

USING A SOFTWARE TEST AND ANALYSIS TOOL FOR TEST CASE DEVELOPMENT

Richard Parry, C.C.P, P.E.

email address:
rparry@qualcomm.com

Home Web Page:
<http://people.qualcomm.com/rparry>

wireless packet address:
W9IF @ K6JCC.#SOCAL.CA.USA.NA

ABSTRACT

This paper describes how a software code coverage analysis tool can be used to aid software testing. The children's card game, "Fish" was used to illustrate the process. Although the program is simple, it serves to show how code coverage analysis can be used for larger, more complex programs. Each function of the "C" code program is carefully analyzed to maximize code coverage. Test cases are developed to exercise the un-executed code. The tool used for coverage analysis is *VisionSoft*, published by the VisionSoft Corporation. The tool provides many code and management features. The paper concentrates on the code coverage capability of the tool.

March, 1997

A program that has not been tested does not work. The ideal of designing and/or verifying a program so that it works the first time is unattainable for all but the most trivial programs. We should strive toward that idea, but we should not be fooled into thinking that testing is easy.

Bjarne Stroustrup, *The C++ Programming Language*

INTRODUCTION

June 4, 1996, all systems are go here in Kourou, French Guiana.

T minus five ... four ... three ... two ... one!

We have ignition and lift off. At 10 seconds into the flight the rocket is 4 miles down range. It's 20 seconds after launch, the rocket is down range 12 miles. At 30 seconds into the flight all systems are nominal. At 39 seconds into the flight, there is a brilliant flash, and then silence ...

That flash of light in the sky was a \$7 billion dollar software defect. The rocket was an Ariane 5, manufactured by the European Space Agency over a 10 year period. The Ariane 5 was intended to give Europe overwhelming supremacy in the commercial space business, but 39 seconds into the flight that hope ended. A later in-depth analysis showed that a small computer program tried to stuff a 64-bit value into the space intended to fit a 16-bit value. The result of that error ended in a ball of flames.

Unfortunately, many other errors like this have occurred with equally disastrous results. As we put computers in more and more applications, the likelihood of disasters are likely to increase if something is not done.

In the not to distant past, software development was a simple two step process, *code* and *test* with emphasis on the former, rather than the latter. However, the proliferation of software into critical applications that span life critical safety systems to multi-billion dollar control systems has changed the way software development is performed. Various methodologies have been developed with more emphasis on every aspect of the software development process, extending from requirements to deployment, with particular attention given to testing. However, just as software is as much an art as it is a science, the same can be said for testing. Fortunately, powerful test tools are now available to aid the software developer at the unit test level. But even with these tools, the test engineer must carefully use these tools as an aid, not as a panacea.

In this paper we will use a commercial software coverage tool, to analyze a simple program to gain a better understanding of how software coverage tools work and how they can be used to develop test cases that help insure maximum code coverage.

CODE COVERAGE, WHAT IT IS AND HOW IT WORKS

In this section we will gain a better understanding of code coverage tools; how they operate, and what they can do to aid the programmer. Most importantly, how they can be used to aid the software test engineer in developing test cases. Toward that end, a specific product, *VisionSoft*, will be used.

From the code coverage metrics provided by *VisionSoft*, new test cases will be developed to help maximize code coverage. *VisionSoft* provides error detection by providing information about what portions of the code were not exercised during unit level testing. This allows the programmer to test new code before moving to the next generation of code. This is important since testing small portions of code is easier than checking large sections. This ability has the additional benefit of saving the programmer from re-testing code that has not changed since all of the tested code is "brought forward" to the new version of the application. It is important to note that *VisionSoft* works on the source code rather than the object code. We will see in the next section exactly how this is done. However, this is an important feature in many applications since it allows the user to use her/his favorite compiler, debugger, and any

other tools without affecting the manner in which the coverage tool works. As we shall see, some code is difficult to exercise. In these cases setting breakpoints with a debugger will allow the tester to force hard to exercise code to be executed without affecting the accuracy of the test.

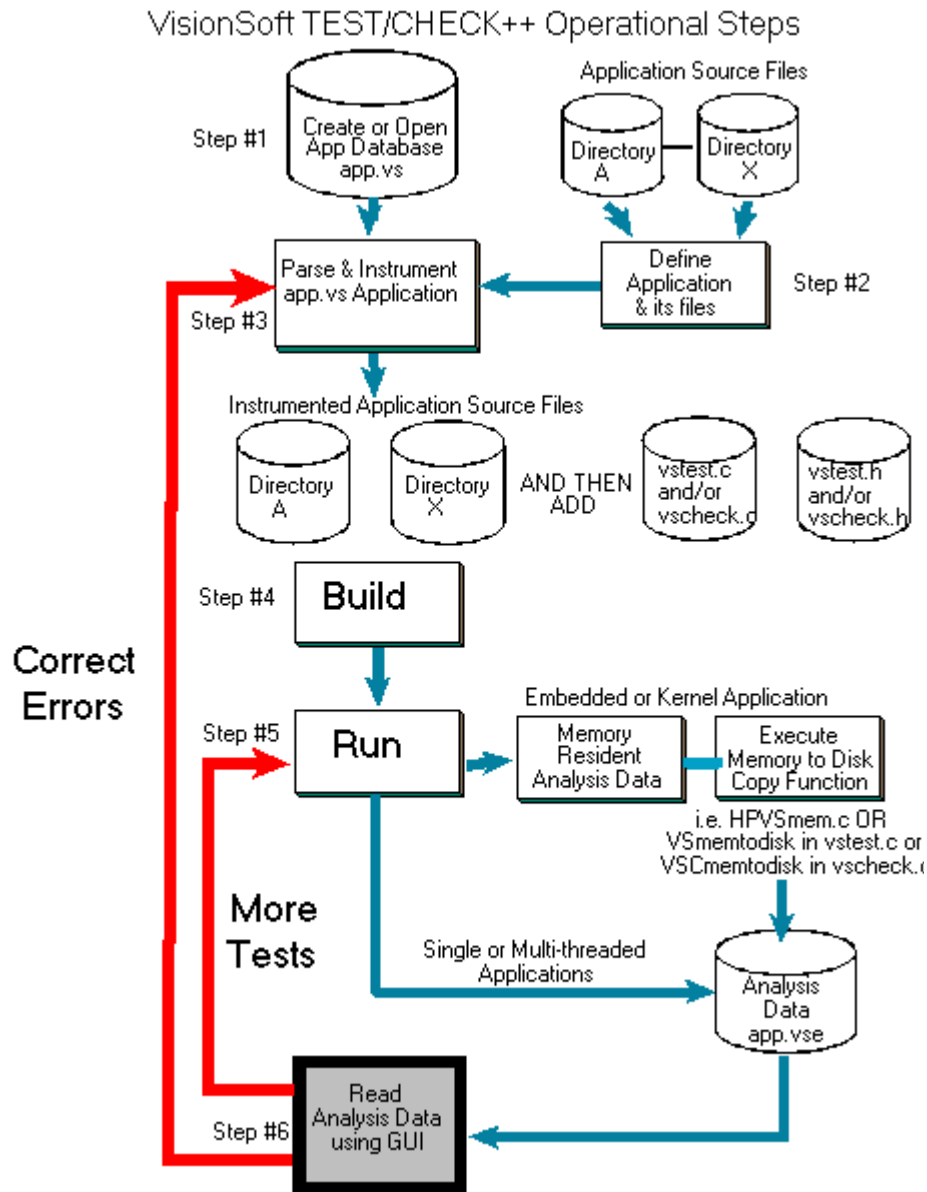


Figure 1 Code Coverage using VisionSoft

Figure 1 shows a high level view of the code coverage process. It begins with instrumenting the original source code, a simple process which takes just moments. Once the code has been instrumented, a new source file is created. This new source file is then compiled in the normal manner. From there, the object file is linked if necessary and executed. As the application is executed, a database is generated which can then be examined.

Code Instrumentation

The process of preparing the source code so that it can generate metrics is called *instrumenting*. During the instrumentation process, additional code is added to insure that the database generated provides the desired information. For example, a user may wish to ascertain merely that functions are executed, loosely speaking this is called *edge coverage*. The tool provides for many types of coverage which we will discuss in greater detail in subsequent sections.

While an in-depth understanding of the code coverage tool is not essential, a basic understanding is very important. In fact, if the user wishes to gain the most from the tool, a thorough understanding is required. The application and the platform under which code coverage will be executed will determine, in-part, how much expertise is required. The following simple program will illustrate instrumentation.

