can specify a starting value for a variable; the compiler will make sure that this loads into memory with the program. If the programmer doesn't assign an initial value, the compiler sets it to 0. Other compilers, even for the same language, do not provide this support. The function must set each variable's value the first time it's called. To avoid resetting each variable every time it's called, the function must know whether it's been called before.

Initialization failures usually show up the first time the function is called or the second time, if it doesn't reinitialize variables correctly. Reinitialization failures may be path-dependent. If you reach a function in a "normal" way, it works fine. However, if you take an "abnormal" route, the program might branch into the function at some point after the initialization code. Programmers often treat backing up to modify data or redo calculations as abnormal.

FAILURE TO SET A DATA ITEM TO 0

Since so many compilers, in so many languages, set data to 0 unless you tell them otherwise, many programmers don't bother specifying the starting value of a variable unless it is non-zero. Their coding style fails as soon as they use a compiler that doesn't automatically zero data.

FAILURE TO INITIALIZE A LOOP-CONTROL VARIABLE

A loop-control variable determines how many times the program will run through a loop. For example, a function prints the first 10 lines of a text file. The program stores the number of the line reaches 11, printing stops. Next time, the function must reset LINE to 1 or it will never start printing because LINE's value is 11 already.

FAILURE TO INITIALIZE (OR REINITIALIZE) A POINTER

A pointer variable stores an address, such as the location in memory where a given string starts. The value of the pointer can changeófor example, it might point to the first character

in a string, then be changed to point to the second, the third, and so on. If the programmer forgets to reset a pointer after changing it, it will point to the wrong part of the string or to the wrong string. If subsequent function calls keep changing the pointer without reinitializing it, the pointer may eventually point to code rather than data.

You should suspect pointer errors if you see string fragments or incorrect array elements displayed.

FAILURE TO CLEAR A STRING

A string variable stores a set of characters. Whereas the value of a numeric variable might be 5, a string might have the value Hello, my name is John. Strings can vary in length. You can change the string from Hello, my name is John to the shorter string, Goodbye. Some routines assume that a string is empty (filled with zeroes) before they use it. A routine that writes Goodbye into the first 7 bytes of a string without terminating it with zeroes might yield Goodbye my name is John rather than Goodbye.

FAILURE TO INITIALIZE (OR REINITIALIZE) REGISTERS

Registers are special memory areas usually found on the central processing unit itself. You can manipulate data stored in registers more quickly than those stored in normal memory. Because of this speed advantage, programs constantly use registers for temporary storage. They copy a few variables' values to registers, work with them, and copy the new values in the registers back. It's easy to forget to load the latest data into one of these registers.

FAILURE TO CLEAR A FLAG

Flags are variables that indicate special conditions. A flag can be set (true, on, up, usually 1) or clear (false, off, down, usually 0 or -1). The flag's value is normally clear. It is set as a signal that a routine has failed, that its variables have been initialized, that the result of a calculation was an overflow or an underflow, that you've just pressed a key, etc. Less desirably, but also common, a flag might tell a routine that it was called from one place in the program rather than another, or that it should perform one type of calculation rather than another.

The flag must be kept current. For example, a routine should clear its error flag each time it's called, returning with a set flag only when it fails. A routine should clear its data-initialized flag whenever any of its variables change from default values that should be changed back the next time the routine is used. Some programs can set or clear the same flag in a dozen different places. It's hard to tell whether the flag's value is current or not.

DATA WERE SUPPOSED TO BE INITIALIZED ELSEWHERE

A function may not initialize all of its data. For example, variables shared by different functions might be initialized together. Suppose that some functions are listed in the same menu, and their shared variables are initialized whenever that menu is displayed. This works

as long as there's no other path to any of these functions, but can you reach one by a back door? Is any function an option on another menu? Might the program call it as part of another function's error recovery?

FAILURE TO REINITIALIZE

Programmers may forget to make sure that a function's variables have the right values the second time around. Simple forgetfulness is especially common when dealing with languages that automatically initialize variables to 0. The programmer didn't have to set it to 0 for the first pass through the function. Why should she think about setting it back to 0 later?

The programmer might also fail to reinitialize a function's data when it's reached "by the back door," especially when you try to back up to change data. Imagine entering data on a form, displayed onscreen. The program initializes all relevant variables when it paints the form. You enter the wrong number, notice it after entering a few other values, move the cursor back, and fix it. Any calculations based on this number now have to be redone. Will the variables in those calculation sections be reinitialized? Moving backward in the program is risky as far as variable reinitialization is concerned. The risk is much higher if the programmer uses GOTOs to move back to lines in the middle of a block of code, rather than at the start of it.

ASSUMPTION THAT DATA WERE NOT REINITIALIZED

Some programs initialize the same variables repeatedly, before they could have changed. This is harmless, except for the waste of computer time.

CONFUSION BETWEEN STATIC AND DYNAMIC STORAGE

A local variable is called dynamic or automatic if it is erased from memory when the function that owns it exits. Each time the program calls the function, it has to redefine the variable, allocate memory for it, and assign starting values. A static variable stays in memory and keeps its value across function calls. In some languages, all local variables are dynamic, in others all locals are static, and in some, the programmer gets to choose which variables are static and which dynamic. When both types of variables exist, confusion is easy. The programmer might forget to reinitialize a static variable because she thinks it's dynamic, and so doesn't need initialization. Similarly, she might forget to update the value of an automatic variable because she forgets that it doesn't keep its value across function calls.

DATA MODIFICATION BY SIDE-EFFECT

After initialization, a routine might use a variable without changing it. The programmer might consider reinitialization unnecessary, since the variable doesn't change, but even if she intends the variable to be local to that routine, the language she's writing in might not recognize the concept of local variables. Any other part of the program can change this variable; after much maintenance, any other part of the program might.

INCORRECT INITIALIZATION
The programmer might assign the wrong value to a variable, or declare it integer instead of floating point, static instead of dynamic, global instead of local. Most of these errors are caught by the compiler, long before you see the program.

RELIANCE ON TOOLS THE CUSTOMER MAY NOT HAVE OR UNDERSTAND
This is rare but it happensóit's surprisingly easy to miss. The programmer expects you to use some other program to modify this one or to set up some aspects of the program's environment. You must do this the first time you use the program, to get its memory limits or whatever into the right initial state. Thereafter, as the tester, you may forget that you ever did it.

CONTROL FLOW ERRORS
The control flow of a program describes what it will do next, under what circumstances. A control flow error occurs when the program does the wrong thing next. Extreme control flow errors stop the program or cause it to run amok. Many simple errors lead to spectacular misbehavior.

PROGRAM RUNS AMOK
The program displays garbage onscreen, saves garbage to disk, starts printing forever, or goes to some otherwise totally inappropriate routine. Eventually, it may stop dead. Whatever the exact behavior, the program's actions are out of your control. These are the most spectacular bugs, and are usually the easiest to find and fix.

From the outside, these bugs can look the same. They all make the program go out of control. The descriptions below are examples of the causes of programs running amok. You would not test specifically for one of these errors unless you knew something about the programming language, the programmer's style, or the internal design.

GOTO somewhere
GOTO transfers control to another part of the program. The program jumps to the specified routine, but this is obviously the wrong place. The program may lock, the screen display may be inappropriate, etc.

The GOTO command is unfashionable. The structured programming movement is centered on a belief that GOTO encourages sloppy thinking and coding (Yourdon, 1976).

Errors involving GOTO are especially likely when:

- The program branches backward, going somewhere it's been before. For example, the

GOTO may jump to a point just past validity checking or initialization of data or devices.
- The GOTO is indirect, going to an address stored in a variable. When the variable's value changes, the GOTO takes the program somewhere else. It's hard to tell, when reading the code, whether the variable has the right value at the right time.

Come-from logic errors A routine uses come-from logic if it changes what it does based on what routine called or jumped to it. Errors arise when the routine fails to correctly identify what called it or does the wrong thing after correctly identifying the caller. Calling routines often set flags or other variables to identify themselves, but a few different routines may use the same flag to mean different things, some may reset a flag when they're done with it while others don't, and some may give it values that the called routine doesn't expect. Come-from logic is fragile, and particularly prone to failure during maintenance programming.

Problems in table-driven programs
A table-driven program uses a table (array) of addresses. Depending on the value of some variable(s), the program selects a table entry and jumps to the memory address stored there. The table may be a data file read from disk that can be changed without recompiling the program. Table-driven programming can make code easier to maintain, but it has risks:

- The numbers in the table might be wrong, especially if they were entered by hand. These incorrect addresses could send the program anywhere.
- If the table is long, it is easy to supply the wrong entry for a given case, and easy to miss this when desk-checking the code.
- Suppose the table has five entries, and the program selects one of the five based on the value of a state variable. What if the variable can take on six values? Where does the program go in the sixth case?
- It's easy to forget to update a jump table when modifying the code.

Executing data
You can't tell from a byte's contents whether it holds a character, part of a number, part of a memory location, or a program instruction. The program keeps these different types of information in different places in memory to keep straight which byte holds what type of data. If the program interprets data as instructions, it will try to execute them and will probably lock. It may print odd things on the screen first. Some computers detect execution of "impossible" commands and stop the program with an error message (usually a hexadecimal message flagging program termination or reference to an illegal machine code.)

