28th International Symposium on Fault-Tolerant Computing: Munich, Germany. 23-25 June 1998.

Improving Software Robustness with Dependability Cases

Roy A. Maxion

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
Tel: 1-412-268-7556
Internet: maxion@cs.cmu.edu

Robert T. Olszewski

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
Tel: 1-412-268-3266
Internet: bobski@cs.cmu.edu

Abstract

Programs fail mainly for two reasons: logic errors in the code, and exception failures. Exception failures can account for up to 2/3 of system crashes [6], hence are worthy of serious attention. Traditional approaches to reducing exception failures, such as code reviews, walkthroughs and formal testing, while very useful, are limited in their ability to address a core problem: the programmer's inadequate coverage of exceptional conditions. The problem of coverage might be rooted in cognitive factors that impede the mental generation (or recollection) of exception cases that would pertain in a particular situation, resulting in insufficient software robustness. This paper describes a study to test the hypothesis that robustness for exception failures can be improved through the use of dependability cases. Dependability cases, derived from safety cases, comprise a methodology based on structured taxonomies and memory aids for helping software designers think about and improve exception-handling coverage. A controlled experiment conducted with 59 subjects revealed a statistically significant 43% increase in exception-handling robustness. An ancillary experiment conducted with 38 subjects provides convergent evidence that the effect is authentic, and not due to programming expertise alone.

KEYWORDS: Dependability, empirical methods, exception handling, safety cases, software engineering/robustness.

1. Introduction

On 4 June 1996, maiden flight 501 of the European Space Agency's new Ariane 5 heavy-lift rocket ended in failure, exploding roughly 40 seconds into the mission. As reported by a board of inquiry [13], the problem was identified as a software exception in the inertial reference system "caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an operand error. The data conversion instructions were not protected from causing an operand error ..." The operand error occurred due to an unexpectedly high value of an internal function result. No justification was found for not making the operand robust against software exceptions. Thus was lost, due to a simple exception failure, an uninsured scientific payload valued at roughly 500 million dollars.

Exception failures occur when a program is prevented by extant circumstances from providing its specified service [6]; one such circumstance is the inadvertent use of zero as a divisor. In general, an exception is any unexpected condition or event, usually environment-driven or data-driven, which would cause an otherwise operational program to fail. Many different types of conditions can cause exceptions. Some examples are: empty data file, insufficient memory, type mismatch, wrong command-line argument, protection violation and bad data returned from another program. These kinds of conditions can be guarded against, yet frequently they are not.

Reducing the occurrence of exception failures is beneficial for several reasons. (1) Software would be more robust; better software means higher availability and lower operating cost. (2) Testing/assurance costs would be reduced. Testing can often consume more than 50% of a development effort [10]. The testing literature suggests that it is less expensive to detect flaws at their source through code inspection than it is to detect them through testing (e.g., [3], [4]). Moreover, since testing tends to be most thorough for portions of the code providing the application functionality, and least thorough for the portions providing exception handling [6], it's not clear that testing is particularly effective for eliminating exception failures; better results might be achieved through avoidance - by getting it right the first time. (3) Finally, a substantial percentage of security vulnerabilities would be eradicated. The computer security community has observed common mechanisms (e.g., buffer overflow) that cause exceptions and security vulnerabilities simultaneously; they claim that eliminating exception failures would eradicate about 50% of security vulnerabilities [5].

Exception failures are a serious problem, not only in mission-critical applications, but also in commercial, shrink-wrapped software systems and laboratory code where quick, accurate results are essential. Programs are often logically correct, but nevertheless fail due to improperly handled exceptions. This paper asks why exceptions are often ignored, and describes a new approach - dependability cases - that has reduced exception failures, and raised robustness by 43%.

2. Problem, background and approach

This paper addresses the problem of reducing the number of exception failures caused by inadequate exception handling in code. The approach taken is to regard instances of inadequate exception handling as errors in coding; such errors are effectively design errors or, ultimately, human errors. Cristian [6] would appear to support this position when he says, citing Toy [17], that "approximately two thirds of system failures are due to design faults in exception handling (or recovery) algorithms." His use of the term "design fault" raises the following kinds of questions. What kinds of errors do programmers make when they fail to cover exception conditions? Why do they make these mistakes? What, if anything, can be done about it?

Many papers in the exception-handling literature allude to the seriousness of the exception-failure problem, but reveal little data on the frequency and severity of exception failures; neither do they ask what can be done to improve exception handling in code. Their main concern is programming-language constructs for handling exceptions. Cristian [6], in a thorough survey of the literature, suggests one reason for the poor state of exception handling: "In operational computer software systems often more than 2/3 of the code is devoted to detecting and handling exceptions. Yet, since exceptions are expected to occur rarely, the exception handling code of a system is in general the least documented, tested and understood part." Even so, this sidesteps elements of human error that may contribute to the problem.

Is the pervasiveness of exception failures in programs due to some underlying peculiarity or frailty of the human cognitive system? Is there something that makes it hard for programmers to remember all the exception conditions that pertain to a given situation? Cognitive scientists began studying human error over a hundred years ago [9]. Reason [14], in a comprehensive review of what is known about human error, offers a number of possibilities. He states that "simple omissions constitute the single largest category of human performance problems ... Mental task analysis shows a close association between omissions and the planning and recall of procedures." In nuclear power plants, for example, omissions accounted for 42.5% of all incidents. As a percentage of different kinds of tasks, e.g., monitoring, inspecting, controlling, testing, modifying, etc., 74% of omissions occurred in tasks involving testing and modifying. For tasks that involved mental activities such as planning, recall, observation, detection, etc., 90.6% of omissions were in planning and recall. Eliminating exception vulnerabilities from a program requires tasks similar to planning, testing, modifying and recall. Hence, it's no surprise that programmers experience omission errors, too, and thereby fail in achieving better exception coverage.