Table 1 Original Source Code

```
#include <stdio.h>

int main()

{
    printf("hello, world\n");
    printf("hello, world\n");
    printf("hello, world\n");
}
```

When the source code is instrumented, additional statements, sometimes called probes, are added that will determine if the portion of the code which the probe has tagged is covered. These additional statements will create a database that will eventually be used by the coverage tool to ascertain if the code segment was executed. These additional lines of code will clearly affect the size and speed at which the code operates. In real-time embedded systems where there are often time and size constraints, the user will have to judiciously select the type of coverage. If speed is critical, the user may select individual statement types to be instrumented to reduce instrumentation overhead.

The new instrumented code is shown below. The instrumentation type was set for edge coverage. The code below shows additional keywords that are specific to the coverage tool used in this paper. Note that two additional header files are declared which define the proprietary commands used to instrument the code. There are also two additional “C” files that are compiled along with the newly instrumented code.

Table 2 Edge Instrumented Source Code

```
#include <stdio.h>

#include "vstest.h" /* inserted 02/24/97 08:32:48 AM */
#include "vscheck.h" /* inserted 02/24/97 08:32:48 AM */
static char VSFAR VSfileopt4096[ ]="hello.vse,s,c,e,o";
int main()

{VSP(2,1,6123L,&VSfileopt4096[0]); {
    printf("hello, world\n");
    printf("hello, world\n");
    printf("hello, world\n");
} }
```

Here again is the same code but in this case it is instrumented for both Edge and Statement coverage.

Table 3 Edge and Statement Instrumented Source Code

```

#include <stdio.h>

#include "vstest.h"    /* inserted 02/19/97 11:38:40 AM */
#include "vscheck.h"  /* inserted 02/19/97 11:38:40 AM */
static char VSFAR VSfileopt4096[ ]="hello.vse,s,c,e,b,l,o";
int main()

{VSP(2,1,622161893L,&VSfileopt4096[0]); {
  VSB(2,2,622161893L);printf("hello, world\n");
  VSB(3,2,622161893L);printf("hello, world\n");
  VSB(4,2,622161893L);printf("hello, world\n");
} }

```

There is more to the instrumentation process, however, this brief discussion is sufficient to allow us to proceed and for the reader to appreciate the results of the analysis and how it is generated.

Coverage Types¹

VisionSoft supports statement, selective, edge, branch, and limits test coverage. The analysis is accomplished by instrumenting the original program as described in the previous section. As the program executes, the coverage data is collected in an execution database which is eventually read by the *VisionSoft* application and displayed for the developer to analyze.

Statement Coverage is a tabulation of the statements in an application, a file or function that has been executed at least once. As program execution proceeds, only the first occurrence of a statement executing is recorded to limit database size and because in code coverage analysis we are interested in knowing only that the line was executed, not how many times. However, results are accumulated if the program executes multiple times.

Selective Coverage is a tabulation of a subset of statements in an application, a file or function that has been executed at least once. The user selects the statement types to be analyzed. This coverage is often used to limit the size of the resulting instrumentation code and to reduce degradation in execution speed that instrumentation normally causes.

Branch Coverage refers to the analysis of the clauses within an *if* statement. Each *if* statement may contain one or more clauses. *If* clauses are *or'ed* during execution to determine whether the *if* statement is true or false.

Limits Coverage is similar to *branch* coverage except the clauses being analyzed are contained in *for* and *while* statements.

Edge Coverage is a form of selective coverage in which the statements that change the program instruction sequence are monitored. The statement types monitored are *if*, *goto*, *return*, *switch*, *case*, *for*, *while* and *default* plus the *if* clauses, *for* and *while* limits. This coverage is patterned after and extends the McCabe complexity metric.

For our analysis of the application program discussed in this paper, we will use both *edge* and *statement coverage*. Edge coverage has the advantage of including both *branch* and *limits* analysis. In addition, since we are not limited by either size or speed and our intention is to exercise all the code, we add *statement coverage* for a more complete analysis.

¹ Definitions described in this section are summarized from VisionSoft documentation.

McCabe Complexity²

McCabe complexity is a count of all the edge statements in an application file or function. The edge statements are counted and the analysis performed according to the following formula:

```
The computation is 1 + number of break statements
                    + number of continue statements
                    + number of if statements
                    + number of while statements
                    + number of goto statements
                    + number of return statements
                    + number of case statements
                    + number of default statements
                    + (number of function end statements-1)
```

Normalized McCabe complexity extends standard McCabe analysis and provides a percentage based complexity calculation relative to the size of a given file or function according to the following formula:

$$100 * \{MCCABE() / st.execstmts\}$$

Where *st.execstmts* is the number of executable statements in an application, file or function.

Object Size and Performance

One may be tempted to instrument code for all available coverage types. However, the coverage type needs to be weighed against other factors since the instrumentation process affects both size and performance. In other words, there is a cost that one pays for instrumenting code. Below is an estimate of how the object size changes based on the coverage type.³

- Edge Coverage20-50%
- Selective Coverage20-100%
- Statement Coverage 75-100%
- Branch Coverage..... 10-30%
- Limits Coverage..... 5-20%
- Program Tracing 75-100%

Reduced execution speed is another price that one pays to instrument code. The performance impact of the instrumentation varies in three ways.⁴

- First, the performance impact is one conditional “if” statement per instrumentation call plus a file open occurs every 20 seconds to look for an activation message.
- Second, the type of analysis and the statements, functions and files instrumented will cause variations. However, for every function called that is instrumented, at least one disk read and possibly one write will be needed if new analysis results were generated while the function is executed.

² Explanation and equations provided from VisionSoft documentation.

³ List taken from VisionSoft documentation.

⁴ The list of three performance considerations summarized from the VisionSoft manual.

- Third, the instrumentation options object size and performance affect the overhead. The object size optimization will minimize the number of instrumentation calls within a function and reduce the CPU time but not the disk read/writes. Performance optimization means that new analysis results generated while a function executes will only be written to disk when the function is exited.

GO FISH

The program to be used is our example is *Fish*, the childhood card game. Although the analysis is relatively simple, it provides an excellent illustration to show how a code coverage tool may be used in a real world software development project. One of the strongest features of the program is its ability to provide the software tester with a clearly defined list of areas that need to be studied .

A complete understanding of the game is not necessary for an appreciation of the test tool, however, the remainder of this paper assumes a nodding knowledge of the game and some of the rules. The instructions are summarized here.

This is the traditional children's card game "Go Fish". We each get seven cards, and the rest of the deck is kept to be drawn from later. The object of the game is to collect "books", or all of the cards of a single value. For example, getting four 2's would give you a "book of 2's".

We take turns asking each other for cards, but you can't ask me for a card value if you don't have one of them in your hand! If I have any cards of the value you ask for, I have to give them to you. As long as I have one of the cards you ask for, you get to keep asking. If you ask me for a card of which I don't have any, then I'll tell you to "Go Fish!" This means that you draw a card from the deck. If you draw the card you asked me for, you get to keep asking me for cards. If not, it's my turn and I ask you for a card.

Sometimes you get to ask first, sometimes I do. I'll tell you when it's your turn to move, I'll draw cards from the deck for you, and I'll tell you what you have in your hand. (Don't worry, I don't look at your hand when I'm trying to decide what card to ask for, honest!)

Your input can be a card name ("A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q" or "K") or the letter "p", or "quit". The letter "p" makes my game much smarter, and the line "quit" stops the game. Just hitting the carriage return key displays how many cards I have in my hand, how many are left in the deck, and which books I've gotten.

Normally, the game stops when one of us runs out of cards, and the winner is whoever has the most books!

Good luck!

RUN 1

Before we begin the analysis we must first state our goal. The instrument type will be determined by our requirements and/or restrictions (e.g., size and speed). As we have previously mentioned, we wish to maximize edge and statement coverage and have instrumented the code to meet this requirement.

After compilation we execute the program to provide a baseline test case. During the first run no adversarial testing is performed. By this we mean that no attempt shall be made to "break" the code or use illegal moves. The purpose is not to win or lose, but to provide input within the normal range of a

standard deck of cards. The interaction for this first run between the user and the computer program is included in the appendix.

Table 4 shows the metrics provided by *VisionSoft*. Note the use of color to indicate the functions that are not covered. As we will see in subsequent sections, the tool will allow the user to see exactly which lines of code were executed and which were not using color.