The program will treat data as if they were instructions under two conditions:

- (a) Data are copied into a memory area reserved for code. The code is overwritten. Examples of how to do this:

    - Pointers are variables which store memory addresses. A pointer might hold the starting address of an array; the programmer could put a value in the fourth element of the array by saying store it in the fourth location after the address stored in this pointer. If the address in the pointer is wrong, the data go to the wrong place. If the address is in the code space, the new data overwrite the program.
    - Some languages don't check array limits. Suppose you have an array MYARRAY, with three elements, MYARRAY[1], MYARRAY[2], and MYARRAY[3]. What happens if the program tries to store a value in MYARRAY[2044]? If the language doesn't catch this error, the data will be stored in the spot that would have been MYARRAY[2044] if that MYARRAY element existed. This memory location is a few thousand bytes past the end address of MYARRAY. It might be reserved for code, data, or hardware I/O, but not for MYARRAY.
- (b) The program jumps to an area of memory that is reserved for data, and treats it like an area containing code.
    - A bad table entry in a table-driven program can lead the program to jump into a data area.
    - Some computers divide memory into segments. The computer interprets anything in a code segment as instructions, and anything in a data segment as numbers or characters. If the program misstates a segment's starting address, what the computer interprets as a code segment will probably be a combination of code and data.

Jumping to a routine that isn't resident

To save room, computers may swap pieces of large programs in and out of memory. These pieces are called overlays: when one is in memory, the others aren't. When another is needed, the computer reads it from disk and stores it in the same area of memory used by the first overlay. When routines in the first overlay are again needed, they are again read into the shared area of memory. The routine in memory right now is resident in memory.

Before using a routine that is part of an overlay, the program must check that the right overlay is resident. Otherwise, when it jumps to what should be the starting address of the routine, it may be jumping into the middle of some other routine.

Overlays can also cause performance problems. The programmer might ensure that a routine is resident by always loading it from disk before jumping to it. This wastes a lot of computer time if he calls the routine many times. The program will also waste time if it alternates between routines that are part of two different overlays. This is called thrashing: the program loads the first overlay, executes the first routine, then overwrites it with the second overlay to execute the second routine, reloads the first overlay, etc. It spends most of its time loading overlays rather than getting work done.

Re-entrance
A re-entrant program can be used concurrently by two or more processes. A re-entrant subroutine can call itself or be called by any other routine while it's executing. Some languages don't support re-entrant subroutine calls: if a routine tries to call itself, the program crashes. Even if the language allows re-entrance, a given program or routine might not be. If a routine is serving two processes, how does it keep its data separate, so that what it does for one process doesn't corrupt what it does for the other?

Variables contain embedded command names Some language dialects ignore spaces. A phrase like PRINTMYNAME would be interpreted by the language as PRINT MYNAME. The program would attempt to print the value of variable MYNAME. This is an error if the user was trying to define a variable named PRINTMYNAME. This type of error is usually caught by the programmer, but occasional ones do survive.

Wrong returning state assumed
Imagine a subroutine that's supposed to set a device's baud rate. The program calls this routine and assumes that the routine did its job successfully. It starts transmitting through the device as soon as possible. This time, the routine failed. The transmission fails and the program hangs waiting for acknowledgment of the data. As another example, suppose a routine usually scales the data passed to it, passing back numbers that lie between 1 and 10. Under exceptional circumstances, the routine will scale from 0 to 10 instead. Because the calling program assumes that it will never receive a 0, it crashes on a divide by 0 error.

Exception-handling based exits
Suppose that a routine designed to calculate square roots sets an error flag but does no computations when asked to take the square root of a negative number. The idea behind error flags is that the calling program can decide how to deal with the problem. One might print an error message, another display a help screen, and a third might send the number to a slower routine built for complex numbers. Subroutines that flag and reject exceptional conditions can be used under more conditions. However, each time one is called, the caller must check that it did what the programmer expected it to do. If the exit-producing conditions are rare, he may miss them. During testing, they may show up as "irreproducible" bugs.

Return to wrong place
The key difference between a subroutine and a GOTO is that when the subroutine ends, it returns to the part of the program that called it, whereas GOTO never returns. Occasionally, a subroutine can return to the wrong place in a program. The next few sections are examples.

Corrupted stack
When a subroutine finishes, program control returns to the command following the call to the subroutine. The address of that command is stored in a data structure called a stack. The top of the stack holds the address most recently pushed onto it. The subroutine returns to the

address stored at the top of the stack. If the stack is only used to hold return addresses, it is called a Call/Return Stack. Most stacks are also used as a temporary spot to stash data.

If a subroutine puts data on the stack and doesn't remove them before finishing, the computer will treat the number(s) at the top of the stack as a return address. The subroutine might "return" anywhere in memory.

Stack under/overflow
The stack might only be able to hold 16, 32, 64, or 128 addresses. Imagine a stack that can only hold 2 return addresses. When the program calls Subroutine 1, it stores a return address on the stack. When Subroutine 1 calls Subroutine 2, another return address goes on the stack. When Routine 2 ends, control goes back to Routine 1, and when Routine 1 ends, control returns to the main body of the program.

What if Subroutine 2 calls Subroutine 3? The stack is storing 2 return addresses already, so it cannot also hold the return address for Subroutine 3. This is a stack overflow condition. Programs (or central processing chips) often compound a stack overflow problem by replacing the oldest stored return addresses with the new one. The program will now return to Subroutine 2 when Subroutine 3 is done. From Subroutine 2 it returns to Subroutine 1. From Subroutine 1 it returns to...???...there is no return address for Subroutine 1. This is a stack underflow.

GOTO rather than RETURN from a subroutine
Subroutine 1 calls routine 2. Routine 2 GOTOs back to 1, rather than returning normally. The return address from routine 2 to 1 is still on the stack. When subroutine 1 finishes, the program will return to the address stored on the stack, which takes it back to subroutine 1. This is rarely intentional.

To avoid this error, subroutine 2 might POP (remove) its return address from the stack when it does its GOTO back to routine 1. Used incorrectly, this can cause stack underflows, returns to the wrong calling routine, and attempts to return to data values stored on the stack with the return addresses.

Interrupts
An interrupt is a special signal that causes the computer to stop the program in progress and branch to an interrupt handling routine. Later, the program restarts from where it was interrupted. Input/output events, including signals from the clock that a specified interval of time has passed, are typical causes of interrupts.

Wrong interrupt vector
When an interrupt signal is generated, the computer has to find the interrupt handling routine, then branch to it. The address of the interrupt handler is stored in a dedicated

location in memory. The computer jumps to the address stored in that location. If the computer can distinguish between several different types of interrupts, it finds a given interrupt's handler in a list of addresses stored in a dedicated section of memory. This list is called the interrupt vector.

If wrong addresses are stored in the interrupt vector, any error might be possible in response to an interrupt-generating event. If the addresses are merely out of order, the program is less likely to run amok but it might try to echo characters onscreen in response to a clock signal, or treat keyboard inputs as if they flagged time-outs.

Failure to restore or update interrupt vector
A program can change the interrupt vector by writing new addresses into the appropriate memory locations. If a module temporarily changes the interrupt vector, it might not restore the old address list on exit. Another might fail to make a permanent (or temporary) change to the vector. In either case, the computer will branch to the wrong place after the next interrupt.

Failure to block or unblock interrupts
Programs can block most interrupts, instructing the computer to ignore blockable interrupts. For example, it's traditional to block interrupts just before starting to write data to a disk and to unblock immediately after output to the disk is complete. This prevents many data transmission errors.

Invalid restart after an interrupt
The program is interrupted, then restarted. In some systems, at restart time, the program gets a message or other indication that it was interrupted. The message usually identifies the type of interrupting event (keyboard I/O, time-out, modem I/O, etc.). This is useful. For example, if a program knows it was interrupted, it can repaint the screen with information it was showing before the interrupt. The programmer might easily specify the wrong action or a branch to the wrong location in response to a signal that a certain type of interrupt has been executed. Programmers are as unlikely to catch these errors as error-handling errors.

PROGRAM STOPS
Some languages will stop a program when certain types of errors are detected. Some programs aren't designed to stop, nor are the languages designed to stop them, but they do anyway.

Not all halts are control flow errors. If the program code says "If this happens, halt," the program is supposed to stop. It is a user interface error, but not a control flow error, if this program stops unexpectedly, without a message.

Dead crash

In a dead crash, the computer stops responding to keyboard input, stops printing, and leaves lights on or off (but doesn't change them). It usually locks without issuing any warnings that it's about to crash. The only way to regain control is to turn off the machine or press the reset key.

Dead crashes are usually due to infinite loops. One common loop keeps looking for acknowledgment or data from another device (printer, another computer, disk, etc.). If the program missed the acknowledgment, or never gets one, it may stay in this wait loop forever.

Syntax errors reported at run-time
An interpreted language may not check syntax until run-time. When the language finds a command that it can't interpret, it prints an error message and stops the program. Any line of code that the programmer didn't test may have a syntax error.

Waits for impossible condition, or combination of conditions
The program stops (usually a dead crash) waiting for an event that cannot occur. Common examples:

- I/O failure: The computer sends data to a broken output device, then waits forever for the device to acknowledge receipt. A similar problem arises between processes in a multi-processing system. One process sends a request or data to another, then waits forever for a response that never arrives.
- Deadly embrace: This is a classic multi-processing problem. Two programs run simultaneously. Each needs the same two resources (say, a printer and extra memory for a printer buffer). Each grabs one resource, then waits forever for the other program to finish with the other resource.
- Simple logic errors: For example, a program is supposed to wait for a number between 1 and 5, discarding all other input. However, the code testing the input reads: IF INPUT > 5 AND INPUT < 1. No number can meet this condition so the program waits forever.

Similarly, in multi-processing systems, one process may wait forever for another to send it an impossible value.

Wrong user or process priority
A computer that runs many programs at once switches between them. It runs one program for a while, then switches to a second, to a third, eventually returning to the first. Multi-processing systems run smoothly because a scheduling program switches back to programs when events like keyboard input happen or when a program has been suspended for too long.

If two programs have been waiting equally long to run, or if the same type of event happens to trigger each, the scheduler must decide which program to run first. It uses a priority

system: priorities might be assigned to users or programs. The program being run by a higher priority user will run first.