Why do programmers make omission errors? Why aren't they better at remembering the kinds of exception cases that need to be covered? What prevents people from generating a comprehensive mental checklist of exception types? Human memory has many natural limitations [19]. Most people aren't good at keeping long lists in memory. Long lists are hard to recall, especially when the list has no salient structure. If there were a way to initiate recall of a structured list, there might be easy ways to extend what is remembered, thus eventually regenerating a very long list from memory. People often use tricks, like mnemonics, to remember lists. One well-known mnemonic, "Roy G. Biv," helps us remember the colors of the visible spectrum (red, orange, yellow, green, blue, indigo and violet). Mnemonics work because they have a salient structure, and because they collapse long sequences into short chunks that are more easily remembered. If programmers had a mnemonic for recalling basic categories of exception failures, the recall of these categories might be easily extended to think about types of exceptions within the categories. For example, if a category is named "null pointer and memory," then programmers, having had the context of memory problems instantiated, would be able to think of many memory problems (e.g., buffer overflow, invalid pointer, etc.).

From a theoretical perspective, a mnemonic would be effective, because the mnemonic would facilitate the initial recall of the categories, and then each category would provide a context for semantically extending the elements of the category to include all or most of the category members (a process that cognitive scientists call priming [19]). If the mnemonic was linked to a physically salient graphic structure, memory would be even further enhanced. An example of such a mnemonic is *children*. "Everyone knows exceptional *children*." The letters of the word *children* can be used to remember the list of exception categories shown in Table 2-1. The associated graphic is shown in Figure 3-2.

C omputational problem
H ardware problem
I /O and file problems
L ibrary function problem
D ata input problem
R eturn-value problem: function or procedure call
E xternal user/client problem
N ull pointer and memory problems

Table 2-1: The exceptional *children* mnemonic.

Such mnemonic devices could address the matter of exception coverage; the issue of *correctly* handling exceptions is a separate matter. The framework proposed for remembering exception cases in an organized fashion is the dependability case, which will be described in the next section.

3. Dependability cases

It is hypothesized that exceptional conditions in code are left unguarded because programmers do not think of them. Dependability cases, developed in this section, comprise an organizing framework and a methodology for thinking about exceptions and the conditions under which they occur. They also improve mental recall of exception conditions.

Dependability cases derive from an evolving body of work on safety cases [15], emanating mainly from the United Kingdom and Europe. A safety case is "essentially a clear, defensible, comprehensive and convincing argument ... aimed at identifying the risks inherent in operating a system, demonstrating that the operating risks are fully understood, that they have been reduced to an acceptable level and are properly managed [16]." Bishop and Bloomfield [2], in their definition of a safety case, add that the system should be "adequately safe for a given application in a given environment." They also note that implementing a safety case requires: "(a) making an explicit set of claims about the system; (b) providing a systematic structure for marshalling the evidence supporting those claims; (c) providing a set of arguments linking the claims to the evidence; (d) making clear the assumptions and judgments underlying those arguments; and (e) provide for different viewpoints and levels of detail."

At this writing, dependability cases are still in the early stages of definition and development, and are currently intended to address only exception conditions in code. They presently mirror some aspects of safety cases: they provide a systematic structure for elucidating exception hazards, a mechanism for establishing causal paths leading to exceptions, and a written defense justifying that every identified exception is handled. Dependability cases, as do safety cases, include methodologies and constructs for systematic identification of the potential hazards in a system. One example is hazard analysis [11]. As described by Leveson [12], hazard analysis encourages creative thinking about all the possible ways in which hazards or operating problems might arise. The technique is able to elicit hazards in new designs, as well as hazards that have not been considered previously (i.e., not included on checklists and standards developed from earlier systems). Basically, hazard analysis explores deviations from expected conditions - essentially, exceptions. Hazard analysis does not provide quantitative results; its strength is that it systematizes a qualitative, brainstorming-type approach.

Another construct employed in both dependability and safety cases is fault tree analysis [18]. Fault trees are widely used, in conjunction with hazard analysis, to set down, in a logical way, the events leading to a hazardous occurrence (e.g., exception). Fault trees are a means for analyzing causes of hazards, not for identifying the hazards themselves. A fault tree is a graphical device that helps organize relational and temporal information about faults and the causes that lead to them. Using logic symbols, fault trees represent a "top event," the causal events that lead to it, and the logical relations among those causes.