Table 4 Results of Run 1

| /home/rparry/C/fish/fish.c 02/19/97 11:48:41 AM C file | | | | | |
|--|---|---|--|---|--|
| Program Units | Statistics | Complexity | Functions | Edges | Statements |
| /home/rparry/C/fish/fish.c | 430 source lines 390 statements 288 executables | Metric(normalized,value) McCabe(39%,113) Edges(37%,109) | 3 unused, 13 used COMPLETE 81% | 90 of 148 edges executed INCOMPLETE 60% | 200 of 288 used, 0 excluded INCOMPLETE 69% |
| GLOBAL area | 71 source lines 6 statements | | | | |
| chkwinner | 28 source lines 35 statements 29 executables | Metric(normalized,value) McCabe(27%,8) Edges(27%,8) | EXECUTED | 9 of 13 edges executed INCOMPLETE 69% | 21 of 29 used, 0 excluded COMPLETE 72% |
| compmove | 16 source lines 18 statements 13 executables | Metric(normalized,value) McCabe(30%,4) Edges(30%,4) | EXECUTED | 5 of 6 edges executed COMPLETE 83% | 12 of 13 used, 0 excluded COMPLETE 92% |
| countbooks | 15 source lines 16 statements 12 executables | Metric(normalized,value) McCabe(41%,5) Edges(41%,5) | EXECUTED | 6 of 7 edges executed COMPLETE 85% | 11 of 12 used, 0 excluded COMPLETE 91% |
| countcards | 9 source lines 7 statements 5 executables | Metric(normalized,value) McCabe(60%,3) Edges(60%,3) | NOT EXECUTED | 0 of 3 edges executed INCOMPLETE 0% | 0 of 5 used, 0 excluded INCOMPLETE 0% |
| drawcard | 21 source lines 24 statements 17 executables | Metric(normalized,value) McCabe(29%,5) Edges(29%,5) | EXECUTED | 8 of 8 edges executed COMPLETE 100% | 17 of 17 used, 0 excluded COMPLETE 100% |
| gofish | 15 source lines 14 statements 11 executables | Metric(normalized,value) McCabe(36%,4) Edges(36%,4) | EXECUTED | 4 of 4 edges executed COMPLETE 100% | 11 of 11 used, 0 excluded COMPLETE 100% |
| goodmove | 22 source lines 16 statements 13 executables | Metric(normalized,value) McCabe(15%,2) Edges(15%,2) | EXECUTED | 2 of 2 edges executed COMPLETE 100% | 13 of 13 used, 0 excluded COMPLETE 100% |
| init | 17 source lines 24 statements 16 executables | Metric(normalized,value) McCabe(37%,6) Edges(37%,6) | EXECUTED | 8 of 10 edges executed COMPLETE 80% | 16 of 16 used, 0 excluded COMPLETE 100% |
| instructions | 16 source lines 18 statements 13 executables | Metric(normalized,value) McCabe(38%,5) Edges(38%,5) | EXECUTED | 4 of 7 edges executed INCOMPLETE 57% | 8 of 13 used, 0 excluded INCOMPLETE 61% |
| main | 50 source lines 57 statements 41 executables | Metric(normalized,value) McCabe(41%,17) Edges(36%,15) | EXECUTED | 14 of 19 edges executed COMPLETE 73% | 33 of 41 used, 0 excluded COMPLETE 80% |
| nrandom | 7 source lines 4 statements 2 executables | Metric(normalized,value) McCabe(100%,2) Edges(100%,2) | EXECUTED | 1 of 1 edges executed COMPLETE 100% | 2 of 2 used, 0 excluded COMPLETE 100% |
| printhand | 18 source lines 23 statements 18 executables | Metric(normalized,value) McCabe(36%,7) Edges(36%,7) | EXECUTED | 12 of 12 edges executed COMPLETE 100% | 19 of 19 used, 0 excluded COMPLETE 100% |
| printplayer | 12 source lines 11 statements 8 executables | Metric(normalized,value) McCabe(62%,5) Edges(37%,3) | EXECUTED | 3 of 3 edges executed COMPLETE 100% | 8 of 8 used, 0 excluded COMPLETE 100% |
| promove | 37 source lines 48 statements 36 executables | Metric(normalized,value) McCabe(55%,20) Edges(55%,20) | NOT EXECUTED | 0 of 30 edges executed INCOMPLETE 0% | 0 of 36 used, 0 excluded INCOMPLETE 0% |
| usage | 5 source lines 4 statements 3 executables | Metric(normalized,value) McCabe(33%,1) Edges(33%,1) | NOT EXECUTED | 0 of 0 edges executed | 0 of 3 used, 0 excluded INCOMPLETE 0% |
| usermove | 53 source lines 64 statements 50 executables | Metric(normalized,value) McCabe(38%,18) Edges(38%,18) | EXECUTED | 14 of 24 edges executed INCOMPLETE 58% | 29 of 50 used, 0 excluded INCOMPLETE 58% |

Table 5 summarizes the results of the first run. An examination indicates that the coverage results can be broken into four categories. The first category contains functions 1, 2, and 3. These functions were not executed with the result that none of the edges or statements were exercised. These are the critical portions that need to be analyzed first. The second group consisting of 4, 5, and 6, are functions that were executed, but the number of edges exercised fell below 70% which is the standard

used by *VisionSoft* as a threshold to make a determination as to whether the code has been adequately covered. The third group contains functions 7, 8, 9, and 10. These were executed and considered complete since more than 70% of the edges were executed. Nevertheless, this code should be studied to ascertain if additional test cases can be developed to increase this even more. The fourth category is comprised of functions that were executed completely. No examination of these six functions will be performed.

Table 5 Results of Run 1 (Function Summary)

| Item | Function | Status | Edges | Statements |
|------|--------------|--------------|-----------|------------|
| 1 | countcards | Not Executed | 0 (<70%) | 0 (<70%) |
| 2 | promove | Not Executed | 0 (<70%) | 0 (<70%) |
| 3 | usage | Not Executed | 0 (<70%) | 0 (<70%) |
| 4 | chkwinner | Executed | 69 (<70%) | 72 (>70%) |
| 5 | instructions | Executed | 57 (<70%) | 61 (<70%) |
| 6 | usermove | Executed | 58 (<70%) | 58 (<70%) |
| 7 | compmove | Executed | 83 (>70%) | 92 (>70%) |
| 8 | countbooks | Executed | 85 (>70%) | 91 (>70%) |
| 9 | init | Executed | 80 (>70%) | 100 (>70%) |
| 10 | main | Executed | 73 (>70%) | 80 (>70%) |
| 11 | drawcard | Executed | 100% | 100% |
| 12 | gofish | Executed | 100% | 100% |
| 13 | goodmove | Executed | 100% | 100% |
| 14 | nrandom | Executed | 100% | 100% |
| 15 | printhead | Executed | 100% | 100% |
| 16 | printplayer | Executed | 100% | 100% |

VisionSoft provides detailed individual metrics for each of the functions. A summary of all the functions can be stated succinctly as shown in Table 6.

Table 6 Results of Run 1 (High Level Summary)

| | |
|--------------------------------|----|
| functions used 13 of 16 | 81 |
| edges executed 90 of 149 (%) | 60 |
| statements used 200 of 288 (%) | 69 |

NON-EXECUTED FUNCTIONS

This section of the paper discusses the three functions that were never called during the execution of the program. A careful analysis of why these functions were not called is important since they may contain defective code.

countcards function

The *countcards* function was never called. An analysis of the code shows that this function appears twice, both times within the *usermove* function. Therefore to force the program to exercise this portion of the code an examination of the calling routine is necessary.

Table 7 critical section of usermove function

```

if (buf[0] == '\n') {
    (void)printf("%d cards in my hand, %d in the pool.\n",
        countcards(comphand), countcards(deck));
    (void)printf("My books:");
    (void)countbooks(comphand);
    continue;
}

```

The code shows that the function *countcards* will be called if we input a null string. In other words, during the initial non-adversarial test no attempt was made to input values other than those found in a standard deck of cards ("A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q" or "K"). Therefore to test the additional code we need to input a null string or in this case, just hit the ENTER key when asked to select a card.

```

countcards 02/19/97 of /home/rparry/C/fish/fish.c
File View Search Trace Notes Windows Help
*00001 countcards(hand)
*00002     register int *hand;
*00003 {
00004     register int i, count;
00005
*00006     for (count = i = 0; i < RANKS; i++)
*00007         count += *hand++;
*00008     return(count);
*00009 }
00010

```

Figure 2 countcards function

- Add Test Case
 - Input ENTER key on keyboard, do not input a data value

It is worthwhile to note that the next time we play the game and input ENTER, this only assures that the function will be called, it does not assure us that all the edges and statements will be covered. For this reason, subsequent test cases may need to be developed.

promove function

The *promove* function was never called, it appears only once in the *compmove* function.

```

if (promode)
    lmove = promote();
else {

```

In order for the *promove* function to be called we need to enter the professional mode (*promode*). An examination of the instructions and/or code shows that the user must input "p" to enter the professional mode which makes the computer a *smarter* player.⁵

