AN EXAMPLE TEST SERIES

THE REASON FOR THIS CHAPTER

Software testing is partly intuitive but largely systematic. Good testing involves much more than just running the program a few times to see whether it works. Thorough analysis of the program lets you test more systematically and more effectively.

This chapter introduces this book by illustrating how an experienced tester could approach the early testing of a simple program. To keep the example easy to understand, we made the program almost ridiculously simple. But we did give it some errors that you’ll see often in real programs.

THE FIRST CYCLE OF TESTING

You’ve been given the program and the following description of it:

The program is designed to add two numbers, which you enter. Each number should be one or two digits. The program will echo your entries, then print the sum. Press <Enter> after each number. To start the program, type ADDER.

Figure 1.1  A first test of the program

<table>
<thead>
<tr>
<th>What you do</th>
<th>What happens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type ADDER and press the &lt;Enter&gt; key</td>
<td>The screen blanks. You see a question mark at the top of screen.</td>
</tr>
<tr>
<td>Press 2</td>
<td>A 2 appears after the question mark.</td>
</tr>
<tr>
<td>Press &lt;Enter&gt;</td>
<td>A question mark appears on the next line.</td>
</tr>
<tr>
<td>Press 3</td>
<td>3 appears after the second question mark.</td>
</tr>
<tr>
<td>Press &lt;Enter&gt;</td>
<td>A 5 appears on the third line. A couple lines below it is another question mark.</td>
</tr>
</tbody>
</table>
**THE FIRST CYCLE OF TESTING**

**Step 1: Start with an Obvious and Simple Test**

Take time to familiarize yourself with the program. Check whether the program is stable enough to be tested. Programs submitted for formal testing often crash right away. Waste as little time on them as possible.

The first test just adds 2 and 3. Figure 1.1 describes the sequence of events and results. Figure 1.2 shows what the screen looks like at the end of the test.

The cursor (the flashing underline character beside the question mark at the bottom of the screen) shows you where the next number will be displayed.

![Figure 1.2 How the screen looks after the first test](image)

The cursor (beside the question mark at the bottom of the screen) shows you where the next number will be displayed.

**Problem Reports Arising from the First Test**

The program worked, in the sense that it accepted 2 and 3, and returned 5. But it still has problems. These are described on Problem Report forms, like the one shown in Figure 1.3.

1. **Design Error:** Nothing shows you what program this is. How do you know you’re in the right program?

2. **Design Error:** There are no onscreen instructions. How do you know what to do? What if you enter a wrong number? Instructions could easily be displayed on the screen where they won’t be lost, as short printed instructions typically are.

3. **Design Error:** How do you stop the program? These instructions should appear onscreen too.

4. **Coding Error:** The sum (5) isn’t lined up with the other displayed numbers.

---

**Submit one Problem Report for each error.**

---

All four errors could fit on the same report, but that’s not a good idea. Problems that are grouped together might not be fixed at the same time. The unfixed ones will be lost. If the programmer wants to group them, she can sort the reports herself. To draw attention to related problems, cross-reference their reports.
YOUR COMPANY’S NAME

CONFIDENTIAL

PROBLEM REPORT # ______

PROGRAM ___________________________ RELEASE _____ VERSION _____

REPORT TYPE (1-6) __ SEVERITY (1-3) __ ATTACHMENTS (Y/N) __
1 - Coding error  4 - Documentation  1 - Fatal
2 - Design issue   5 - Hardware    2 - Serious
3 - Suggestion    6 - Query       3 - Minor

PROBLEM SUMMAR Y ______________________________________

CAN YOU REPRODUCE THE PROBLEM? (Y/N) __

PROBLEM AND HOW TO REPRODUCE IT ______________________________________

____________________________________

SUGGESTED FIX (optional) ______________________________________

____________________________________

REPORTED BY ___________________________ DATE __/__/____

ITEMS BELOW ARE FOR USE ONLY BY THE DEVELOPMENT TEAM

FUNCTIONAL AREA______________________ ASSIGNED TO______________________

COMMENTS ______________________________________

____________________________________

STATUS (1-2) __ PRIORITY (1-5) _____
1 - Open   2 - Closed

RESOLUTION (1-9) __ RESOLUTION VERSION ______________
1 - Pending 4 - Deferred 7 - Withdrawn by reporter
2 - Fixed   5 - As designed 8 - Need more info
3 - Irreproducible 6 - Can’t be fixed 9 - Disagree with suggestion

RESOLVED BY ___________________________ DATE __/__/____

RESOLUTION TESTED BY ___________________________ DATE __/__/____

TREAT AS DEFERRED (Y/N) _____

Figure 1.3 The Problem Report form
THE FIRST CYCLE OF TESTING

**Step 2: Make some notes about what else needs testing**

After your first burst of obvious tests, make notes about what else needs testing. Some of your notes will turn into formal test series: well-documented groups of tests that you will probably use each time you test a new version of the program. Figure 1.4 is a test series that covers the valid inputs to the program—pairs of numbers that the program should add correctly.