Some programs run at such low priorities, they may be suspended for hours before being restarted. This may be appropriate. In other cases, priorities were incorrectly assigned or interpreted. Less extreme priority errors are more common but harder to detect unless they trigger race conditions.

LOOPS

There are many ways to code a loop, but they all have some things in common. Here's one example:

```
1 SET LOOP_CONTROL = 1
2 REPEAT
3               SET VAR = 5
4               PRINT VAR * LOOP_CONTROL
5               SET LOOP_CONTROL = LOOP_CONTROL + 1
6 UNTIL LOOP_CONTROL > 5
7 PRINT VAR
```

The program sets LOOP_CONTROL to 1, sets VAR to 5, prints the product of VAR and LOOP_CONTROL, adds 1 to LOOP_CONTROL then checks whether LOOP_CONTROL is greater than 5. Since LOOP_CONTROL is only 2, it repeats the code inside the loop (lines 3, 4, and 5). The loop keeps repeating until LOOP_CONTROL reaches 6. Then the program executes the next command after the loop, printing the value of VAR.

LOOP_CONTROL is called the loop control variable. Its value determines how many times the loop is executed. If the expression written after the UNTIL is complex, involving many different variables, it is a loop control expression, rather than a loop control variable. The same types of errors arise in both cases.

Infinite loop

If the condition that terminates the loop is never met, the program will loop forever. Modify the example so that it loops until LOOP_CONTROL was less than 0 (never happens). It will loop forever.

Wrong starting value for the loop control variable

Suppose that, later in the program, there is a GOTO to the start of the loop at line 2. LOOP_CONTROL could have any value. It probably isn't 1. If the programmer expects this loop to repeat five times (as it would if the GOTO was to line 1), he is in for a surprise.

Accidental change of the loop control variable

In the example, the value of LOOP_CONTROL changed inside the loop. A bigger loop might change LOOP_CONTROL in more than one place (especially if it calls a subroutine that uses LOOP_CONTROL), and the program might repeat the loop more or less often than the programmer expects.

Wrong criterion for ending the loop
Perhaps the loop should end when LOOP_CONTROL > 5 rather than when LOOP_CONTROL >_ 5. This is a common mistake. And, if the ending criterion is more complex, it is more prone to error.

Commands that do or don't belong inside the loop
In the example, SET VAR = 5 is inside the loop. The value of VAR doesn't change inside the loop, so VAR is still 5 the second, third, fourth, and fifth times the loop executes. Resetting VAR to 5 each time is wasteful. Some loops repeat thousands of times: unnecessary repetition within them is significant.

Alternatively, suppose VAR did change inside the loop. If the programmer wants VAR to start at 5 each time the loop repeats, he has to say SET VAR = 5 at the head of the loop.

Improper loop nesting
One loop can be nested (completely included) inside another. It is not possible (without error) for one loop to start inside another but to end outside of it.

IF, THEN, ELSE, OR MAYBE NOT
An IF statement has the form:

```
IF This_Condition IS TRUE
     THEN DO Something
     ELSE DO Something_Else
```

For example:

```
IF VAR > 5
     THEN SET VAR_2 = 20
     ELSE SET VAR_2 = 10
```

The THEN clause (SET VAR_2 = 20) is only executed if the condition (VAR > 5) is met. If the condition is not met, the ELSE clause (SET VAR_2 = 10) is executed. Some IF statements only specify what to do if the condition is met. They don't include an ELSE clause. If the condition is not met (VAR =< 5) the program skips the THEN clause and moves on to the next line of code.

Wrong inequalities (e.g., > instead of >_)
The tested condition (VAR > 5) might be incorrect, or incorrectly stated. Programmers often forget to consider the case in which the two variables are equal.

Comparison sometimes yields wrong result
The condition tested by the IF is usually the right one, but not always. Suppose the programmer wants to test whether three variables are the same. He might write IF (VAR + VAR_2 + VAR_3) / 3 = VAR.

If VAR, VAR_2, and VAR_3 are the same, the average of them will have the same value as any one of them. Further, for almost all values, if VAR, VAR_2, and VAR_3 are not the same, their average will not equal VAR. But suppose that VAR is 2, VAR_2 is 1, and VAR_3 is 3. (VAR+VAR_2+ VAR_3) / 3 = VAR, but VAR, VAR_2, and VAR_3 aren't equal. Shortcuts like this that try to combine a few comparisons into one regularly go awry.

Not equal versus equal when there are three cases
The three-case problem often comes up during maintenance programming. The initial code may have restricted VAR's values to 0 and 1, but later changes allow it to be 2 as well. In the original program, an IF statement for VAR = 0 was fine. The THEN clause covered VAR = 0, and, since VAR could only be 0 or 1, the ELSE clause said what to do when VAR = 1. Now that VAR can also be 2, the ELSE clause is probably wrong.

It's risky to compare a variable to only one value (like VAR = 0), leaving all the others to the same ELSE clause. There are so many other possible values: some may arise as originally unanticipated special cases.

Testing floating point values for equality
Floating point calculations are subject to truncation and round off errors. For example, rather than being exactly zero, a variable's value might be 0.000000008 because of small computational errors. This is close, but it wouldn't pass a test of equality (IF VAR = 0).

Confusing inclusive and exclusive OR
Many IF statements test whether one of a group of conditions is true (IF A OR B is true, THEN ...) Unfortunately, "or" is ambiguous:

- inclusive or: satisfied if A is true, B is true or both A and B are true
- exclusive or: satisfied if A is true or B is true, but not if A and B are both true

Incorrectly negating a logical expression
IF statements sometimes take the form, IF A is NOT true, THEN. Programmers often carry out the negation incorrectly or don't think through what the negation means. For example, IF NOT (A or B) THEN ... means IF A is false AND B is false, THEN.... The THEN clause

will not be taken if A or B is true, even if the other is false.

Assignment-equal instead of test-equal
In the C language, if (VAR = 5) means SET VAR = 5, then test whether it's nonzero. Programmers often write this instead of if (VAR == 5), which means what some people think if (VAR = 5) should mean.

Commands belong inside the THEN or ELSE clause
Here's a simple example of this type of error:

```
IF
VAR = VAR_2
                THEN SET VAR_2 = 10
                SET VAR_2 = 20
```

Clearly, SET VAR_2 = 20 belongs inside an ELSE clause. As it is now, VAR_2 is always set to 20. Setting VAR_2 to 10 first, when VAR = VAR_2, has no effect.

Commands that don't belong inside either clause
Sometimes the programmer will include a command inside a THEN or an ELSE clause that should always be executed (i.e., in both cases). If he repeats the command inside both clauses, he wastes code space, but this usually doesn't matter. If he includes it only inside one clause, it will be missed whenever the other clause (ELSE or THEN) is executed.

Failure to test a flag
For example, the program calls a subroutine, which is supposed to assign a value to a variable. The subroutine fails, sets its error flag, and leaves the variable alone. The program doesn't check the error flag. Instead, it does its usual IF test on the variable. Whatever code the program executes from here is wrong, or is right only by luck. The value stored in the variable is junk. That's what the subroutine means when it sets the flag.

Failure to clear a flag
A subroutine set its error flag the last time it was called. The flag is still set. This time the subroutine does its task and leaves the error flag alone. The error flag is still set. The program will believe this error flag, ignore the subroutine's output, and do error recovery instead.

MULTIPLE CASES
An IF statement considers only two cases: an expression is either true or false. Commands like CASE, SWITCH, SELECT, and computed GOTO are used when a variable might have many different values and the programmer wants to do one of many different things depending on the value.

The typical command of this type is equivalent to this:

IF VAR is 1 do TASK-1
IF VAR is 2 do TASK-2
IF VAR is 3 do TASK-3
IF VAR is anything else, do DEFAULT-TASK

If there is no default case, the program falls through to the commands following this multiple-choice block.

Missing default
A programmer who thinks VAR can only take on the values listed may not write a default case. Because of a bug or later modifications to the code, VAR can take on other values. A default case could catch these, and print any unexpected value of VAR.

Wrong default
Suppose the programmer expects VAR to have only four possible values. He explicitly deals with the first three possibilities, and buries the other one as the "default." Will this default be correct for VAR's unanticipated fifth and sixth values?

Missing cases
VAR can take on five possible values but the programmer forgot to write a CASE statement covering the fifth case.

Case should be subdivided
Some cases cover too much: perhaps one case covers all values of VAR below 30, but the program should do one thing if VAR is below 15 and something else for larger values. The most common example of this problem is the default case. The programmer doesn't think it matters what happens if VAR has certain values, so he covers them all with the default.

Overlapping cases
The CASE statements are equivalent to this:

IF VAR > 5 then do TASK_1
IF VAR > 7 then do TASK_2
etc.

The first and second cases overlap. If VAR is 9, it fits in both cases. Which should be executed? The first task is the usual choice. Sometimes both are. Sometimes the second one is the correct choice.

Invalid or impossible cases
The program executes TASK_16 only if VAR < 6 AND VAR > 18. TASK_16 can never run because VAR can't meet this condition. Similarly, the program might specify a value that VAR can't reach in practice, even though it's not an impossible number. You won't see this type of problem unless you look at the code, but it wastes code space and may reflect fuzzy thinking.

ERRORS IN HANDLING OR INTERPRETING DATA

Data are passed from one part of a program to another, and from one program to another. In the process, the data might be misinterpreted or corrupted.

PROBLEMS WHEN PASSING DATA BETWEEN ROUTINES

The program calls a subroutine and passes it data, perhaps like so:

DO SUB(VAR_1, VAR_2, VAR_3)

The three variables, VAR_1, VAR_2, and VAR_3 are passed from the program to the subroutine. They are called the subroutine's parameters. The subroutine itself might refer to these variables by different names. The statement at the start of the subroutine definition might look like this:

SUB(INPUT_1, INPUT_2, INPUT_3)

The subroutine receives the first variable in the list passed by the program (VAR_1) and calls it INPUT_1. It calls the second variable in the list (VAR_2) INPUT_2. INPUT_3 is its name for the last variable (VAR_3).