⁵ This most likely means that the computer remembers what cards the user requested.

```

promove 02/19/97 of /home/rparry/C/fish/fish.c
File View Search Tools Notes Windows Help
*00001 promove()
*00002 {
00003     register int i, max;
00004
*00005     for (i = 0; i < RANKS; ++i)
*00006         if (userasked[i] &&
00007             comphand[i] > 0 && comphand[i] < CARDS) {
*00008             userasked[i] = 0;
*00009             return(i);
00010         }
00011     if (nrandom(3) == 1) {
*00012         for (i = 0; ++i)
00013             if (comphand[i] && comphand[i] != CARDS) {
*00014                 max = i;
*00015                 break;
00016             }
*00017         while (++i < RANKS)
*00018             if (comphand[i] != CARDS &&
*00019                 comphand[i] > comphand[max])
*00020                 max = i;
*00021         return(max);
00022     }
00023     if (nrandom(1024) == 0723) {
*00024         for (i = 0; i < RANKS; ++i)
*00025             if (userhand[i] && comphand[i])
*00026                 return(i);
00027     }
00028     for (;;) {
*00029         for (i = 0; i < RANKS; ++i)
*00030             if (comphand[i] && comphand[i] != CARDS &&
*00031                 !asked[i])
*00032                 return(i);
*00033         for (i = 0; i < RANKS; ++i)
*00034             asked[i] = 0;
00035     }
00036     /* NOTREACHED */
*00037
00038

```

Figure 3 promove function

- Add Test Case
 - Select professional mode, hit “p” key on keyboard

Note that when we go to the professional mode, the likelihood of the user winning is decreased, which also means that some other portion of the code may not be executed. So while the *promove* function will be called during the new test run, other lines may be skipped. For this reason, it is important to keep a running total of what portions of the code have been exercised. Fortunately, the tool keeps a running total. Therefore, in subsequent runs, we may actually test less code, but as long as new lines of code are tested, the coverage metric will continue to rise.

usage function

The *usage* function was also never called. A *grep* of the source file shows that the function never appears. The word *usage* appears three times, however, twice are in normal use of the word (standard English usage), and once for the declaration of the function.

```
usage 02/19/97 of /home/rparry/C/fish/fish.c
File View Search Trace Notes Windows Help
*00001 usage()
*00002 {
*00003     (void)fprintf(stderr, "usage: fish [-p]\n");
*00004     exit(1);
*00005 }
00006
```

Figure 4 usage function

- Add Test Case
 - **Error, this function can never be called!**

This appears to be a software defect. An examination of the code indicates the function was meant to be called when an input error is detected. There is no way to force this code to be executed other than using a debugger. A visual examination indicates the function is innocuous, nevertheless it should be removed.⁶

EXECUTED, INCOMPLETE EDGES (< 70% COVERAGE)

Unlike complete functions that were never called, discussed in the previous section, here we turn our attention now to three functions that were partially executed. *VisionSoft* considers code that has less than 70% code coverage to be insufficiently tested, therefore an analysis of these functions is important.

checkwinner function

The *checkwinner* function was called with 9 of 13 edges (69%) executed. This function is called to determine a winner. There is always a winner (or tie) so it should be obvious that the function is always called. However, depending on who won will determine what portion of the code is executed. Unfortunately, winning is easier said than done, especially if the program is placed in the professional mode. In addition, a line by line analysis of the program shows that a random number determines some paths. In the event the value is equal to that specified by the *if* statement, a computer response will be provided to the user. To force this code to be exercised, the software tester can change the code so that the value is fixed to assure the code will be executed. However, this is a bad practice since altering the code may inject other errors. The easiest way to force this portion of the code to be exercised is to use a debugger. A debugger enables the tester to force the value of the random variable so that the desired code segment is executed.

⁶To prove that the code was unnecessary, the function was removed from the source code was compiled again, with no errors occurred.

```

chkwinner 02/19/97 of /home/parry/C/ishfish.c
File View Search Tools Notes Window Help
00001 chkwinner(player, hand)
00002     int player;
00003     register int *hand;
00004 }
00005     register int cb, i, ub;
00006
00007     for (i = 0; i < CARDS; ++i)
00008         if (hand[i] > 0 || hand[i] < CARDS)
00009             return;
00010     printplayer(player);
00011     (void)printf("don't have any more cards\n");
00012     (void)printf("My books:");
00013     cb = countbooks(cephand);
00014     (void)printf("Your books:");
00015     ub = countbooks(userhand);
00016     (void)printf("\nI have %d, you have %d.\n", cb, ub);
00017     if (cb > ub) {
00018         (void)printf("\nYou win!!\n");
00019         if (random(1024) == 0723)
00020             (void)printf("Cheater, cheater, pumpkin eater!\n");
00021     } else if (cb > ub) {
00022         (void)printf("\nI win!!\n");
00023         if (random(1024) == 0723)
00024             (void)printf("Hah! Stupid peasant!\n");
00025     } else
00026         (void)printf("\nTie!\n");
00027     exit(0);
00028 }
00029

```

Figure 5 chkwinner function

- Add Test Case
 - Play the game until you win, or use a debugger to force this state
 - Play the game until you tie, or use a debugger to force this state
 - Using a debugger, force the value of the desired random number

This section provides the reader with the beginnings of an understanding of just how difficult testing can be. After all, this is not a real-time, object oriented, animation application, and yet we are faced with developing many new test cases to insure all paths are taken.

instructions function

This function was partially exercised. Specifically, the metric shows that 4 of 7 edges were executed giving a 57% edge coverage metric. The portion of code that was not exercised is due to the user response “n” given when asked for instructions. Asking for instructions needs to be added to the list of new test cases.

```

00001 instructions()
00002 {
00003     int input;
00004     char buf[1024];
00005
00006     (void)printf("Would you like instructions (y or n)? ");
00007     input = getchar();
*00008     while (getchar() != '\n');
*00009     if (input != 'y')
00010         return;
00011
*00012     (void)sprintf(buf, "%s %s", _PATH_MORE, _PATH_INSTR);
00013     (void)system(buf);
*00014     (void)printf("Hit return to continue...\n");
*00015     while ((input = getchar()) != EOF && input != '\n');
*00016 }
00017

```

Figure 6 instructions functions

- Add Test Case
 - When asked for instruction, respond with “y”

This example and a few others show that some code coverage metrics can be increased easily. However, again, we are fortunate to be using a very simple program.

usermove function

The *usermove* function like others discussed in this section was incompletely executed. Specifically, 14 of 24 edges were exercised giving 58% edge coverage. This code segment is a little larger than previous code segments and several test cases need to be developed to increase the metric above the acceptable 70% level.

The *if* statement on line 13 needs to be true so that line 14 will be exercised. The *if* statement is checking the buffer size of the input given by the user. Normally the user provides a single character to represent the selection of a playing card. However, there is nothing preventing the user from giving a character string in excess of BUFSIZ⁷.

The *if* statement on line 15 was also not exercised and therefore the body of the statement. This *if* statement is checking for a null input, specifically no input or only the ENTER key. We have previously discussed the need to provide a test case with a NULL input when we discussed the need to call the *countcards* function. So in this case, by inputting a NULL, we will be testing the un-executed function, but also the additional lines of code before and after the function call located on line 20.

The section of code beginning on line 26 also shows what we already know from a previous function analysis, we need to enter the *professional mode*. This will insure that the flag is set on line 26 and ultimately lines 27 and 28 will be executed.

Line 30 contains an *if* statement that checks to see if the user wishes to end the game. This is done by typing *quit*.. Note that the *quit* command must be selected before the game ends, since the game will automatically terminate when there is a winner or tie. Therefore a new test case needs to be developed in which the user requests to quit the game early.

All good programs must check for both valid and invalid input. In the first run of the program, there were no illegal responses with the result that the code on line 36, which checks for illegal input,

⁷ This variable is operating system dependent.

was not exercised. These lines can be easily executed by providing the program with characters outside the accepted range.

Lastly, there are several lines of code starting on line 44 that were not executed. This occurred since the random number generated was not equal to the required value. As in the previous cases where a random number prevented code from being exercised, a debugger should be used to exercise the code.