The abbreviated fault tree shown in Figure 3-1 depicts a chain of events leading to failure in a simple copy machine. This figure was included in the materials for the experiment described in Section 4. The top event is "copy failure." Among the causal contributors to copy failure, some may cause failure all by themselves, and some may cause failure only when they co-occur with others. These situations are represented in the fault tree by OR and AND gates, respectively. Neither fault 3 (indicated by the

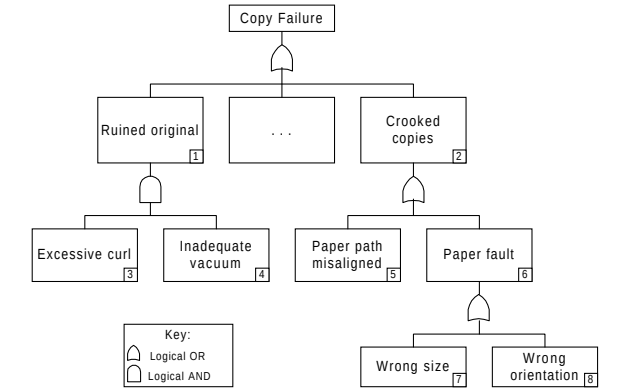

Figure 3-1: Fault tree for copier.

number in the lower right corner of the box labeled excessive curl) nor fault 4 alone will cause copy failure, but both faults occurring together will cause failure; these two faults are connected by an AND gate. Fault 2 is connected to faults 5 and 6 with an OR gate; either of these faults alone will cause copies to be crooked. Similarly, a paper fault can be caused either by paper being the wrong size for the feeder tray (7) or by paper being oriented the wrong way in the tray (8). Notice that for the numbered conditions in the fault tree, a written addendum, numbered in correspondence with the conditions, would indicate how the faults are guarded against.

Together, hazard analyses and fault trees can be used to obtain a reasonably clear picture of a system's exception vulnerabilities. A hazard analysis produces a list of hazards to guard against, either by fault avoidance or by fault tolerance. A fault tree takes that list of hazards and helps to elucidate the causal events or conditions that lead to a given hazard. Hazard analyses rely to some extent on the imagination and experience of the analyst in generating a list of hazards; it helps to have prior knowledge of what kinds or classes of vulnerabilities to look for. In such cases, a checklist or taxonomy of exception types would be useful. In checking a system design, however, long lists of hazards or events are apt to go unused; they are hard to remember, and are seldom immediately available in printed form when needed. Moreover, such lists are difficult to reproduce from memory, even if they are organized taxonomically or hierarchically. One needs an anchor from which to start.

One way to make lists easier to remember is to encode them as graphical structures that are perceptually salient. Bertin [1] provides guidance on the perception of graphical structures. One graphical technique that facilitates recall and provides relational structure is the so-called fishbone diagram, also known as a cause-and-effect diagram or an Ishikawa diagram, after its founder, Kaoru Ishikawa [8]. Fishbone diagrams are widely used in

quality control. Figure 3-2 shows an example, roughly in the shape of a fish, that depicts exceptions that could be encountered in a software system. At the "head" of the fishbone is the phenomenon to be avoided: exception failures. The ribs are labeled with categories of events that cause exception failures, and the events within each rib are examples of specific causes. For instance, the rib labeled "computational problem" lists divide-by-zero as an exemplar. The fishbone in the figure, although possibly incomplete, is an attempt to lay out a fairly comprehensive set of exception causes covering most programs. The exemplars were obtained via hazard analysis. Notice that the first letters of the rib labels spell the word *children*, the mnemonic discussed in Section 2: Everyone knows exceptional children.

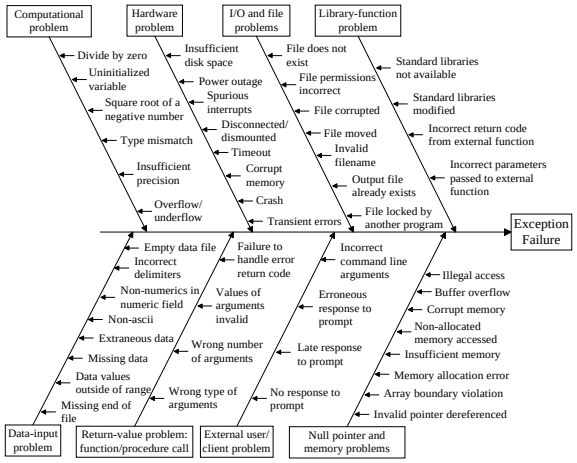

Figure 3-2: Fishbone diagram showing exception types and exemplars. The first letters of the rib labels spell the mnemonic *children*.

An ideal dependability case might be little different from a safety case, although it's unclear that all of the elements of a safety case are necessary if one's goal is simply to improve coverage of software exceptions. The term "safety case" is suggestive of safety-critical software, and may not be taken seriously by programmers working on such projects as spreadsheets which, at first blush, do not appear to be safety critical; so, the term "dependability case" is introduced here in an effort to broaden the appeal to all programmers. The present work defines a dependability case in terms of the steps shown in Table 3-1. The written defense in the last bullet may be constructed in the abbreviated form suggested in the text accompanying Figure 3-1; this will focus attention on omissions often exposed by the writing process.

The hypothesis set forth in this paper is that if programmers were given a structure for organizing exception-failure concepts, as well as a memory aid for priming

```
1. Generate a LIST of hazards/exceptions to be
   guarded against.

      Use hazard analysis to elucidate possible
      exceptions.

      Use the exception fishbone as a reminder
      of exception types.

2. Establish the CAUSAL PATHS leading to each
   exception possibility.

      Select fault-tree top events, using the
      aforementioned exception list as a guide.

      Perform a fault-tree analysis to guide causal
      path discovery for each top event/exception.

3. Write dependability DEFENSE for each top event.

      Defend the statement: this code is completely
      robust against exception failures.  Use the
      fault-tree structure to guide the defense.
```

Table 3-1: Outline of dependability-case steps: list exceptions, establish causal paths, write defense.

those concepts, they would write more robust code; their code would be more resistant to exception failures. The following sections describe experiments to test this hypothesis.