The program's and subroutine's definitions of these variables must match. If VAR_1 is an integer, INPUT_1 should be as well. If VAR_2 is a floating point value for someone's temperature, that's what the subroutine had better expect to find in INPUT_2.

Parameter list variables out of order or missing
If the program says DO SUB(VAR_2, VAR_1, VAR_3), the subroutine will associate INPUT_1 with VAR_2, and INPUT_2 with VAR_1. Programmers routinely type the variable names in the wrong order in these lists.

Missing parameters are less common in many, but not all, languages because their compilers catch this problem.

Data type errors
Suppose the program defines VAR_1 and VAR_2 as two-byte integers but the subroutine defines INPUT_1 and INPUT_2 as one-byte integers. What happens is language dependent, but it would be no surprise if INPUT_1 got the first byte of VAR_1 and INPUT_2 got the second byte.

The data type specifies how the data are stored. Integers, floating points, and character strings are simple examples. Arrays, records (like a record in a database, with fieldsósee Chapter 12) and arrays of records are common examples of slightly more complex data structures. There are also stacks, trees, linked lists, and others. (See Elson, 1975, for descriptions of these.)

Sometimes a mismatch between the structure of data in the calling and called routines is deliberate. The calling program might pass a three-dimensional array which the subroutine treats as a bigger one-dimensional array. The calling routine might pass an array of characters to a subroutine that treats them as an array of numbers. Some languages ban this, but it is standard form in others. As you can imagine, it can lead to a big mess, especially when the variable(s) being reinterpreted are part of a larger list. If the calling and called routines differ in the amount of memory they expect these mismatched variables to use, anything that comes after these in the parameter list might be misread.

Aliases and shifting interpretations of the same area of memory
If two different names refer to the same area of memory at the same time, they are aliases. If VAR_1 and FOO are aliases for each other, then if the program says SET FOO = 20, VAR_1 becomes 20. Some aliases are trickier. Suppose VAR_1 and VAR_2 are both one-byte integers and FOOVAR is a two-byte integer whose first (high order) byte just happens to be VAR_1 and whose second byte is VAR_2. In this case, SET FOOVAR = 20 sets VAR_2 to 20 and VAR_1 to 0.

It is easy to forget an alias. Expect maintenance programmers to miss it. In either case, the programmer will freely change one variable without realizing the effect on the "other." This can cause all sorts of unexpected results, even more so when the aliasing is more complex than two variables with the same name.

Misunderstood data values
The program passes temperature in centigrade to a subroutine which interprets the value as fahrenheit. The subroutine puts a 1 into an error flag to indicate that the flag is clear. It uses -1 to indicate a set flag. The program expects 0 for a clear flag and any other value if the flag is set.

Inadequate error information
The subroutine fails to set an error flag (maybe there is no error flag). Or it does signal an

error but doesn't say enough else for the calling program to decide how to handle it.

Failure to clean up data on exception-handling exit
The subroutine detects an error or a special case and exits quickly. Before it detected the problem, it changed the values of variables passed to it. If possible, it should reset these to their original values before returning to the calling program.

Outdated copies of data
Two processes may keep their own copies of the same data. Two routines within the same process might do the same. When the data change, both (all) copies have to be updated. It's common to find one process or routine working with an outdated copy of the data, because another changed the data without indicating that it did so.

Related variables get out of synch
One variable is usually a multiple of another, but one was changed and the other was not updated. In contrast to the outdated copy problem just described, which is more likely a problem between processes, this one is common within the same routine.

Local setting of global data
Global variables are defined in the main program. Any subroutine can use them óread their values or change them. Subroutines' changes of a global variable are often accidental. The programmer thought that a second variable, local to the subroutine, had this name and that the change would be to that local variable.

Global use of local variables
A variable is local to a subroutine if only that subroutine can use it. Most languages distinguish between global and local variables, but not all. In many BASICs, for example, all variables are global. In these, variables are kept local only by usage: they appear only in one subroutine. Especially if the variables are not carefully named, the programmer might unintentionally refer to a local variable in other places in the program.

Wrong mask in bit field
To save a few bytes and microseconds, some processes pass data in bit fields. Each byte might hold eight variables, with one bit assigned to each. Or, the processes might use the first two bits for one variable, the next three for a second, leaving one bit each for third, fourth, and fifth variables. A mask is a bit pattern that allows the programmer to focus only on the bits of interest. It has 0's in the other bit positions. The program refers to these to zero out the irrelevant bits in the bit field. If the mask has the wrong bits set, the program looks at the wrong "variable."

Wrong value from a table
Data are often organized in tables (arrays or records). Pointer variables indicate where in the

table a value should be stored or retrieved. The program might look in the wrong place (bad pointer) or it might look in the right place but find an incorrect value.

DATA BOUNDARIES
The program may use the wrong starting or ending address of a set of data.

Unterminated null terminated strings
STRING_VAR is a string variable. It can hold the string Hello or the string I am a string variable or a much longer string. The number of characters stored by a string variable isn't fixed, so there must be a way to indicate the end of the string. One approach puts a null character (all bits zero) after the last character. This is called the string terminator. In languages which use null terminators, all string handling routines look for the null. Occasionally, the null character is forgotten, overwritten, or just not copied with the rest of the string. A routine that works with this unterminated string, perhaps copying or printing it, will include the string and everything past what should be the string's end until it reaches a null or the end of the computer's memory. In copying the string to another variable, the routine might fill the new variable's data space plus hundreds of bytes past it, overwriting other variables or code.

Early end of string
STRING_VAR is supposed to hold I am a string variable but a routine that operates on STRING_VAR (perhaps copying or printing it), behaves as if it held I am a str. Perhaps a null character was copied into the middle of the string. If string length is kept in a separate byte (the length byte), perhaps this value was miscalculated or overwritten.

Read/write past end of a data structure, or an element in it
An array is a good enough example of a data structure for this description. The program might miscalculate the length of each element and so err when it tries to read the value of a specific element. In doing so, it might read a memory location well past the end of the array. The routine will probably also overshoot the end of the array if it expects the array to have more elements than it does. This kind of error usually happens when one routine sends the array to another, and their definitions of the data stored in it don't match.

READ OUTSIDE THE LIMITS OF A MESSAGE BUFFER
A buffer is an area of memory used for temporary storage. Messages between processes often include buffers: the message includes a pointer to the start of the buffer and, in effect, says "for more details, read this. " When the process receiving the message is done with it, it "releases" the buffer which becomes "free memory" again, ready for other uses as needed by the operating system.

The receiving routine might get the address or the length of the message buffer wrong. It could start reading memory locations that precede the start of the buffer or it could keep

reading data out of locations past the buffer's end.

Compiler padding to word boundaries

Depending on the computer, a word might be 12 bits, 1, 2, 3, or 4 bytes long, or some other length. A word is a computer's most natural unit of storage. Some compilers "pad" all one-byte variables to make them a word long. For example, a variable whose value was 255 now has the value 00255: the leading zeros are padding. The variable takes more space but keeps the same value. Such a compiler might pad individual variables, individual elements of arrays, or the full array itself, to ensure that variables (or the array) have their first byte at the start of a word. Another compiler for the same language might not do this type of padding. Some routines calculate how many bytes a piece of data should be from the start of a data structure (perhaps the start of a message buffer). Such routines will fail when the programmer switches from a compiler that uses one set of padding rules to another compiler whose rules are different.

Value stack under/overflow

Earlier in the Appendix ("Control flow errors") we described stack problems as they relate to subroutine calls. The programmer might also store data on the stack. A stack that holds data only, noóreturn addressesóis a value stack.

Suppose a stack can hold 256 bytes and the programmer tries to store 300 bytes on it. The stack overflows: the last 256 bytes stored are usually kept, and the first 44 values lost, overwritten by the others. When the program tries to retrieve these data from the stack, it can only get the last 256. When it tries to pop the 257th value off the stack (the 44th pushed onto the stack) there is an underflow conditionóthe program is trying to retrieve a value from a stack that is now empty.

Trampling another process' code or data

This is especially common when processes share areas of memory, rather than passing data back and forth using messages. One process loses a null terminator from a string or miscalculates the length of a data structure or just runs amok. It writes junk into areas of memory that it shares with another process, or it writes into areas it can reach even though they should be private to that other process.

MESSAGING PROBLEMS

The safest way for two processes to communicate is via messages. If they pass data through shared memory areas instead, a bug in one process can trash data used by both, no matter how defensively the other process was written. The most prevalent problems arising out of messaging architectures are race conditions, which are discussed in the next section. There are also errors in sending and receiving the data in a message.

Messages sent to wrong process or port

A message can go to the wrong place. Even if it goes to the right process, that process may expect messages from this one to arrive only at certain ports (think of ports as virtual receiving areas). Even a message that goes to the right process and port may carry an invalid ID (such as the name of the communication protocol in use between the processes). In any of these cases, the message will be rejected.

Failure to validate an incoming message
A process must check messages that it receives to make sure that they are intended for it, that they contain the right identifiers, etc. The process that sent this message may have sent it to the wrong place or it may be running amok. It is up to the receiving process to ensure that it accepts and acts on no garbage.

Lost or out of synch messages
One process may send many messages to another, in a predictable order. Sometimes, however, a process will send MESSAGE_2 before MESSAGE_1. It might send a request to write something to a disk file before sending a message that names the file and requests that it be opened. Messages can get out of order for hundreds of reasons; not all of them are bugs. The receiving program should be able to cope with this, perhaps by saving MESSAGE_2 until it gets MESSAGE_1, or by telling the other process that it discarded MESSAGE_2 because it came out of order.