During Run 1, the user did not *cheat* (asking for cards he/she does not have). Therefore the code that checks for cheating was not executed and needs to be added as a new test case.

```

00001 usermove()
00002 {
00003     register int n;
00004     register char **p;
00005     char buf[256];
00006
00007     (void)printf("\nYour hand is:");
00008     printhand(userhand);
00009
00010     for (;;) {
00011         (void)printf("You ask me for: ");
00012         (void)fflush(stdout);
00013         if (!fgets(buf, BUFSIZ, stdin) || buf[0] == '\0')
00014             exit(0);
00015         if (buf[0] == '\0')
00016             continue;
00017         if (buf[0] == '\n') {
00018             (void)printf("%d cards in my hand, %d in the pool.\n",
00019                 countcards(compband), countcards(deck));
00020             (void)printf("My books:");
00021             countbooks(compband);
00022             continue;
00023         }
00024         buf[strlen(buf) - 1] = '\0';
00025         if (!strcmp(buf, "p") && !promode) {
00026             promode = 1;
00027             (void)printf("Entering pro mode.\n");
00028             continue;
00029         }
00030         if (!strcmp(buf, "quit"))
00031             exit(0);
00032         for (p = cards; *p; ++p)
00033             if (!strcmp(*p, buf))
00034                 break;
00035         if (!*p)
00036             (void)printf("I don't understand!\n");
00037             continue;
00038         n = p - cards;
00039         if (userhand[n])
00040             userasked[n] = 1;
00041             return(n);
00042         }
00043     }
00044     if (nrandom(3) == 1)
00045         (void)printf("You don't have any of those!\n");
00046     else
00047         (void)printf("You don't have any %s'a!\n", cards[n]);
00048     if (nrandom(4) == 1)
00049         (void)printf("No cheating!\n");
00050     (void)printf("Guess again.\n");
00051 }
00052 /* NOTREACHED */

```

Figure 7 usermove function

- Add Test Case
 - Input a string of character in excess of BUFSIZ (>1024)
 - Input ENTER key on keyboard, do not input a data value
 - Select professional mode, hit “p” key on keyboard
 - End a game prematurely by inputting *quit*
 - Provide the program with illegal input (not A, 2,3,4,5, etc.)
 - Using a debugger force the value of the magic random number

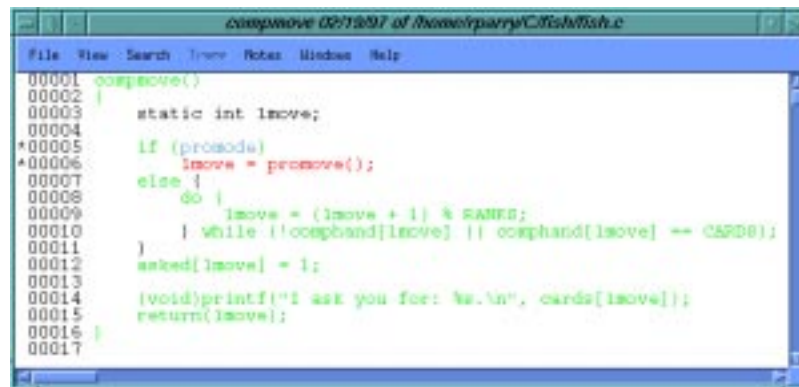
- Ask for a card you do not have (cheat)

EXECUTED COMPLETE EDGES (>70% COVERAGE)

This section contains four functions which were executed and show greater than 70% edge coverage. However, this “complete” threshold is somewhat arbitrary and depends on the criticality of the code. Clearly if this code were being used to control an airplane or used in a nuclear reactor, every line would be required to be exercised.⁸

compmove function

The *compmove* function is called when it is the computer’s turn to move. As one might expect, most of the code is covered. Of the 6 edges, 5 are exercised giving an 83% code coverage metric. A quick glance at the code shows that in order for lines 5 and 6 to be executed, we must be in the professional mode, which we already knew from the analysis of a previous function.



```

00001 compmove()
00002 {
00003     static int move;
00004
00005     if (procode)
00006         move = promote();
00007     else {
00008         do {
00009             move = (move + 1) % SAHES;
00010             } while (!cospband[move] || cospband[move] == CARD0);
00011
00012     asked[move] = 1;
00013
00014     (void)printf("I ask you for: %s.\n", cards[move]);
00015     return(move);
00016 }
00017
  
```

Figure 8 compmove function

- Add Test Case
 - Select professional mode, hit “p” key on keyboard

This portion of the code provides an example that shows the interdependence of code. We have shown that other functions and/or statements were not executed due to a single path taken by the program when not in the professional mode. Specifically, the non-professional mode prevented the *promove* function from being called and a small code segment in another function. The *compmove* function under analysis here is a third reason to insure that we add the professional mode to the list of new test cases.

countbooks function

The *countbooks* function was almost completely tested, 6 of 7 edges were exercised giving 86% coverage. The code beginning on line 11 was not executed since the user never asked for the score. When the score or number of books is requested, the code shown on lines 11 and 12 will be executed.

⁸ Actually if this code were used in a airplane or nuclear reactor, a great deal more than 100% coverage would be required.

This test case has been addressed elsewhere, all we need to do is input ENTER (a null input) and the computer will execute the uncovered code and respond with the score. However, it is important that we ask for the score when the computer has no books. In other words, if the computer is holding at least one book of cards, this portion of the code will not be executed.

```

00001 countbooks(hand)
00002     int *hand;
00003 {
00004     int i, count;
00005
00006     for (count = i = 0; i < RANKS; i++)
00007         if (hand[i] == CARDS) {
00008             ++count;
00009             PRC(i);
00010         }
00011     if (!count)
00012         (void)printf(" none");
00013     (void)putchar('\n');
00014     return(count);
00015 }
00016

```

Figure 9 countbooks function

- Add Test Case
 - Hit ENTER at very beginning of game when computer has no books

This example serves to point out again the difficulty in selecting test cases. We have found that we must input ENTER, but we also need to carefully specify when we do it. For line 12 to be exercised, it is imperative that it be done when the computer is not holding any books. If we execute the test case late in the game, the computer will likely have obtained at least one book of cards, with the result that the line will never be exercised.

init function

The *init* function has 80% edge coverage, 8 of the 10 edges were exercised. The *while* statement is false based on a random number. To insure the statement is exercised, a debugger should be used to force the value to the necessary value.

```
init 02/19/97 of /home/rparry/C/fish/fish.c
File View Search Trace Notes Windows Help
00001 init()
00002 {
00003     register int i, rank;
00004
00005     for (i = 0; i < RANKS; ++i)
00006         deck[i] = CARDS;
00007     for (i = 0; i < HANDSIZE; ++i) {
*00008         while (!deck[rank = nrandom(RANKS)]);
00009         ++userhand[rank];
00010         --deck[rank];
00011     }
00012     for (i = 0; i < HANDSIZE; ++i) {
*00013         while (!deck[rank = nrandom(RANKS)]);
00014         ++comphand[rank];
00015         --deck[rank];
00016     }
*00017 }
00018
```

Figure 10 init function

- Add Test Case
 - Using a debugger, force the value to the necessary random number

main function

The last function that should be examined is the *main* function. The *while* statement was evaluated to be false and therefore the *switch* statement along with all the *cases* were not executed. Specifically, there are three test cases that need to be added. Each of the new test cases requires that a Unix style command line argument be passed to the program. The arguments are: “p”, “?”, and any “non p” or “non ?” character.

```

File View Search Tools Notes Windows Help
00001 main(argc, argv)
00002 int argc;
00003 char **argv;
00004 {
00005     int ch, move;
00006
00007     while ((ch = getopt(argc, argv, "p")) != EOF)
00008     switch(ch) {
00009         case 'p':
00010             promode = 1;
00011             break;
00012         case '?':
00013             default:
00014                 (void)fprintf(stderr, "usage: fish [-p]\n");
00015                 exit(1);
00016     }
00017
00018     srand(time((time_t *)NULL));
00019     instructions();
00020     init();
00021
00022     if (random(2) == 1) {
00023         printplayer(COMPUTER);
00024         (void)printf("get to start.\n");
00025         goto istart;
00026     }
00027     printplayer(USER);
00028     (void)printf("get to start.\n");
00029
00030     for (;;) {
00031         move = usermove();
00032         if (!comphand(move)) {
00033             if (gofish(move, USER, userhand))
00034                 continue;
00035             | else {
00036                 goodmove(USER, move, userhand, comphand);
00037                 continue;
00038             }
00039
00040         istart:
00041             for (;;) {
00042                 move = compmove();
00043                 if (!userhand[move]) {
00044                     if (!gofish(move, COMPUTER, comphand))
00045                         break;
00046                     | else
00047                         goodmove(COMPUTER, move, comphand, userhand);
00048                 }
00049             }
00050     } /* NOTREACHED */
00051 }

```

Figure 11 main function

- Add Test Case
 - Start program from the command line as: fish -p
 - Start program from the command line as: fish -?
 - Start program from the command line as: fish -x (non p or ? char)

RESULTS OF ADDITIONS RUNS

In this section we run the program again with the new test cases summarized in Table 8.