4. Experimental method

This section describes the experimental work conducted to test the hypothesis that robustness to exception failure improves when programmers use dependability cases. Two experiments were conducted. The first experiment involved no programming. It tested only the idea that using dependability cases improves coverage of exception conditions; increased coverage was indicated by generating larger numbers of potential exception conditions. The success of this experiment led to a second experiment that was the same as the first, but required programming.

4.1. Experimental design

A two-by-two design was used, with nonprogramming vs. programming groups, and control vs. treatment groups.

Non-programming condition. In the first of two experiments, programmers were recruited to generate a list of exception conditions, testing only their ability to think of exceptions, not to write code to protect against them. Participants were asked to write a list of all the exception conditions they could think of, within the context of the given task. The participants were divided into control and treatment groups. Controls had no special instructions; treatments had special instructions for writing an accompanying dependability case justifying the claim that the resulting program would be completely robust against exception failures. The objective was to see if the treatment group could think of more exceptions than the control group could. Details are given in the sections below.

Programming condition. The second experiment resembled the first one, except that the participants were required to write C code to implement a simple programming task (described in Section 4.3). The programmers were divided into two groups: control and treatment. The control group performed the programming task without any special instructions; the treatment group performed the same task with special instructions for writing an accompanying dependability case, justifying the claim that the resulting program would be completely robust against exception failures. The objective was to see if the treatment group's programs would be more robust against a test set of exception-generating data files than those of the control group.

4.2. Selection of participants

Four groups of university student subjects were selected from undergraduate and graduate university courses at two American universities. Groups A and B were from one university; groups C and D were from the other. Group A contained 27 graduate and undergraduate students enrolled either in a masters-level course in distributed operating systems, or in a course in computer systems administration. Some of these students had industrial programming experience. Group B was composed of 14 students enrolled in a third-semester undergraduate course in data structures. Group C consisted of 18 graduate students enrolled in a masters course in software engineering. Group D comprised 38 undergraduate students enrolled in a fourth-semester data structures course. Groups A, B and C participated in the programming portion of this experiment; Group D participated only in the nonprogramming portion. The obvious nonhomogeneity of the groups was intentional, representing a range of experience levels found in populations of programmers.

Subjects in each letter group were assigned to control and treatment subgroups. Subjects in Groups A, B and C were sorted by grade-point average, and then assigned to control and treatment subgroups on an alternating basis; thus the distribution of high-grade-point and low-grade-point subjects in each group was approximately equal. Subjects in Group D were assigned randomly to control and treatment subgroups.

4.3. Instructions to participants

All participants, both control and treatment, were given specifications for a simple C program (standard C for a Unix-based platform) that would compute the mean and standard deviation of numbers provided in an input file to be read by the program. The formulae for computing a mean and standard deviation were given. All participants were told to consider that they had been asked to write one of many software modules that would be embedded in a medical device for radiation therapy. The module will accept data from a certain place (a file specified on the command line), will compute a result based on those data, and will return a result to standard output. They were told that the safe, dependable operation of the entire device may depend on the integrity of this one module.

The instructions advised that a correctly-formatted input file should contain an arbitrary number of rows such that each row is delimited from the next by a newline character. Each row should contain an arbitrary number of floating-point values such that each value is delimited from the next by white space (one or more spaces and/or tabs). The input file will be terminated with an end-of-file character. Computed results were to be printed to standard output in row form such that each row contained the mean and standard deviation, respectively, of its corresponding column; i.e., row 1 contains the results from column 1, and so on. Table 4-1 shows the input- and output-data examples given in the instructions.

1.2	2.7	3.8		
1.1	2.3	3.4		
1.1	2.2	3.0		
1.9	2.1	3.3		
1.4	2.6	3.2		
1.2	2.2	3.3	1.40	0.29
1.2	2.7	3.4	2.36	0.26
1.8	2.2	3.0	3.32	0.25

Table 4-1: Example input data (left) and output data (right) shown in instructions (e.g., mean of input column 1 is 1.40, standard deviation is 0.29).

Participants were admonished that their programs must perform dependably, returning either a computed result or a message constituting graceful termination in the event of an exception. The program should indicate success if it is able to compute the mean and standard deviation for each column. The program should indicate failure if it is unable to compute the mean and standard deviation for any column for any reason; any ambiguity should resolve to a dependably safe outcome. In ambiguous cases, neither the mean nor standard deviation were to be computed or printed for any column; rather, a clear and informative error message (i.e., suggestive of how the user/operator should isolate/correct the problem) was to be printed to standard output, and the program should indicate error by returning 1 from main(). Otherwise, success should be indicated by returning 0 from main().

General instructions. All nonprogramming participants, both control and treatment, were instructed to imagine writing the code specified in the instructions, and then to produce a list of all the exceptions that they would need to guard against in the code. Participants were given 30 minutes to complete the task. Results were scored as described in Section 5.

All programming participants, both control and treatment, were instructed to write the code specified in the instructions, covering all exception conditions. Treatment participants were asked, additionally, to construct an accompanying dependability case defending the code's robustness, as described below. No collaboration was permitted. Finished programs were submitted via email within one week. Programs received by email were compiled, tested and scored automatically, as described in Section 5.

Special instructions. Participants in the control group, both programming and nonprogramming, were given no special instructions. They received only the general instructions described above.

Participants in the treatment group, both programming and nonprogramming, received special instructions for writing a dependability case to accompany their programs. These instructions included the dependability-case steps shown in Table 3-1. In addition, they were given a short (3 single-sided pages) tutorial on the use of fault trees and fishbone diagrams. The tutorial included two paragraphs that discussed the consequences of failure in mission-critical systems.