Mismatching state information is a common symptom of a failure to cope with badly ordered messages. In the state table of one process, the disk file is open, the printer is initialized, the phone is offhook, etc. According to the other process, the disk file has not been opened, the printer is not ready, and the phone is on hook. It doesn't matter which process is right. The mismatch causes all sorts of confusion.

Message sent to only N of N+1 processes
Suppose that many (N+1) processes keep private copies of the same data, and update their local databases when they get a message instructing them to do so. Or suppose that many people are at their terminals, a separate control and communications process is assigned to each terminal, and an urgent message is sent out, to be printed on each screen, saying that the system will go down in three minutes. Sometimes, one of the processes doesn't get the message. This is usually the most recently activated process or the one most recently coded.

DATA STORAGE CORRUPTION
The data are stored on disk, tape, punch cards, whatever. The process corrupts stored data by putting bad values into these files.

Overwritten changes
Imagine two processes working with the same data. Both read the data from disk at about the same time. One saves some changes. The second doesn't know anything about changes made

by the first. When it saves its changes, it overwrites the data saved by the first process. Some programs use field, record, or file locking to prevent processes from changing fields, records, or files that another process is changing. These locks are not always present, and they don't always work.

Data entry not saved
The program asks for data, which you enter. For some reason, maybe because the file is locked, it doesn't succeed in storing your entries on disk.

Too much data for receiving process to handle
The receiving process might not be able to cope with messages beyond a certain length, or with more than so many messages per minute. It might discard the excess, crash, or print error messages. What it won't do is process the excess messages successfully.

Overwriting a file after an error exit or user abort
You enter data but try to stop the program before it saves them. It saves the new (bad) data first, then stops.

RACE CONDITIONS
In the classic race, there are two possible events, call them EVENT_A and EVENT_B. Both events will happen. The issue is which comes first. EVENT_A almost always precedes EVENT_B. There are logical grounds for expecting EVENT_A to precede EVENT_B. However, under rare and restricted conditions, EVENT_B can "win the race," and occur just before EVENT_A. We have a race condition whenever EVENT_B precedes EVENT_A. We have a race condition bug if the program fails when this happens. Usually the program fails because the programmer didn't anticipate the possibility of EVENT_B preceding EVENT_A, so he didn't write any code to deal with it.

Few testers look for race conditions. If they find an "irreproducible" bug, few think about timing issues (races) when trying to reproduce it. Many people find timing issues hard to conceptualize or hard to understand. We provide more than our usual amount of detail in the examples below, hoping that this will make the overall concept easier to understand.

RACES IN UPDATING DATA
Imagine that one routine reads a credit card balance from the disk, adds the amount of the card holder's latest purchase, and writes the new balance back to the disk. A second routine reads the same balance, subtracts the latest payment, and saves the result to disk. A third routine adds foreign currency transactions. Each of these routines can run concurrently. Each runs quickly, and there are many different card holders, so the following scenario is most unlikely:

A credit card has a balance of $1000. The card holder has just made a purchase for $100 and a payment of $500. The correct balance is thus $600. However, the first routine reads the $1000 balance from the disk. While it adds the $100 purchase, the second routine reads the same card holder's balance (still $1000). Then the first routine stores the new balance ($1100) to disk. The second routine subtracts the $500 payment amount from the $1000 balance that it read from the disk. It saves the new balance ($500) to disk. The $100 addition made by the first routine has been completely lost because the second routine read the balance before the first routine had finished updating it.

This is a race condition: it should almost never happen that the second routine will read the balance after the first routine has started changing it but before the first routine finishes. However, it can happen and occasionally it will.

ASSUMPTION THAT ONE EVENT OR TASK HAS FINISHED BEFORE ANOTHER BEGINS

The previous and the next sections provide examples of this type of problem.

ASSUMPTION THAT INPUT WON'T OCCUR DURING A BRIEF PROCESSING INTERVAL

You type a character. The editing program you're testing receives it, moves other displayed characters around on the screen so it can display this one at the cursor location, echoes the received character, then looks for your next input. Naturally, since the computer is faster than the finger, the program should get everything done and be ready for the next input long before you're ready to type it. Accordingly, the program doesn't allow for the possibility that other characters will arrive before it's done with this one. However, a fast typist might enter two, three, or more characters before the editor is ready for them. The editor catches the last one typed and misses the others, which were typed while it was in the middle of dealing with the first one.

ASSUMPTION THAT INTERRUPTS WON'T OCCUR DURING A BRIEF INTERVAL

The program is doing time-critical operations, such as:

- writing bits to the right place on a spinning disk or a moving cassette tape
- getting a pen to draw at the right place on a moving sheet of paper
- responding to a message or acknowledging input within a short time period

The programmer realizes that these operations take very little time. Since it's so unlikely for an interrupt-triggering event to happen in this brief interval, why take the time to block interrupts during it? Usually all goes well, but every now and again the program will be interrupted.

Failure to block interrupts was raised earlier ("Program runs amok: Interrupts"). There the focus was on the problems of interrupts. Here the point is one of timing. Even if part of a program is brief, if it lasts long enough that an interrupt-triggering event can happen during this interval, then some day an interrupt-triggering event will happen during the interval.

RESOURCE RACES: THE RESOURCE HAS JUST BECOME UNAVAILABLE

Two processes both need the same printer. The one that takes control of the printer first gets it. The other has to wait. Even though there's a "race" here, this is not a race condition in concurrent systems. Programs are, or should be, written to expect the printer (or other sharable resources) to be temporarily unavailable.

Suppose, though, that one process checks whether a printer is available. If the printer is busy, the program does something else. If the printer is available, the program starts to use it. Since the program knows the printer is available, it doesn't consider the possibility that the printer is unavailable.

Unfortunately, there is a short window of vulnerability between the time that a process checks whether the printer is available and the time it takes the printer over. It takes a little time to examine the variable that says that the printer is free, call the right routine when the printer is available,find the data it's supposed to print, etc. During this short period, a second routine might take over the printer and start printing.

Some programmers would argue that this is a rare event. They're right. The window of vulnerability is so small that it is hard to set up a situation in which the second process can snatch away the printer just before the first one gets back to it. However, these processes maybe run by customers thousands or millions of times. Even unlikely race conditions will occur in use. If the consequences of a race condition bug are severe enough, it must be fixed even it will only happen once per million times that the program is used.

ASSUMPTION THAT A PERSON, DEVICE, OR PROCESS WILL RESPOND QUICKLY

For example, the program puts a message on the screen and waits for a response for a few seconds. If you don't respond during this time-out interval, the program decides that you aren't there and halts. Similarly, another program trying to initialize a printer will wait only so long. The program will report that the printer is unavailable if it doesn't respond by the end of the time-out interval. Programs also impose time-outs while waiting for messages from other processes.

Very short time-out intervals cause races. If you have to press a key within a few tenths of a second after a program displays its message, you will often lose the race, the program will decide that you aren't there, and stop. If it gives you a few seconds, you will usually win the race, but sometimes the time-out interval will end just as you notice and respond to the

message. If the program gives you minutes to respond, it is probably safe to assume that you are not there or are not going to respond. The intervals are different for device, or process responses, but the principle is the same. Some intervals are too short, some are just a little too short, and some are plenty long enough.

If the interval is too short, the programmer has probably anticipated that the program will time out before the person, device or process has had a chance to complete its response. Since this isn't an unusual case, he probably has good recovery code to deal with this. This isn't a classical race condition.

If the interval is just a little too short, the risks are higher. The programmer might believe that if the program doesn't receive a response within the specified period, it will never receive a response. What happens if the response arrives milliseconds after the time-out interval has ended? The program might interpret this as a response to some other message, or it might just crash. This is a classic race because it is unlikely, but not impossible, for the response to occur after the time-out period is over.

OPTIONS OUT OF SYNCH DURING A DISPLAY CHANGE

The computer displays a menu and waits for your response. Triggered by a time-out or by another event (a message or a device input), the program switches to another menu. You press a key just as the program is writing the new menu. Here are two possible errors:

- Even though it's displaying the new options, the program will interpret your keypress as selecting a choice from the old menu if it hasn't yet updated its list of choices associated with keystrokes.
- Even though it's displaying the old options, the program will interpret your keypress as selecting a choice from the new menu, because it updated its key-to-option list before displaying the new values onscreen.

This is a real-user problem. Experienced users of a program know when the menu will change. Many make their responses as soon as possible, so they will frequently press a key just as the screen is being repainted.

TASK STARTS BEFORE ITS PREREQUISITES ARE MET

The program starts sending data to the printer just before the printer is ready, starts trying to fill memory with data just before it's assigned a memory area to work with, etc. Perhaps the program is supposed to wait until it receives a specific message from another process before starting the task. But based on other information (such as other messages), the programmer knows that the trigger message will come soon. He starts this task early, to improve performance. The prerequisite tasks are usually completed in time, but occasionally the program is just barely ahead of them.

MESSAGES CROSS OR DON'T ARRIVE IN THE ORDER SENT
Suppose you have $1000 in a bank account, and you try to do three things, in order:

(a) withdraw $1000
(b) deposit $500
(c) withdraw $100

The first withdrawal goes through. The deposit is accepted, but when you try withdraw the $100, you're told that your account's balance is zero, not $500. For some reason, your deposit has taken longer to process than your request for a withdrawal.

Problems of this class are common in message-passing systems: some messages are transmitted along circuitous routes, or their contents have to be verified, or for some other reason they don't arrive at their target process or aren't read by it before another message that was sent later. As a result, until the system catches up, you aren't (are) able to do something that you should (shouldn't) be able to do.