Table 8 Summary of Test Cases Based on Code Coverage Analysis

| Item | Function | Description |
|------|--------------|--|
| 1 | countcards | 1. Input ENTER key on keyboard, do not input a data value |
| 2 | promove | 2. Select professional mode, hit "p" key on keyboard |
| 3 | usage | 3. Error, this function can never be called |
| 4 | chkwinner | 4. Play the game until you win, or use a debugger to force these states 5. Play the game until you tie, or use a debugger to force these state 6. Using a debugger force the value of the magic random number |
| 5 | instructions | 7. When asked for instruction, respond with "y" |
| 6 | usermove | 8. Input a string of character in excess of BUFSIZ (>1024) 9. Input ENTER key on keyboard, do not input a data value 10. Select professional mode, hit "p" key on keyboard 11. End a game prematurely by inputting quit 12. Provide the program with illegal input (not A, 2,3,4,5, etc.) 13. Using a debugger force the value of the magic random number 14. Ask for a card you do not have |
| 7 | compmove | 15. Select professional mode, hit "p" key on keyboard |
| 8 | countbooks | 16. Input ENTER at very beginning of game when computer has no books |
| 9 | init | 17. Using a debugger force the value of the magic random number |
| 10 | main | 18. Start program from the command line as: fish -p 19. Start program from the command line as: fish -? 20. Start program from the command line as: fish -x (non p or ? char) |
| 11 | drawcard | • No additional test case necessary |
| 12 | gofish | • No additional test case necessary |
| 13 | goodmove | • No additional test case necessary |
| 14 | nrandom | • No additional test case necessary |
| 15 | printhead | • No additional test case necessary |
| 16 | printplayer | • No additional test case necessary |

Test Cases for Run 2

- 20. Start program from the command line as: fish -x (non p or ? char)

Test Cases for Run 3

- 18. Start program from the command line as: fish -p
- 16. Input ENTER at very beginning of game when computer has no books
- 1. Input ENTER key on keyboard, do not input a data value
- 9. Input ENTER key on keyboard, do not input a data value
- 11. End a game prematurely by inputting quit

Test Cases for Run 4

- 2. Select professional mode, hit "p" key on keyboard
- 10. Select professional mode, hit "p" key on keyboard
- 15. Select professional mode, hit "p" key on keyboard
- 14. Ask for a card you do not have (cheat)
- 12. Provide the program with illegal input (not A, 2,3,4,5, etc.)

Test Cases for Run 5

- 7. When asked for instruction, respond with "y"

Test Cases for Run 6

- 4. Play the game until you win, or use a debugger to force these states

Test Cases for Run 7

- 19. Start program from the command line as: fish -?

Test Cases for Run 8

- 5. Play the game until you tie, or use a debugger to force these state

Difficult or Impossible Test Cases

- 6. Using a debugger force the value of the magic random number
- 13. Using a debugger force the value of the magic random number
- 17. Using a debugger force the value of the magic random number
- 3. Error, this function can never be called
- 8. Input a string of characters in excess of BUFSIZ (>1024)

The result of the new test cases is summarized in Table 9. The *function*, *edge* and *statement* coverage has been increased to 93%, 83%, and 89% respectively.

Table 9 Summary of Test Cases Based on Code Coverage Analysis

| Coverage\ Run # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---------------------|----|----|----|----|----|----|----|----|
| functions unused | 81 | 81 | 93 | 93 | 93 | 93 | 93 | 93 |
| edges executed (%) | 60 | 63 | 76 | 79 | 81 | 82 | 82 | 83 |
| statements used (%) | 69 | 71 | 82 | 86 | 87 | 88 | 88 | 89 |

Figure 10 shows the detailed individual function metrics. Note that all functions except the *usage* and *promove* functions were exercised to greater than 70%. In fact, omitting these two functions, the lowest coverage metric is 84%. The *usage* function cannot be executed and should be removed. To increase coverage for the *promove* function requires the use of a debugger to force specific random numbers.

Test case #8, which required a character string in excess of BUFSIZ to be input was not performed successfully. The program checks for input greater than BUFSIZ which is system dependent. For example, on a Linux OS BUFSIZ is 1024. When a string greater than 1024 was provided, a segmentation fault occurred. On a Sun Solaris OS, a string greater than 256 caused the process to freeze requiring a “kill” to stop the process. This test case was useful, but it did not increase code coverage.

Table 10 Final Results (Function Summary)

| Item | Function | Status | Edges | Statements |
|------|--------------|--------------|-------|------------|
| 1 | countcards | Executed | 100% | 100% |
| 2 | promove | Executed | 40% | 47% |
| 3 | usage | Not Executed | 0% | 0% |
| 4 | chkwinner | Executed | 84% | 89% |
| 5 | instructions | Executed | 85% | 100% |
| 6 | usermove | Executed | 87% | 90% |
| 7 | compmove | Executed | 100% | 100% |
| 8 | countbooks | Executed | 100% | 100% |
| 9 | init | Executed | 90% | 100% |
| 10 | main | Executed | 100% | 100% |
| 11 | drawcard | Executed | 100% | 100% |
| 12 | gofish | Executed | 100% | 100% |
| 13 | goodmove | Executed | 100% | 100% |
| 14 | nrandom | Executed | 100% | 100% |
| 15 | printhead | Executed | 100% | 100% |
| 16 | printplayer | Executed | 100% | 100% |

There is no life today without software. Fortunately, really important software has a reliability of 99.9999999 percent. At least, until it doesn't.

Frank Lanza, Lockheed Martin, Executive Vice President

CONCLUSION

Like it or not, software is everywhere. Software is no longer limited to the computer on our desktops. Computers are under the hood of our cars, in the wristwatches we wear, in our homes in the bread maker, dishwasher, and clothes dryer. They are in the cellular phones we communicate with, in the stereo system we listen to, and in the ubiquitous remote TV controller. Software is in the thermostat we use to set our comfort level, in the security system we rely on to protect our homes and lives. There is no getting away from software.

As computers begin to be used in more life safety applications such as fire alarm and security systems, the need to test the reliability of the software becomes more important. Unfortunately, testing software is not a trivial matter. In fact, it is downright hard, and in most cases impossible to test all paths and possibilities.

Intel® Corporation learned just how costly a software error can be when it sold defective Pentium processors. The European Space Agency learned how expensive a simple, almost insignificant error can be when it lost a \$7 billion Ariane 5 rocket and uninsured satellites costing another \$500 million. AT&T learned how a single source line error can cause millions of people to lose telephone service which can translate to loss of life due to the inability to commute in emergencies.

Fortunately, organizations are beginning to place more emphasis on code testing. Testing from simple code reviews, peer reviews, to more in-depth analysis using modern software coverage tools such as the one described in this paper. However, the tools are not a panacea, they merely show what was executed. The programmer must be very careful to request the correct type of test and then carefully study the information provided to develop new test cases. This is not an easy task as this paper has illustrated using a simple program. Like writing code, software testing is time consuming, albeit necessary and well worth the effort.

*“How to test?” is a question that cannot be answered in general. “When to test?” however, does have a general answer: **As early and as often as possible.** Test strategies should be generated as part of the design and implementation efforts or at least should be developed in parallel with them. As soon as there is a running systems, testing should begin. Postponing serious testing until “after the implementation is complete” is a prescription for slipped schedules and/or flawed releases.*

Bjarne Stroustrup, *The C++ Programming Language*

BIBLIOGRAPHY

Borenstein, Nathaniel S., *"Programming as if People Mattered,"* Princeton University Press, Ewing, N.J. 1991.

Brill, Alan E., *"Techniques of EDP Projects Management,"* Prentice Hall, Englewood Cliffs, N.J.

DeMarco, Tom, *"Controlling Software Projects,"* Yourdon Press, Prentice Hall, Englewood Cliffs, N.J.

Maguire, Steve, *"Writing Solid Code,"* Microsoft Press, Redmond, WA., 1993.

McConnell, Steve, *"Code Complete,"* Microsoft Press, Redmond, WA., 1993.

McConnell, Steve, *"Rapid Development,"* Microsoft Press, Redmond, WA., 1994.

Stroustrup, Bjarne, *"The C++ Programming Language,"* Addison-Wesley Publishing Company, Reading, MA., 1991.

Wilson, Rodney C., *"Unix Test Tools & Benchmarks,"* Prentice Hall, Englewood Cliffs, N.J. 19?? (TBD).

Yourdon, Edward, *"Decline and Fall of the American Programmer,"* Prentice Hall, Englewood Cliffs, N.J. 1993.

VisionSoft Web Site:

<http://www.vsonline.com>

BIOGRAPHY



Richard Parry, holds a BS in Electrical Engineering from the University of Illinois, (Urbana, Illinois), an MBA from Northern Illinois University, (DeKalb, Illinois) and a MSCS from North Central Collage (Naperville, Illinois). He is currently attending the University of California San Diego where he is studying computer science. He is a licensed Professional Engineer (Texas) and has authored papers in various areas including: Wireless Packet Networks, Oxygen Monitoring Systems, Programmable Electronic Safety Systems, Computerized Security Systems, speech synthesis and recognition, management tools, and amateur radioteletype. He is a big fan of the Linux Operating System where he spends entirely too much time. Other interests include amateur radio and satellite communication.