Included in the tutorial were two examples and illustrations. One was the fault tree for a copy machine (already shown in Figure 3-1), and the other was a fishbone diagram, also for a copier, shown in Figure 4-1. The intention of using a copier as an example was to demonstrate how these graphic and organizing tools are used, without introducing bias in the programming domain.

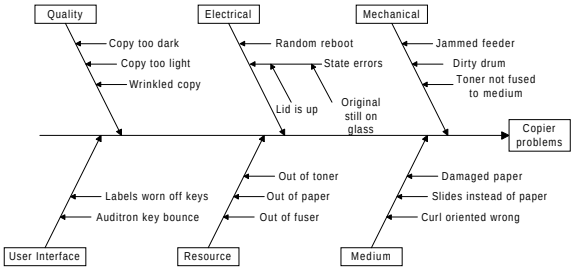

Figure 4-1: Copier fishbone for treatment groups.

The treatment groups were also provided with a skeleton fishbone diagram for exception conditions in computer programs. This diagram, shown in Figure 4-2, provided examples of categories of exceptions as shown in the labeled boxes at the ends of the fishbone ribs. These categories, such as problems with input data, memory, computations, functions or procedure calls, clients or users, hardware, file I/O, and library-functions, served as semantic anchors for generating exception-condition exemplars within each category. One category, computational problems, included two exemplars as illustrative examples. Participants were free to invent categories of their own.

Participants were encouraged to create a fault tree, as an option, covering only the conditions in their particular program. Participants were encouraged, but not required, to (i) draw a fault tree, (ii) number the boxes as exemplified in Figure 3-1, and (iii) provide a numbered list of protections corresponding with the numbered conditions in the fault tree.

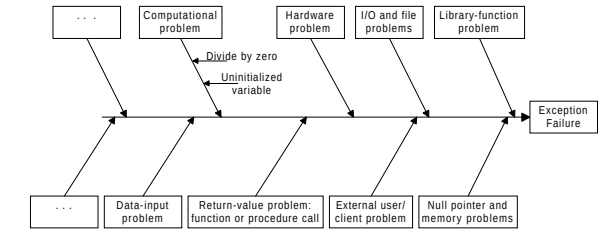

Figure 4-2: Blank fishbone for treatment groups.

4.4. Procedure

Separate instruction packets were distributed to participants, depending on their group: control vs. treatment and programming vs. nonprogramming. Programming participants were given a week to complete the task; source code was to be sent via electronic mail to a specified Internet address. Nonprogramming participants were given 30 minutes to write their results; papers were handed in to experimenters at the end of the session.

Each of the 59 programs was run against the set of 25 test cases shown in Table 4-2. Entries in the table marked by "-" were expected to generate exceptions; other entries were for general tests of functionality. Entry number 16 was ambiguous; this condition could have been guarded by protection from divide by zero, or by a restriction against having only one data row; computing the standard deviation of a single data point isn't sensible. Each program was stripped from email, compiled, tested and scored for quantitative pass/fail automatically. Qualitative scoring (for quality of error messages) was done by a panel of judges. Scoring is discussed in Section 4.5.

4.5. Scoring criteria for programs

This section shows how programming and nonprogramming results were scored. Results for the nonprogramming group were reported in terms of the number of exception conditions listed by each participant. Results for the programming groups were reported in terms of the number of test cases handled by a given program.

Scoring: Nonprogramming condition. Scoring was done on the basis of the number of valid exceptions identified. To guard against the possibility that any exceptions listed by participants were not valid (e.g., Pentium divide error, forgot a semicolon, etc.), such entries were eliminated by agreement of a panel of judges before scoring. All other entries were credited as being valid exceptions.

Scoring: Programming condition. Two types of test cases were used: normal and fault. A normal case was not expected to generate an exception when processed. An example of a normal case is number 7 in Table 4-2 (col_3.dat) which contained three columns of good data;

```
 1- First line is blank
 2- Blank lines interspersed throughout data
 3- Characters intermixed with numbers
 4. One column of data
 5. Many columns of data
 6. Two columns of data
 7. Three columns of data
 8- Data file does not exist
 9- Empty file
10- Extra values interspersed throughout data
11- Extra value in first row
12- Large values (overflow)
13- Missing values throughout data
14- Missing values in first row
15- Data file not provided on command line
16. One row of data
17. Many rows of data
18. Two rows of data
19. Three rows of data
20. All values the same in each column
21- Characters exclusively (no numbers)
22. Separating row values by >= 1 space
23. Separating row values by >= 1 space and/or tab
24. Separating row values by more than one tab
25. Negative numbers
```

Table 4-2: Test cases. Entries marked by "-" are exception cases, expected to generate exceptions; others are general cases, intended to test general functionality.

nothing about this test case would be expected to generate an exception. Fault cases were intentionally seeded with errors that, when processed by a program, would be expected to generate exceptions. An example of a fault case is number 3 (char.dat) which included alphabetic characters interspersed in numeric data. The character data would generate an exception during a computation. These characterizations of normal and fault are useful when computing the scores for programs submitted to the experiment, as discussed in subsequent sections.