The most annoying version of this involves contradictory messages that cross each other's path. One process requests an action of another. The second process sends a message indicating that it can do that task (e.g., gives a receipt for the $500), but then sends a message saying that it can't (your balance is zero). The verification message that you deposited $500 reaches you and the central database at the same time, and just after your request for $100 reaches the database. To the database, it seemed that you asked for $100, then deposited $500, but because you received early verification, it seems to you that the database should have known about the $500 when it rejected the withdrawal.

LOAD CONDITIONS
Programs misbehave when overloaded. A program may fail when working under high volume (lots of work over a long period) or under stress (maximum amount of work all at once). It may fail when it runs out of memory, printers or other "resources." It may fail because it's required to do too much in too little time. All programs have limits. The issues are whether a program can meet its stated limits and how horribly it fails when those limits are exceeded.

Also, some programs create their own load problems, or, in multi-processing situations, make problems for others. They hog computer time or resources or create unnecessary extra work to such an extent that other processes (or themselves later) can't do their tasks.

REQUIRED RESOURCE NOT AVAILABLE
The program tries to use a new device or store more data in memory, but can't. Run separate

tests run for the program's handling of each of the conditions below, and similar tests for any other device you use (plotters, telephones, etc.). The following conditions should be self-explanatory:

- Full disk
- Full disk directory
- Full memory area
- Full print queue
- Full message queue
- Full stack
- Disk not in drive
- Disk drive out of service
- No disk drive
- Printer off line
- Printer out of paper
- Printer out of ribbon
- No printer
- Extended memory not present

DOESN'T RETURN A RESOURCE

Systems may run out of resources because one or a few processes hog them all. Programmers are good at making sure that their programs get the resources they need. They are not as conscientious about returning resources they no longer need. Since the program won't crash (usually) if it hangs on to the printer or a memory buffer for too long, this type of problem seems less urgent.

The following sections are examples to consider when designing test cases.

Doesn't indicate that it's done with a device

For example, a process uses the printer. All other processes have to wait until this one signals that it's done. The process fails to send that signal, and so prevents others from using an unused device.

Doesn't erase old files from mass storage

The program doesn't erase outdated backups and internal-use temporary files. There are limits on how much erasure should be done automatically, but the process doesn't get rid of files that obviously should go.

Doesn't return unused memory

In multi-processing systems, a memory management process can assign segments of memory to processes on a temporary basis (such as data and message buffers). The process is supposed to signal the memory manager when done with a segment. The manager takes

back control of the segment and assigns it for use by other processes as needed. Failure to return buffers, especially message buffers, is extremely common.

Wastes computer time
The process checks for events that are no longer possible or does other things that used to be necessary but now aren't.

NO AVAILABLE LARGE MEMORY AREAS
In a message passing, multi-processing system, a pool of memory is available to be assigned as message buffers to any process. Some buffers are large, others just a few bytes. A large block of memory might be divided into many small buffers. What happens when the program needs a larger block of memory than any of these small ones?

Some memory management programs don't attempt to merge used buffers into the memory pool. Instead, they reuse these old buffers whenever a buffer their size is needed. As a result, there are few large areas of memory in the common pool.

INPUT BUFFER OR QUEUE NOT DEEP ENOUGH
The process loses keystrokes, messages, or other data because too much comes at once and there's nowhere to put it all. When a process receives more individual data items (like keystrokes) than it can immediately handle, it usually stores the extras in an input buffer, reading them from the buffer and dealing with them when it has time. Similarly, it might store packets of information (like messages) in a queue, getting to them one at a time.

If the process' input buffer is 10 characters deep, what happens if you type 11 (or more) characters quickly? Does it signal that the buffer is full? What if the device sending input is a computer, connected to a modem? Does the program tell the sending program to stop for a while?

If the process' message queue is 256 messages deep, what happens when 256 messages are waiting, one is being processed, and the 257th arrives? Is it discarded or is it returned to the sender with an error code signaling that the message queue is full? If a few messages are discarded without notification, what is the most consequential message discarded? What important information has the receiving process lost that the sending process will assume was received?

DOESN'T CLEAR ITEMS FROM QUEUE, BUFFER, OR STACK
Suppose the program receives messages, puts them in a queue, and reads them from the queue when it has time. It can store up to 256 messages in the queue. When it reads a message, the process should remove it from the queue, making room for a new one. However, programmers may forget to remove debugging messages, so the queue fills as soon as the program tries to use the queue for the 257th time.

In more subtle cases (similarly for failures to return buffers), messages are usually but not always removed from the queue. In one special case, the program doesn't discard old messages. It fails on the 257th time that this bug is triggered. It's because of these kinds of errors that you should occasionally test a program for a long time without rebooting it. Make sure that a minor problem, invisible over short periods, doesn't eventually devastate the system. A "long time" is defined in terms of your customers' needs. They will restart a word processor much more often than a telephone system. You might test the word processor for hours before rebooting, the telephones for weeks or months.

LOST MESSAGES

The operating system might lose some messages (presto, vanish). Or the receiving process might, if enough messages arrive at once. If a process is supposed to be able to handle a queue of 256 messages, what happens to the message it's working on, the message at the start of the queue, and the one at the end, when the 256th message arrives? What happens with the 257th, 258th, and 259th messages? Are they returned to the sending process with a "send-me-again-later" notification or are they just thrown away? Does the sending process need to know that the process it sent a message to was too busy to read it?

PERFORMANCE COSTS

When the workload is high, everything slows down. Larger arrays have to be searched, more users or processes have to be served, etc. A program that must respond within a certain time or process so many events per second might fail because it's running under too busy a multi-processing system or because it's trying to juggle too many inputs itself. Other programs, expecting that this one will respond within a short interval, might also fail if this one responds too slowly.

RACE CONDITION WINDOWS EXPAND

As performance gets worse, race conditions become more likely. In the classic race, two events can happen. One almost always precedes the other but sometimes the second will (or will appear to) precede the first by a tiny bit. As you slow the system down, it may take the computer longer to generate or detect the first event. If the second event is an input (keystroke or modem input), the person or machine that generates it may not be affected by the increased load. The second event will happen as quickly as usual even though the processing of the first has slowed. Thus there is more time for the second event to appear to beat the first.

DOESN'T ABBREVIATE UNDER LOAD

Some processes make lots of output. Formatting all this information and sending it to the printer or screen takes lots of computer time. When the computer is operating under heavy load, an output-intensive process should try to send out less. It might express error messages more tersely, or abbreviate system log messages to short codes or send them to a buffer or disk file to be printed later. Only urgent messages should go to the printer or screen

immediately.

Use your judgment before criticizing a program that doesn't abbreviate or run at lower priority. A word processing program that is about to print the agenda for a board meeting that starts in three minutes should neither delete items from the agenda nor wait till tomorrow before printing it.

DOESN'T RECOGNIZE THAT ANOTHER PROCESS ABBREVIATES OUTPUT UNDER LOAD

Imagine a multi-user system in which all programs log failures and other "interesting" events to the system operator's console. Usually, the log messages include a few numbers plus English-language descriptions. Under heavy load, the messages are abbreviated to short numeric codes. No one can understand the codes without looking them up in a book, but at least the messages themselves don't further slow down the system.

Now suppose these logs are stored on disk. At the end of every day (week, month), a maintenance program reads the files and analyzes the system's failures, perhaps running hardware diagnostics in response to some of the messages. This maintenance program will have to be able to cope with the short code abbreviation system, or it will fail whenever it has to read a message that was saved when the system was under heavy load. A dismaying number of analysis programs do not know how to deal with abbreviated output.

LOW PRIORITY TASKS NOT PUT OFF

Under heavy load, any task that doesn't have to be done immediately should be postponed. In a multi-processing system, programs or people are assigned priorities. Those with higher priorities should get a higher share of available machine time than those with lower priorities.

LOW PRIORITY TASKS NEVER DONE

You can put off changing the oil in your car for a while, but eventually it must be done or else. Many low priority tasks are like this: you don't have to do them right away, but they must be done eventually. High priority tasks cannot be allowed to use all of the computer's time under prolonged periods of heavy load. Some time must be given to lower priority tasks.

HARDWARE

Programs send bad data to devices, ignore error codes coming back, try to use devices that aren't there, and so on. Even if the problem is truly due to a hardware failure, there is also a software error if the software doesn't recognize that the hardware is no longer working correctly.

WRONG DEVICE
For example, the program prints data on the screen instead of the printer.

WRONG DEVICE ADDRESS
In many systems, a program writes data to a device by writing them to one or a few addresses in memory. The physical copying of data from these special memory locations to the devices themselves is taken care of by hardware. The program might write the data to the wrong memory location(s).

DEVICE UNAVAILABLE
See "Required resource not available" earlier in this Appendix.

DEVICE RETURNED TO WRONG TYPE OF POOL
For example, in a multi-processing system there might be many dot matrix printers and many laser printers. A program uses a dot matrix printer, then signals that it's done with it. On returning it to the pool of available devices, the resource manager erroneously marks it as an available laser printer.

DEVICE USE FORBIDDEN TO CALLER
For example, you might not be allowed to use this expensive or delicate device. Programs running under your user ID can't use the device and must be able to recover from the refusal.

SPECIFIES WRONG PRIVILEGE LEVEL FOR A DEVICE
To use a device (for example, to read a certain file or to place a long distance phone call), the program must supply a code that indicates its privilege level (often the privilege level of the person using the program.) If its privilege level (or priority) is high enough, the program gets the device.

NOISY CHANNEL
The program starts using a device, such as a printer or a modem. A communication channel links the computer and the connected device. Electrical interference, timing problems or other oddities might cause imperfect transmission of information over the channel (we.e., the computer sends a 3 but the device receives a 1). How does the program detect transmission errors? What does it do to signal or correct them?

CHANNEL GOES DOWN
The computer is sending data through one modem across a telephone line to another computer (and modem). One of the modems is unplugged halfway through the transmission. How do the sending and receiving computers recognize that they are no longer connected, how long does it take them, and what do they do about it? Similarly, how does a computer recognize that it's connected to a no-longer-printing printer, and what does it do about it?