APPENDIX A (SAMPLE GAME)

The following is the user/computer interaction that represented RUN 1 of the test cases described in this paper.

```
[rparry@abura VSinstr]# fish
Would you like instructions (y or n)? n
You get to start.

Your hand is: 5 6 9 9 10 J Q
You ask me for: 5
I say "GO FISH!"
You drew 6.
I ask you for: 2.
You say "GO FISH!"

Your hand is: 5 6 6 9 9 10 J Q
You ask me for: 6
I say "GO FISH!"
You drew 10.
I ask you for: 3.
You say "GO FISH!"

Your hand is: 5 6 6 9 9 10 10 J Q
You ask me for: 9
I have 2 9's.
You made a book of 9's!
You get another guess!

Your hand is: 5 6 6 10 10 J Q + Book of 9
You ask me for: 10
I say "GO FISH!"
You drew Q.
I ask you for: 5.
You have 1 5.
I get another guess!
I ask you for: 8.
You say "GO FISH!"

Your hand is: 6 6 10 10 J Q Q + Book of 9
You ask me for: 10
I say "GO FISH!"
You drew 4.
I ask you for: Q.
You have 2 Q's.
I get another guess!
I ask you for: K.
You say "GO FISH!"
I drew Q and made a book of Q's!

Your hand is: 4 6 6 10 10 J + Book of 9
You ask me for: j
I say "GO FISH!"
You drew 3.
I ask you for: 2.
You say "GO FISH!"
I drew 5 and made a book of 5's!

Your hand is: 3 4 6 6 10 10 J + Book of 9
You ask me for: 3
I have 2 3's.
You get another guess!

Your hand is: 3 3 3 4 6 6 10 10 J + Book of 9
```

You ask me for: 3
I say "GO FISH!"
You drew 4.
I ask you for: 8.
You say "GO FISH!"

Your hand is: 3 3 3 4 4 6 6 10 10 J + Book of 9
You ask me for: 4
I say "GO FISH!"
You drew 3 and made a book of 3's!
I ask you for: K.
You say "GO FISH!"

Your hand is: 4 4 6 6 10 10 J + Books of 3 9
You ask me for: 6
I say "GO FISH!"
You drew K.
I ask you for: A.
You say "GO FISH!"

Your hand is: 4 4 6 6 10 10 J K + Books of 3 9
You ask me for: 10
I say "GO FISH!"
You drew 7.
I ask you for: 2.
You say "GO FISH!"

Your hand is: 4 4 6 6 7 10 10 J K + Books of 3 9
You ask me for: j
I say "GO FISH!"
You drew J.
You drew the guess!
You get to ask again!

Your hand is: 4 4 6 6 7 10 10 J J K + Books of 3 9
You ask me for: k
I have 1 K.
You get another guess!

Your hand is: 4 4 6 6 7 10 10 J J K K + Books of 3 9
You ask me for: j
I say "GO FISH!"
You drew 7.
I ask you for: 8.
You say "GO FISH!"

Your hand is: 4 4 6 6 7 7 10 10 J J K K + Books of 3 9
You ask me for: 4
I say "GO FISH!"
You drew 2.
I ask you for: 10.
You have 2 10's.
I get another guess!
I ask you for: A.
You say "GO FISH!"

Your hand is: 2 4 4 6 6 7 7 J J K K + Books of 3 9
You ask me for: 6
I say "GO FISH!"
You drew A.
I ask you for: 2.
You have 1 2.
I made a book of 2's!
I get another guess!
I ask you for: 4.
You have 2 4's.

I get another guess!
I ask you for: **8**.
You say "GO FISH!"
I drew 4 and made a book of 4's!

Your hand is: A 6 6 7 7 J J K K + Books of 3 9
You ask me for: **7**
I say "GO FISH!"
You drew J.
I ask you for: **10**.
You say "GO FISH!"

Your hand is: A 6 6 7 7 J J J K K + Books of 3 9
You ask me for: **j**
I say "GO FISH!"
You drew 10.
I ask you for: **A**.
You have 1 A.
I get another guess!
I ask you for: **6**.
You have 2 6's.
I get another guess!
I ask you for: **8**.
You say "GO FISH!"
I drew 6 and made a book of 6's!

Your hand is: 7 7 10 J J J K K + Books of 3 9
You ask me for: **k**
I say "GO FISH!"
You drew A.
I ask you for: **10**.
You have 1 10.
I made a book of 10's!
I get another guess!
I ask you for: **A**.
You have 1 A.
I made a book of A's!
I get another guess!
I ask you for: **8**.
You say "GO FISH!"

Your hand is: 7 7 J J J K K + Books of 3 9
You ask me for: **7**
I have 1 7.
You get another guess!

Your hand is: 7 7 7 J J J K K + Books of 3 9
You ask me for: **j**
I say "GO FISH!"
You drew J and made a book of J's!
You drew the guess!
You get to ask again!

Your hand is: 7 7 7 K K + Books of 3 9 J
You ask me for: **k**
I say "GO FISH!"
You drew 7 and made a book of 7's!
I ask you for: **8**.
You say "GO FISH!"
I drew the guess!
I get to ask again!
I ask you for: **8**.
You say "GO FISH!"
I drew the guess!
I get to ask again!
I ask you for: **8**.

You say "GO FISH!"

Your hand is: K K + Books of 3 7 9 J

You ask me for: **k**

I have 1 K.

You get another guess!

Your hand is: K K K + Books of 3 7 9 J

You ask me for: **k**

I say "GO FISH!"

You drew 8.

I ask you for: **8**.

You have 1 8.

I made a book of 8's!

I don't have any more cards!

My books: A 2 4 5 6 8 10 Q

Your books: 3 7 9 J

I have 8, you have 4.

I win!!!

APPENDIX B (SOURCE CODE)

The following is the source code for the program used in this paper. It is available at:
ftp://sunsite.unc.edu/pub/Linux/games/bsd-games-src.1.3.tar.gz.

```

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 *
 * This code is derived from software contributed to Berkeley by
 * Muffy Barkocy.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 * must display the following acknowledgement:
 * This product includes software developed by the University of
 * California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its contributors
 * may be used to endorse or promote products derived from this software
 * without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#ifdef lint
char copyright[] =
"@(#) Copyright (c) 1990 The Regents of the University of California.\n\
 All rights reserved.\n";
#endif /* not lint */

#ifdef lint
static char sccsid[] = "@(#)fish.c      5.4 (Berkeley) 1/18/91";
#endif /* not lint */

#include <sys/types.h>
#include <sys/errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pathnames.h"

#define RANKS 13
#define HANDSIZE 7
#define CARDS 4

#define USER 1
#define COMPUTER 0

```

```

#define      OTHER(a)      (1 - (a))

char *cards[] = {
    "A", "2", "3", "4", "5", "6", "7",
    "8", "9", "10", "J", "Q", "K", NULL,
};
#define      PRC(card)      (void)printf(" %s", cards[card])

int promode;
int asked[RANKS], comphand[RANKS], deck[RANKS];
int userasked[RANKS], userhand[RANKS];

main(argc, argv)
    int argc;
    char **argv;
{
    int ch, move;

    while ((ch = getopt(argc, argv, "p")) != EOF)
        switch(ch) {
            case 'p':
                promode = 1;
                break;
            case '?':
            default:
                (void)fprintf(stderr, "usage: fish [-p]\n");
                exit(1);
        }

    srandom(time((time_t *)NULL));
    instructions();
    init();

    if (nrandom(2) == 1) {
        printplayer(COMPUTER);
        (void)printf("get to start.\n");
        goto irstart;
    }
    printplayer(USER);
    (void)printf("get to start.\n");

    for (;;) {
        move = usermove();
        if (!comphand[move]) {
            if (gofish(move, USER, userhand))
                continue;
        } else {
            goodmove(USER, move, userhand, comphand);
            continue;
        }
    }

irstart:
    for (;;) {
        move = compmove();
        if (!userhand[move]) {
            if (!gofish(move, COMPUTER, comphand))
                break;
        } else
            goodmove(COMPUTER, move, comphand, userhand);
    }
    /* NOTREACHED */
}

usermove()
{
    register int n;
    register char **p;