Table 4-3 details criteria for the two levels of scoring used: general and exception. Both of these were integrated into an automated scoring algorithm. General scoring was the strictest, demanding that everything be absolutely correct and in exact accordance with the specifications issued to the participants. For example, according to the table, mean and standard deviation needed to be correct and in the right format, and error messages were required to be informative and to identify a problem precisely. On the other hand, exception scoring recognizes that the fundamental goal of the experiment was neither to test adherence to specifications nor to test general coding abilities, but rather to test exception-handling coverage. The table shows, for example, how a program would pass exception-level scoring for fault test cases: it would accept any return code (don't care); it would accept any error message; and output formatting would not be an issue (not applicable), since the fault test case would prevent a result from being computed anyway. The only criterion for an exception-level pass was that the exception itself be handled correctly, and that the program not crash. Although these criteria seem lax, they are all that's needed to determine whether or not an exceptional condition has been properly handled.

Normal Test Cases	General Scoring	Exception Scoring
Return Code	0	Don't care *
Computed Values	Mean & sdev correct	Mean & sdev correct
Output Formatting	Two columns Two decimals No extraneous text	Don't care
Fault Test Cases	**General Scoring**	**Exception Scoring**
Return Code	1	Don't care *
Error Message	Exception precisely identified	Any error message **
Output Formatting	Not applicable	Not applicable

* except signals raised by operating system during program execution, e.g., segmentation fault or floating exception.

** except errors due to artificial limitations introduced by the programmer, such as hard-coding the maximum number of columns or rows that may appear in a data file.

Table 4-3: Scoring criteria for programs: general-level and exception-level scoring for normal and fault test cases.

Error-message adequacy. Judging the adequacy of error messages was based on the length of time it would take an operator to locate a problem when presented with the given error message. These rules were used only in cases in which it was expected that the program should generate an error message (fault test cases). The error-message rules, ordered from least to most strict in Table 4-4, were incorporated into the automated scoring algorithm. A general-pass required that the error message be at level 5 or 6; an exception-pass required at least level 3; all others failed. These rules were used only in the programming experiment, not the nonprogramming experiment.

5. Results

This section presents the results of two experiments conducted to test the hypothesis that code written by programmers using dependability cases will be more robust against exception failures than code written by programmers not using dependability cases. Note that the test set for the programming experiment included two kinds of cases: normal (for testing general functionality) and fault (for testing exception handling). The results will be discussed in terms of these two kinds of cases.

```
1 - No Error Message Generated
        No message; or erroneous display of numerical output
        instead of identifying the exception.
2 - Internal Exception Reported
        "You can only have ten columns."
3 - Misidentification of Exception
        Correctly stated that something was wrong, but identified
        wrong thing; misleading.  Would have taken more time to
        fix the problem based on this error message than if the
        message had correctly identified the problem;  would
        possibly take more time, having been mislead, than if
        there were no error message at all.
4 - Error Message Vague or Unspecific
        "There is bogus data."
5 - Exception Correctly Identified
        "Found character 'a' in the data."
6 - Exception Correctly Identified and Located
        "Found character 'a' in line 3 of the data."
```

Table 4-4: Rules (and examples) for judging error-message adequacy: ordered least to most strict.

The programming experiment required participants to write and submit program code to the experimenters. This experiment tested not only the extent to which programmers could cover all exception conditions, but also their ability to write code that correctly handled each exception. The nonprogramming experiment required only that participants generate lists of exception conditions that they thought would be raised if the program were to be written; it was a thought experiment, not a coding experiment. It was conducted to provide convergent evidence showing that programmers in the treatment group were influenced mentally to generate or consider more exception conditions than their peers in the control group.

5.1. Results: Nonprogramming-experiment

The nonprogramming experiment included 38 programmers (group D), divided into 23 control cases and 15 treatment cases, all of whom were given the same materials as the programmers in the programming experiment. These nonprogramming participants were instructed to write down all the exception failures they could think of, without writing any code.

The statistical results of the nonprogramming experiment are displayed in Table 5-1. One-tailed and two-tailed t-tests were used, because these are the simplest effective statistical analyses for determining the significance of the difference between two groups of scores. Note that there are no special scoring levels employed here, because only the number of exceptions listed is being analyzed. The single-tailed t-test is significant at the .001 level, and the stronger 2-tailed t-test is significant at .002; both tests indicate a substantial effect of treatment over control. Significance at the .001 level means that these results could have been obtained by chance with a probability of only .001.

The 79.1% improvement of treatment over control conditions, from a mean score of 4.65 to a mean score of 8.33, is remarkable. This very strong result indicates that using dependability cases to help think about exceptions is almost certainly useful in expanding exception coverage. Note, however, that being aware of exceptions

Treatment		Control		T-test
Mean	Std	Mean	Std	p-value
8.33	4.34	4.65	2.33	0.001*

Table 5-1: Nonprogramming results: 15 treatment cases, 23 control cases, 1-tailed t-test; * indicates statistical significance at the .001 level.

that need to be handled in a program does not particularly mean that they will in fact be handled correctly. The programming-experiment results address this latter issue.

5.2. Results: Programming-experiment

The programming experiment included 59 programmers, all of whom wrote code according to the same specification; 33 of these programmers were in the control group, and had no special instructions other than to be sure that their programs worked; 26 of the programmers were in the treatment group, and used dependability cases to increase their awareness of exception failures. The results displayed in this section are pooled so as to reduce uncertainty in estimating the standard deviation used in the test statistic.