TIME-OUT PROBLEMS
The program sends a signal to a device and expects a response within a reasonable time. If it gets no response, eventually the program must give up, deciding perhaps that the connected device is broken. What if it just didn't wait long enough?

WRONG STORAGE DEVICE
The program looks for data or a code overlay on the wrong floppy disk, removable hard disk, cartridge, or tape reel. Some programs announce that the information isn't there, then ask you to insert the right disk. Some look for the information on other drives first, then ask. Others just die. In one particularly feisty operating system, programs could destroy a floppy disk's directory while looking for a file that wasn't there.

DOESN'T CHECK DIRECTORY OF CURRENT DISK
Insert one disk (hard disk pack, tape), work with it, then remove it and insert a different one into the same drive. Some operating systems don't detect the swap. They copy the directory of a disk into memory and don't read it again from the disk unless you explicitly tell them to. If you don't force a reboot or a directory reread, they'll use the old directory when trying to read or write to the new disk, reading gibberish and destroying the new disk's data when they write.

DOESN'T CLOSE A FILE
When the program finishes with a file (especially if it's been writing to the file), it should close the file. Otherwise, changes made to the file during this session might not be saved on disk, or further changes may be added inadvertently. Open files can be destroyed or corrupted when you turn the machine off. Programs should close all open files as part of their exit procedures.

UNEXPECTED END OF FILE
While reading a file, the program reaches the end-of-file marker. Suppose the program expects to find specific data later in the file. Does it ignore the end-of-file and try to keep reading? Does it crash?

DISK SECTOR BUGS AND OTHER LENGTH-DEPENDENT ERRORS

Disk storage is done in chunks (sectors) of perhaps 256 or 512 bytes. Many other values, usually powers of 2, are common. Some programs fail when they try to save or read a file that is an exact multiple of a sector size. For example, if sectors are 1024 bytes, a program might be unable to save files that are 1024, 2048, 3072, etc., bytes long. (Similarly, a program that copies data to an output buffer of a fixed size might fail if the number of bytes to go is the same size as the buffer.)

The last character of each sector, or only the last character of the file, might be miscopied,

copied twice, or dropped. In more extreme cases, the program corrupts the entire file or overwrites the next file on disk.

WRONG OPERATION OR INSTRUCTION CODES
The program sends a command to the terminal that is supposed to reposition the cursor onscreen but it turns on inverse video display mode instead. The program sends a command to the printer to do a form feed, but the printer line feeds instead.

Devices are not standardized. Two printers probably require two different commands to do the same thing. Similarly for terminals, plotters, and A/D converters. The program must issue the right command for this device to get the right task done.

MISUNDERSTOOD STATUS OR RETURN CODE
The program sends a command to the printer telling it to turn on boldfacing. The printer may respond, saying it can or can't do this. It may indicate why it can't carry out the command (e. g., no paper, no ribbon, no such command, option module not installed.). Many programs ignore these codes, or compare them against a list of codes written for the wrong machine or compiled years ago.

DEVICE PROTOCOL ERROR
The communication protocol between the computer and a device or between two computers specifies such things as when the computer can send data, at what speed, and with what characteristics (parity, stop bits, etc.). It also specifies whether and how the receiving device will signal that it got the data, that it's ready for more, or that it can't take any more until it clears some of the buffer that it's working with.

A device might send data or respond out of turn, or it might send the data in the wrong format.

UNDERUTILIZES DEVICE INTELLIGENCE
As a simple example, if the printer can print boldface text directly, why try to simulate boldface by printing on top of the same character three or four times? One possible answer is that the program was designed with less capable printers in mind, and has not been updated to take advantage of this printer's features.

A connected device might be able to define its own fonts, detect its own error states, etc., but the program using the device must recognize this or it won't make any use of these advanced capabilities.

This can be a touchy issue. Printers' control codes differ by so much that it is very expensive to try to support all the built-in features of every printer. Some printer manufacturers have made this problem even more complex by including certain control codes in one ROM

version for a printer, but including different codes in other ROMS, plugged into the same (from appearance and model number) printer.

PAGING MECHANISM IGNORED OR MISUNDERSTOOD
This is a memory storage issue. Memory might be divided into sections called pages. A program might not be able to read from all pages at once or switch pages (or memory banks) correctly.

Larger computers use disk storage as virtual memory. The program refers to data without knowing whether they reside in memory or on disk. If the program references a nonresident set of data or code, a page fault has occurred. The computer fetches the page containing this information from disk automatically, overwriting data that were resident. The operating system usually takes care of paging (swapping data and code between main memory and a disk), but a few programs try to do it themselves. A program might overwrite a memory area without first saving new data that were stored there.

A program that thrashes is constantly generating page faults: the computer spends more time moving data in and out of main memory than it does executing the program. With a little reorganization of the code or the data, many programs can avoid thrashing.

IGNORES CHANNEL THROUGHPUT LIMITS
Examples:

- The program tries to send 100 characters per second across a connection that only supports transmission of up to 10 characters per second.
- The program can send data at a fast rate until the connected device's input buffer is full. Then it has to stop until the device makes more room in its input buffer. Some programs don't recognize signals that the device is no longer ready to receive more data.

ASSUMES DEVICE IS OR ISN'T, OR SHOULD BE OR SHOULDN'T BE INITIALIZED
Before sending text to the printer, a word processing program sends an initialization message telling the printer to print ten characters per inch in a certain font, without making them bold or italicized. Should it have sent this message? This wastes time if the printer was already initialized. It is irritating if you deliberately initialized the printer to a different setting before trying to print the file. On the other hand, if the printer is set to an unsuitable font, the printout will be unsatisfactory and the failure to initialize will have wasted time and paper. Which error is more serious?

ASSUMES PROGRAMMABLE FUNCTION KEYS ARE PROGRAMMED CORRECTLY

A programmable function key might be able to generate any code or any reasonably short sequence of codes when pressed. The program might expect these keys to generate specific codes, but if you can reprogram the keys, the program might be wrong. For example, suppose that function key normally generates . A program says Press PF-1 to Print. It switches to its Print Menu when it receives . What if you reprogram so that it generates instead? Press PF-1 to Print is no longer true.

If the program relies on the assignment of special values to programmable function keys, when it starts it has to make sure that those keys have been assigned those values.

SOURCE, VERSION, AND ID CONTROL
If you're supposed to have Version 2.43 of the program, but some of the pieces you have are from 2.42 and others are advance bits of 2.44, you have a mess. You must know what you have, you must be able to tell from the code what you have, and what you have must be what you're supposed to have. If not, report a bug.

Some people calls these Bureaucracy Bugs because they reflect failures of labeling and procedure rather than operational failures. Only bureaucrats would worry about such things, right? Wrong. Or maybe right, but so what? They must be worried aboutóotherwise the products shipped to customers will not be what you think.

OLD BUGS MYSTERIOUSLY REAPPEAR
Old problems can reappear simply because the programmer linked an old version of one subroutine with the latest version of the rest of the program. Many programs are split across dozens or hundreds of files: Programmers who don't purge old files frequently link old code with new by accident.

FAILURE TO UPDATE MULTIPLE COPIES OF DATA OR PROGRAM FILES
Some programmers repeat the same code in many different program modules. When they have to change this code, they may update 20 of the 25 copies of it, forgetting the others. As a result, they might fix the same error 20 times, but you might still find 5 more just like it the next time you test.

NO TITLE
The program should identify itself when it starts. You should know right away that you are now running Joe Blow's Super Spreadsheet, not Jane Doe's Deluxe Database.

NO VERSION ID
The program should display its version identification when it starts or when you give it a display version command. Customers should be able to find this ID easily, so they can tell it

to you when they call to complain about the program. You should be able to find the ID easily so that you can tell it to the programmer when you find bugs.

If the program is made of many independently developed pieces, it pays to be able to identify the version of each piece. These IDs may not display automaticallyóyou may have to use a debugger or a special editor to find them. They are useful if they exist and if they are kept up to date by the programmers. However, unless you have firm management backing, do not insist that programmers compile separate version IDs for each module.

WRONG VERSION NUMBER ON THE TITLE SCREEN

Programs usually display a version number on the title screen or in an About dialog box. Usually, the programmer can change the code much faster than she can keep the correct version number updated on the title screen. The result is you might be using software version 2.1 but the title screen still shows 2.0.

NO COPYRIGHT MESSAGE OR A BAD ONE

The program should display a copyright message as soon as it starts. The message should include the copyright symbol (it is common to use (C)), the year(s) that the program was developed, copyrighted or shipped, your company's name and address, and the words *All Rights Reserved*. We use the following form:

Copyright © 1979, 1983, 1987, 1993
Cem Kaner, Human Interface Technologies
801 Foster City Blvd. #101, Foster City, CA 94404
All Rights Reserved

If an earlier version of the program showed a copyright year of 1979, you should still say 1979 in this notice, along with this year's date. For more details, ask your company's attorney.

ARCHIVED SOURCE DOESN'T COMPILE INTO A MATCH FOR SHIPPING CODE

Before releasing a product to any customer, archive the source code. If the customer finds a bug, your company must be able to recompile this code and regenerate the product that the customer has. Without this starting point, you will have major problems addressing that customer's difficulties. This should be obvious, but it seems not to be. I've been amazed at how many companies can't recreate products they sell. They may have archival copies of source code, but the code in their vaults is a bit different from the code in the product they shipped. This is begging for a disaster. If you report that archives are not up to date with code that is about to be shipped, and are rebuffed, take it to a higher level. The president and the company lawyer might be much more concerned by this problem than mid-level engineering or marketing managers.