```

```

char buf[256];

(void)printf("\nYour hand is:");
printhead(userhand);

for (;;) {
    (void)printf("You ask me for: ");
    (void)fflush(stdout);
    if (fgets(buf, BUFSIZ, stdin) == NULL)
        exit(0);
    if (buf[0] == '\0')
        continue;
    if (buf[0] == '\n') {
        (void)printf("%d cards in my hand, %d in the pool.\n",
            countcards(comphand), countcards(deck));
        (void)printf("My books:");
        (void)countbooks(comphand);
        continue;
    }
    buf[strlen(buf) - 1] = '\0';
    if (!strcasecmp(buf, "p") && !promode) {
        promode = 1;
        (void)printf("Entering pro mode.\n");
        continue;
    }
    if (!strcasecmp(buf, "quit"))
        exit(0);
    for (p = cards; *p; ++p)
        if (!strcasecmp(*p, buf))
            break;
    if (!*p) {
        (void)printf("I don't understand!\n");
        continue;
    }
    n = p - cards;
    if (userhand[n]) {
        userasked[n] = 1;
        return(n);
    }
    if (nrandom(3) == 1)
        (void)printf("You don't have any of those!\n");
    else
        (void)printf("You don't have any %s's!\n", cards[n]);
    if (nrandom(4) == 1)
        (void)printf("No cheating!\n");
    (void)printf("Guess again.\n");
}
/* NOTREACHED */
}

compmove()
{
    static int lmove;

    if (promode)
        lmove = promove();
    else {
        do {
            lmove = (lmove + 1) % RANKS;
        } while (!comphand[lmove] || comphand[lmove] == CARDS);
    }
    asked[lmove] = 1;

    (void)printf("I ask you for: %s.\n", cards[lmove]);
    return(lmove);
}

```

```

promove()
{
    register int i, max;

    for (i = 0; i < RANKS; ++i)
        if (userasked[i] &&
            comphand[i] > 0 && comphand[i] < CARDS) {
            userasked[i] = 0;
            return(i);
        }
    if (nrandom(3) == 1) {
        for (i = 0; ++i)
            if (comphand[i] && comphand[i] != CARDS) {
                max = i;
                break;
            }
        while (++i < RANKS)
            if (comphand[i] != CARDS &&
                comphand[i] > comphand[max])
                max = i;
        return(max);
    }
    if (nrandom(1024) == 0723) {
        for (i = 0; i < RANKS; ++i)
            if (userhand[i] && comphand[i])
                return(i);
    }
    for (;;) {
        for (i = 0; i < RANKS; ++i)
            if (comphand[i] && comphand[i] != CARDS &&
                !asked[i])
                return(i);
        for (i = 0; i < RANKS; ++i)
            asked[i] = 0;
    }
    /* NOTREACHED */
}

drawcard(player, hand)
int player;
int *hand;
{
    int card;

    while (deck[card = nrandom(RANKS)] == 0);
    ++hand[card];
    --deck[card];
    if (player == USER || hand[card] == CARDS) {
        printplayer(player);
        (void)printf("drew %s", cards[card]);
        if (hand[card] == CARDS) {
            (void)printf(" and made a book of %s's!\n",
                cards[card]);
            chkwinner(player, hand);
        } else
            (void)printf(".\n");
    }
    return(card);
}

gofish(askedfor, player, hand)
int askedfor, player;
int *hand;
{
    printplayer(OTHER(player));
    (void)printf("say \"GO FISH!\n\n");
    if (askedfor == drawcard(player, hand)) {

```

```

        printplayer(player);
        (void)printf("drew the guess!\n");
        printplayer(player);
        (void)printf("get to ask again!\n");
        return(1);
    }
    return(0);
}

goodmove(player, move, hand, opphand)
    int player, move;
    int *hand, *opphand;
{
    printplayer(OTHER(player));
    (void)printf("have %d %s%s.\n",
        opphand[move], cards[move], opphand[move] == 1 ? "" : "'s");

    hand[move] += opphand[move];
    opphand[move] = 0;

    if (hand[move] == CARDS) {
        printplayer(player);
        (void)printf("made a book of %s's!\n", cards[move]);
        chkwinner(player, hand);
    }

    chkwinner(OTHER(player), opphand);

    printplayer(player);
    (void)printf("get another guess!\n");
}

chkwinner(player, hand)
    int player;
    register int *hand;
{
    register int cb, i, ub;

    for (i = 0; i < RANKS; ++i)
        if (hand[i] > 0 && hand[i] < CARDS)
            return;
    printplayer(player);
    (void)printf("don't have any more cards!\n");
    (void)printf("My books:");
    cb = countbooks(comphand);
    (void)printf("Your books:");
    ub = countbooks(userhand);
    (void)printf("\nI have %d, you have %d.\n", cb, ub);
    if (ub > cb) {
        (void)printf("\nYou win!!!\n");
        if (nrandom(1024) == 0723)
            (void)printf("Cheater, cheater, pumpkin eater!\n");
    } else if (cb > ub) {
        (void)printf("\nI win!!!\n");
        if (nrandom(1024) == 0723)
            (void)printf("Hah! Stupid peasant!\n");
    } else
        (void)printf("\nTie!\n");
    exit(0);
}

printplayer(player)
    int player;
{
    switch (player) {
    case COMPUTER:
        (void)printf("I ");

```

```

        break;
    case USER:
        (void)printf("You ");
        break;
    }
}

printhand(hand)
int *hand;
{
    register int book, i, j;

    for (book = i = 0; i < RANKS; i++)
        if (hand[i] < CARDS)
            for (j = hand[i]; --j >= 0;)
                PRC(i);
            else
                ++book;
    if (book) {
        (void)printf(" + Book%s of", book > 1 ? "s" : "");
        for (i = 0; i < RANKS; i++)
            if (hand[i] == CARDS)
                PRC(i);
    }
    (void)putchar('\n');
}

countcards(hand)
register int *hand;
{
    register int i, count;

    for (count = i = 0; i < RANKS; i++)
        count += *hand++;
    return(count);
}

countbooks(hand)
int *hand;
{
    int i, count;

    for (count = i = 0; i < RANKS; i++)
        if (hand[i] == CARDS) {
            ++count;
            PRC(i);
        }
    if (!count)
        (void)printf(" none");
    (void)putchar('\n');
    return(count);
}

init()
{
    register int i, rank;

    for (i = 0; i < RANKS; ++i)
        deck[i] = CARDS;
    for (i = 0; i < HANDSIZE; ++i) {
        while (!deck[rank = nrandom(RANKS)]);
        ++userhand[rank];
        --deck[rank];
    }
    for (i = 0; i < HANDSIZE; ++i) {
        while (!deck[rank = nrandom(RANKS)]);
        ++comphand[rank];
    }
}

```

```
        --deck[rank];
    }
}

nrandom(n)
    int n;
{
    long random();

    return((int)random() % n);
}

instructions()
{
    int input;
    char buf[1024];

    (void)printf("Would you like instructions (y or n)? ");
    input = getchar();
    while (getchar() != '\n');
    if (input != 'y')
        return;

    (void)sprintf(buf, "%s %s", _PATH_MORE, _PATH_INSTR);
    (void)system(buf);
    (void)printf("Hit return to continue...\n");
    while ((input = getchar()) != EOF && input != '\n');
}

usage()
{
    (void)fprintf(stderr, "usage: fish [-p]\n");
    exit(1);
}
```

APPENDIX C

Visionsoft uses color to enhance communication and readability. The following is list of color definitions.

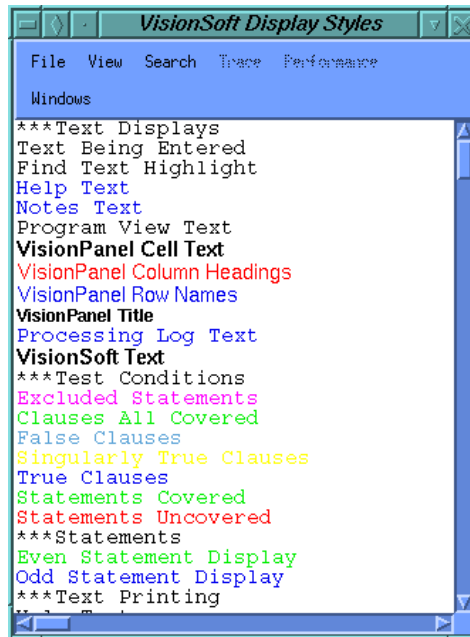


Figure 12 VisionSoft

The following color codes describe the state of each statement or clause:

- Red** The execution limit was
- Green** The
- Dark Blue** The clause was only true during execution.
- Light Blue** The clause was only false during execution.
- Black** The statement was not part of the coverage analysis. These are usually data declaration statements or comments.
- Maroon** The clause has been excluded.

Color Map

describe the state of each statement was not executed or the if clause, for clause, or while clause never executed. clause was both true and

was false during execution or the statement was executed. The clause was only true during execution. The clause was only false during execution. The statement was not part of the coverage analysis. These are usually data declaration statements or comments. The clause has been excluded.