An analysis of variance (two-way ANOVA with interaction) was performed on the pooled data for all 59 participants. The analysis found no significant interaction between treatment and group ($p = 0.6883$), meaning that the three groups of programmers (groups A, B and C) did not perform differently from one another due to any effect of the treatment. An ANOVA without interaction, however, shows statistically significant differences in group effects ($F = 4.24$, $p = 0.0194$) and in treatment effects ($F = 5.97$, $p = 0.0178$), meaning that there were real differences among the three groups, and that the treatment was, with 98.22% confidence, effective in changing the overall behavior of the treatment groups with respect to the control groups. On the whole, the treatment groups performed 43% better than the control groups in terms of covering exception failures. Details are given below.

Figure 5-1 shows the total number of test cases that were handled correctly under the two different scoring levels. The graph clearly demonstrates that there are positive differences between the control and treatment groups in both scoring conditions, but the largest difference (and the best overall performance) is in the exception scoring condition. Exception scoring is the condition that most emphatically illustrates exception-handling adequacy. In this condition 43.2% of all test cases (normal plus fault) were handled correctly by control-group programs, and 59.2% were handled correctly by treatment-group programs.

The data in Table 5-2 show the means and standard deviations for control and treatment programs under both scoring levels, as well as the single-tailed t-test p-value for each scoring level. Although both scoring levels show an improvement of the treatment group over the control group, only the exception score shows strong statistical

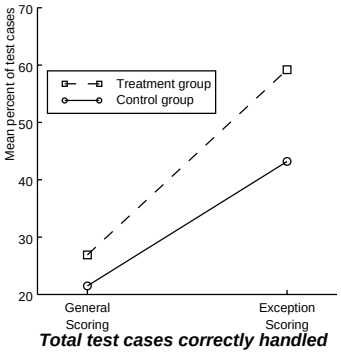

Figure 5-1: Percent of all 25 test cases handled correctly under different scoring levels. Exception scoring shows 37.3% improvement of treatment over control.

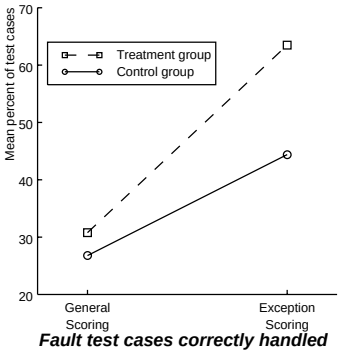

Figure 5-2: Percent of fault test cases handled correctly under different scoring levels. Exception scoring shows 43% improvement of treatment over control.

significance, as indicated by the asterisk. The estimated magnitude of the treatment effect is a 37.3% improvement over the control condition. This effect is statistically significant at the .017 level.

Scoring Level	Treatment Mean	Treatment Std	Control Mean	Control Std	T-test p-value
General	6.73	6.74	6.03	6.62	0.346
Exception	14.81	6.26	10.79	7.65	0.017*

Table 5-2: All 25 test sets, pooled results: 26 treatment cases, 33 control cases, 1-tailed t-test; * indicates statistical significance at the .017 level.

Evaluating the programs against *all* test cases means that programs are being tested for two things: their ability to function correctly on normal cases that would not be expected to generate exceptions, *and* their ability to handle exceptions generated by fault cases. The focus of this paper is on exceptions, however, not normal conditions, so it seems reasonable to isolate the results of testing with fault cases from the results of testing with normal cases. This will reveal control and treatment differences on fault cases alone, and hence show results that are closer to the heart of the study. Of the 25 total test cases, 12 were fault (expected to generate exceptions) and 13 were normal. Figure 5-2 shows the numbers of strictly fault cases correctly handled by control and treatment groups. The figure demonstrates a trend similar to that shown in Figure 5-1 where performance is indicated over all 25 test cases, but the trend is now more pronounced. Again, exception scoring most emphatically illustrates exception-handling adequacy. In this condition, 44.4% of all fault cases were handled correctly by control-group programs, and 63.5% of all fault cases were handled correctly by treatment-group programs.

The data in Table 5-3 show the means and standard deviations for control and treatment programs for both scoring levels, including the t-test p-values. The estimated magnitude of the treatment effect is a 43% improvement over the control condition. This effect is statistically significant at the .01 level.

Scoring Level	Treatment Mean	Treatment Std	Control Mean	Control Std	T-test p-value
General	3.69	3.51	3.21	3.25	0.294
Exception	7.62	3.50	5.33	3.81	0.011*

Table 5-3: Fault test sets only: pooled results, 26 treatment cases, 33 control cases, 1-tailed t-test; * indicates statistical significance at the .01 level.

6. Summary and conclusion

Programmers do not always cover all the exception conditions encountered by programs. This paper asked: What is the reason for this, and what can be done about it? It was hypothesized that programmers ignore some exception cases because they forget. It was further hypothesized that if programmers were provided with a structure that made it easier to remember or to generate exception categories and exemplars, then they would achieve better coverage in programs. This prospective approach would have advantages over retrospective approaches such as testing, because avoiding problems in the first place is almost always better than detecting and fixing problems later.

Two experiments were conducted to validate the hypotheses above. A nonprogramming experiment demonstrated that, with the structuring methods prescribed in dependability cases, programmers con-

sidered 79.1% more exception cases than they did without them. This result was statistically significant at the .001 level. A programming experiment demonstrated that, using the same structuring methods, programmers cover 43% more exception conditions than they do without using the method. This result was statistically significant at the .01 level. At the same time as exception coverage improved, programmers also improved by 37.3% their coverage of purely functional aspects of programs. The structure of the dependability case did not appear to improve adherence to directives specifying output formats, error-message usefulness or return-code correctness.