MANUFACTURED DISKS DON'T WORK OR CONTAIN WRONG CODE OR DATA

When the disks have been duplicated and the product is ready to ship, check a few disks. We are not suggesting that you take over the role of manufacturing QA. We are suggesting that any error that occurs in all manufactured copies of the product is an error you should find.

Disk duplicators might crank out blank disks instead of the copies you expected them to make. It is embarrassing when you ship blanks as the product, and expensive to send customers replacement disks. This does happen. Over the last two years, I've received blank disks as a customer three times (three different companies). Similarly, the manufacturing group might duplicate the wrong version (Version 1.0 again instead of 2.0) or the wrong program (buy a database, get a spreadsheet instead), sometimes because you gave them the wrong disks to duplicate.

TESTING ERRORS

This section deals with technical, procedural, and reporting errors made by testers and Testing Groups. Even though these aren't problems in the programs per se, you'll run into them when testing programs. Our focus is on suggestions for dealing with these problems, since they're under your control.

MISSING BUGS IN THE PROGRAM

You will always miss bugs because you can't execute all possible tests. However, you'll probably miss more bugs than you have to. When a bug is discovered in the field or late in testing, ask why. Not to assign blame but to look for ways to strengthen your test procedures.

Failure to notice a problem

You may miss a bug that a test exposes because:

- You don't know what the correct test results are. Whenever possible, include expected results in the test notes. In automated tests, display them beside test results on the screen and on printouts.
- The error is buried in a massive printout. People scan long outputs quickly. Keep printouts as short as you can, and make errors obvious at a glance. Patterned outputs are good. If possible, redirect long outputs to a disk file; have the computer check this against a known good file.
- You don't expect to see it exposed by this test. While a test may be designed to focus on one small part of a program it can still reveal other, unexpected, bugs. Beware of tunnel vision.
- You are bored or inattentive. Rotate tasks across testers. Try not to have the same

person run the same test more than three times.
- The mechanics of running the test are so complicated that you pay more attention to them than to the test outputs. You will be distracted by files or printouts with erroneous comparison data, poorly organized checklists, procedures that require you to swap disks or tapes frequently, and tasks that you have to redo from the start if you make an entry error.

Misreading the screen
You can easily miss errors like spelling mistakes, missing menu items, and misaligned text. Reserve some time exclusively for scrutinizing the screen. It's just like proofreading manuscripts: unless you're consciously looking for spelling and layout errors, you'll see what you expect to see, filling in gaps and correctly spelling mistakes unconsciously.

Failure to report a problem
You may find a problem and not report it because:

- You keep poor notes
- You're not sure if it's a bug and are afraid to look silly
- You think it's too minor or you don't think it will be fixed
- You're told not to report bugs like this any more

These are not acceptable reasons. If you're not sure whether something is a problem, say so in the report. Appeal to higher management to relieve criticism for reporting minor or politically inconvenient bugs. It is your responsibility to report every problem you find. Deliberate suppression of bug reports leads to confusion, poorer tester morale, and a poorer product. It can also bring you into the middle of nasty office politics, possibly as a scapegoat.

Failure to execute a planned test
You may not execute a planned test because:

- Your test materials or notes are disorganized. You've lost track of what has been tested.
- You are bored. The test series is repetitive. You take shortcuts by skipping tests that are similar to others. To reduce this, rotate tasks among testers. Reduce repetition by combining cases, cutting some out, or running some tests only on every second or third cycle of testing.
- You have combined too much into one test. If one test is buried inside another, or depends on another, then if that other test fails, this test probably won't be executed. Overly complex combinations of test cases can lead to missed tests because they confuse you.

Failure to use the most "promising" test cases

If two test cases cover essentially the same code, you should use the one most likely to reveal an error (see Chapter 6).

Ignoring programmers' suggestions
The programmer knows better than anyone else which areas of the program he tested least, and which ones proved least stable under his testing. He knows which areas he coded quickly. He knows which special types of tests have exposed bugs so far. Rigid test plans and bad politics are problems in their own right, but they are not excuses for ignoring programmers' tips.

FINDING "BUGS" THAT AREN'T IN THE PROGRAM
You report an error. Eventually, the problem is traced to a flaw in your test procedure, a misunderstanding of the program, or to something else that you did. This wastes time and does your credibility no good.

Errors in testing programs
When you automate tests, you write programs to drive test cases. Your test programs will have bugs. Some will abort your tests, or skip them. Others will make the program appear to fail tests that it can actually pass. It is common to compare test data against incorrect "known good" results. Your disk files and printed constants are no more likely correct than the program's output.

You should manually reproduce any automated tests that reveal errors. This doesn't take all that much time because you only redo tests that the program fails. Unless the program is in disastrously bad shape, it won't fail many tests.

Corrupted data file
Some apparent bugs are due to a bad data file that you're using while testing. Programs will trash input, output, and comparison files. Your files may be corrupted at any time, even by program segments that, if they were error-free, wouldn't read or write these files. When a program is in testing, it doesn't matter that it isn't supposed to touch a file. If the program worked the way it was supposed to, you wouldn't have to test it. It is wise to keep three backup copies of test files on separate disks or tapes. Before reporting an error, check your working copies of the input and comparison files against the backups.

Misinterpreted specifications or documentation
You think the program works incorrectly because you've misunderstood the documentation. This is unavoidable. Specifications are outdated, and early versions of the documentation are rough. You rarely have much time to read before starting to test the program.

When you find an error, unless you're sure you understand what's happening in this part of the program, reread the relevant sections of the documentation and specifications. If you're

not sure whether what you've got is an error, write your report as a Query. If the manual's unclear, file a Problem Report on that part of the manual too.

POOR REPORTING

It's not enough to find a bug. You have to communicate it to someone who can fix it, in a way that makes it as easy as possible for that person to figure out what went wrong and what to do about it. How well you describe the problem will directly affect how easily it is resolved.

Illegible reports

If the programmer finds it hard to read a report, he will ignore it for as long as possible. Many reports are hard to read because you pack too much information into them. Put separable problems on separate report forms. If a single problem requires a long description, type it on a separate page and attach it to the Problem Report.

Failure to make it clear how to reproduce the problem

You report a problem without outlining, step by step, what the programmer must do to see it. This is the most common error in problem reporting. It saves time to skip the details, but realize that the first thing that the programmer is going to do with your report is sit at the machine and try to see the problem himself. If he can't reproduce the problem, he won't fix it.

For anything complicated, attach a copy of any data files you were using, a keystroke by keystroke list of things you did, a printed dump of the screen if your operating system supports this, or any other comments or materials that will make the programmer's job easier. The nearer you are to the development deadline, the more important this is.

Failure to say that you can't reproduce a problem

If you can't consistently reproduce the problem, say so. This tips off the conscientious programmer that he should try variations on the conditions you describe. A non-conscientious programmer might ignore your report as soon as she sees that you can't replicate it, but she'd toss away the report anyway after trying exactly what you say you did and, like you, failing to see the bug.

Failure to check your report

After writing a report, but before submitting it, follow it step by step to reproduce the problem. This costs a few moments but it catches transcription and other reporting errors that you make. It is all too easy to omit or misdescribe important details, especially if you're writing the report from notes or memory, long after seeing the bug.

Failure to report timing dependencies

You might not notice that to reproduce a bug you have to press two keys within milliseconds of each other, or that you have to wait at least 5 minutes between keystrokes. Sometimes you

will just not realize that you're dealing with a race condition or other time-dependent bug. If you do notice a time dependence, say so. Clock it as well as you can. If you didn't notice a time dependence, look for one when a report comes back to you as irreproducible.

Failure to simplify conditions
You will often use complex test cases, combining many different tests into one, for speed of testing. If all goes well, you've gotten through many tests quickly. When a bug does show up, spend time looking for the simplest series of steps possible to reproduce it. Try not to lay out a long and complicated series that includes irrelevancies. Complex reports are disheartening to read and tempting to ignore.

Concentration on trivia
Don't make big issues over small problems. Don't get too far drawn into long arguments over wording, or style of presentation. Don't exaggerate the severity of bugs. Be wary of getting a reputation as a nitpicker.

Abusive language
If you refer to work as "unprofessional," "sloppy," or "incompetent," expect the programmer who did it to get angry. Don't bet that he'll fix the bug, even if it's serious. It can be useful to shock a programmer occasionally, but be conscious of what you're doing. Do it rarely (once a year).

POOR TRACKING OR FOLLOW-UP
It's not enough to just report a bug. You've got to make sure that it's noticed and not forgotten. Otherwise, bugs will "slip through the cracks" and make it into the shipping product.

Failure to provide summary reports
Don't assume that just because you gave it to a programmer, it's being dealt with. Some programmers lose reports. Others use them to make paper airplanes or wrap fish. Some also hide reports from their managers. Every week or two, you should circulate a brief description of unfixed bugs. Make this a standard procedure, for all bugs, to keep it impersonal and uncontroversial.

Failure to re-report serious bugs
If the bug is serious, don't automatically accept a response of Deferred or Works to Spec. Figure out a way to make it look a little worse and report it again. If it's an ugly, horrible bug, make it sound that way the second time. If that doesn't work, send a copy of the third report to a more senior manager.

Failure to verify fixes
A programmer reports that he fixed a problem. Don't take his word for it without retesting.

Up to a third of the fixes either won't work or will cause other problems. Further, some programmers only address the exact, reported symptoms. Instead of investigating the causes of a problem, they write special-case code to handle the precise circumstances reported. If you skimp on regression testing, you will assuredly miss bugs.

Failure to check for unresolved problems just before release
Just before the product is released for use or sale, check for problems that are neither fixed nor deferred. It's good practice, which we recommend highly, to make sure that all Problem Reports are resolved one way or another before the product is released. At a minimum, make sure no one's sitting on anything serious. This is your last chance to remind people of serious bugs.

Return to top

Testing Computer Software
Second Edition

APPENDIX: COMMON SOFTWARE ERRORS