It seems clear that dependability cases, by mere virtue of being constructed, can expose problems and can prompt system designers to think about how to avoid them. A very positive contribution that dependability cases make to system robustness is to provide a framework for thinking about what can go wrong, and how to avoid or handle failures. The structural elements provided by some of the methodologies intrinsic to safety cases, dependability cases and quality control (e.g., hazard analyses, fault trees and Ishikawa diagrams) appear to be ideal for helping programmers recall or generate lists of exception conditions that need to be covered in a given program. The investment in learning and employing the dependability-case methodology is minimal (about 20 minutes in the simple experiments conducted here).

Given the effectiveness and simplicity of dependability cases, it is suggested that even such a small thing as teaching the mnemonic "Everyone loves exceptional *children*" to beginning programmers might be an efficacious way to improve exception-handling coverage in the long term. According to skilled-memory theory, the problem of remembering information learned and stored in long-term memory can be solved by associating the stored information with a retrieval cue or structure [7]. The structure inherent in dependability cases accomplishes just that. Being able to recall or regenerate a comprehensive list of cases to guard against must certainly be an improvement over forgetting.

7. Future

The present study begs for replication, and this is planned on a broader scale in the next academic year. A wider range of programming tasks should be used, as well as a more complete set of test cases. More work is needed to tease apart the individual effects of the hazard analysis, the fault tree and the fishbone diagram. It is possible that one of these structures carries more of the effect found in the present experiments, and it would be useful to discover which one it is.

8. Acknowledgements

It is a pleasure to acknowledge the contributions of David Banks, Kobey DeVale, Fred Harris, Phil Koopman (for the *children* mnemonic), Patty McMillan, Gary Powers, Dan Siewiorek, Mark Stehlik, James Tomayko, Ed Wishart, and all the students from the Fall 1997 graduate course in Dependable System Design at Carnegie Mellon University.

References

[1] Bertin, Jacques, *Semiology of Graphics*, University of Wisconsin Press,, Madison, Wisconsin, 1983.

[2] Bishop, P.G. and Bloomfield, Robin E., ‘‘The SHIP Safety Case Approach’’, In *SafeComp-95: 14th International Conference on Computer Safety, Reliability and Security*, Gerhard Rabe (Ed.). European Workshop on Industrial Computer Systems Technical Committee 7 - Reliability, Safety, and Security. Berlin: Springer-Verlag, 11-13 October 1995, pp. 437-451, Belgirate, Italy.

[3] Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[4] Bush, Marilyn, ‘‘Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory’’, In *12th International Conference on Software Engineering*. Los Alamitos, California: IEEE Computer Society Press, 26-30 March 1990, pp. 196-199, Nice, France.

[5] Computer Emergency Response Team (CERT), Personal communication. CMU Software Engineering Institute, Pittsburgh, Pennsylvania: October 1997.

[6] Cristian, Flaviu, Exception Handling and Tolerance of Software Faults, In *Software Fault Tolerance*, Michael R. Lyu (Ed.). Chichester: Wiley, 1995. pp. 81-107, Ch. 4.

[7] Ericsson, K. Anders and Pennington, Nancy, The Structure of Memory Performance in Experts: Implications for Memory in Everyday Life, In *Memory in Everyday Life*, Graham M. Davies and Robert H. Logie (Eds.). Amsterdam: Elsevier North-Holland, 1993. pp. 241-282, Ch. 6.

[8] Ishikawa, Kaoru, *Guide to Quality Control*, Asian Productivity Organization, Tokyo, 1982.

[9] James, William, *The Principles of Psychology*, Henry Holt and Company, New York, 1890.

[10] Kit, Edward, *Software Testing in the Real World: Improving the Process*, Addison-Wesley, Harlow, England, 1995.

[11] Kletz, Trevor, *Hazop and Hazan: Identifying and Assessing Process Industry Hazards*, Institution of Chemical Engineers, Rugby, Warwickshire, England, 1992.

[12] Leveson, Nancy G., *Safeware: System Safety and Computers*, Addison-Wesley, Reading, Massachusetts, 1995.

[13] Lions, Jacques-Louis, *ARIANE 5 Flight 501 Failure: Report by the Inquiry Board*, World-Wide Web, http://www.esrin.esa.it /htdocs/tidc/Press/Press96/ariane5rep.html, 1996, (Hardcopy available from Maxion & Olszewski, authors of this paper.)

[14] Reason, James, *Human Error*, Cambridge University Press, Cambridge, England, 1990.

[15] Shaw, Roger, editor, *Safety and Reliability of Software Based Systems*, Springer Verlag, Berlin, 1997.

[16] Shaw, Roger, ‘‘Safety Cases - How Did We Get Here?’’, In *Safety and Reliability of Software Based Systems*, Roger Shaw (Ed.). Berlin: Springer-Verlag, (CSR Workshop held 12-15 September 1995). 1997, pp. 43-95, Brugge (Bruges), Belgium.

[17] Toy, Wing N., Fault-tolerant Design of Local ESS Processors, In *The Theory and Practice of Reliable System Design*, Daniel P. Siewiorek and Robert S. Swarz (Eds.). Bedford, Massachusetts: Digital Press, 1982. pp. 461-496, Ch. 12.

[18] Vesely, W.E.; Goldberg, F.F.; Roberts, N.H. and Haasl, D.F., *Fault Tree Handbook*, U.S. Nuclear Regulatory Commission, Washington, D.C., 1981, Technical Report NUREG-0492.

[19] Wickelgren, Wayne A., *Learning and Memory*, Prentice Hall, Englewood Cliffs, New Jersey, 1977.