**Figure 1.4 Tests of "Valid" Input**

<table>
<thead>
<tr>
<th>Test case</th>
<th>Expected results</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>99 + 99</td>
<td>198</td>
<td>Largest pair of numbers the program can add.</td>
</tr>
<tr>
<td>-99 + -99</td>
<td>-198</td>
<td>The spec didn't say you couldn't use negative numbers.</td>
</tr>
<tr>
<td>99 + -14</td>
<td>85</td>
<td>A large first number might affect the program's interpretation of the second.</td>
</tr>
<tr>
<td>-38 + 99</td>
<td>61</td>
<td>Check addition of a negative to a positive number.</td>
</tr>
<tr>
<td>56 + 99</td>
<td>155</td>
<td>Large second number's effect on first.</td>
</tr>
<tr>
<td>9 + 9</td>
<td>18</td>
<td>9 is the largest one-digit number.</td>
</tr>
<tr>
<td>0 + 0</td>
<td>0</td>
<td>Programs often fail on 0.</td>
</tr>
<tr>
<td>0 + 23</td>
<td>23</td>
<td>The program may treat 0 as a special case. It should be tested in the first and second entry position.</td>
</tr>
<tr>
<td>-78 + 0</td>
<td>-78</td>
<td></td>
</tr>
</tbody>
</table>

In the first test, you entered two numbers, didn't try to change them, and examined the result. Another 39,600 tests are similar to this. It would be crazy to run them all. Figure 1.4 includes only eight of them. How did we narrow it down to these eight? A minor factor in determining specific values was that we wanted to use each digit at least once. Beyond that, we restricted the choices to the tests that we considered most likely to reveal problems. A powerful technique for finding problem cases is to look for boundary conditions.

---

1 To confirm that there are 39,601 possible tests, consider this. There are 199 valid numbers ranging from -99 to 99. You can enter any of these as the first number. Similarly, you can enter any of these 199 as the second number. There are thus $199^2 = 39,601$ pairs of numbers you could use to test the program. Note that this is before we even start thinking about what happens if you do something complicated, like pressing <Backspace>. Once editing keys are allowed, the sky is the limit on the number of possible tests.

Calculating the number of possible test cases is an application of a branch of mathematics called combinatorial analysis. It's often a simple application. You can get the formulas you need from almost any introductory probability textbook, such as Winkler and Hays (1975). For an excellent introduction, read the first 100 or so pages of Feller's *An Introduction to Probability Theory and Its Applications* (1950).
LOOKING FOR BOUNDARY CONDITIONS

If you test \(2 + 3\), and then \(3 + 4\), your tests aren’t exact repetitions of each other, but they’re close. Both ask what happens when you feed the program two one-digit positive numbers. If the program passes either test, you’d expect it to pass the other. Since there are too many possible tests to run, you have to pick test cases that are significant.

*If you expect the same result from two tests, use only one of them.*

If you expect the same result from two tests, they belong to the same class. Eighty-one test cases are in the class of “pairs of one-digit positive numbers.” Once you realize that you’re dealing with a class of test cases, test a few representatives and ignore the rest. There’s an important trick to this:

*When you choose representatives of a class for testing, always pick the ones you think the program is most likely to fail.*

The best test cases are at the boundaries of a class. Just beyond the boundary, the program’s behavior will change. For example, since the program is supposed to handle two-digit numbers, 99 and any number smaller should be OK, but 100 and anything larger are not. The boundary cases for these two classes are 99 and 100.

All members of a class of test cases cause the program to behave in essentially the same way. Anything that makes the program change its behavior marks the boundary between two classes.

Not every boundary in a program is intentional, and not all intended boundaries are set correctly. This is what most bugs are—most bugs cause a program to change its behavior when the programmer didn’t want or expect it to, or cause the program not to change its behavior when the programmer did expect it to. Not surprisingly, some of the best places to find errors are near boundaries the programmer did intend. When programming a boundary it doesn’t take much to accidentally create an incorrect boundary condition.

There are no magic formulas for grouping tests into classes or for finding boundaries. You get better at it with experience. If you looked for boundary conditions by reading the code, you’d find some that aren’t obvious in normal program use. However, the programmer should have tested anything obvious in the program listing. It’s your task to analyze the program from a different point of view than the programmer’s. This will help you find classes, boundary conditions, critical tests, and thus errors that she missed. You should classify possible tests according to what you see in the visible behavior of the program. This may lead to a set of tests very different from those suggested by the listings, and that’s what you want.

A final point to stress is that you shouldn’t just test at one side of a boundary. Programmers usually make sure that their code handles values they expect it to handle, but they often forget to look at its treatment of unexpected values (ones outside the boundaries). They miss errors here, that you should not miss.

**Step 3: Check the valid cases and see what happens**

The test series in Figure 1.4 only covers valid values. In your next planning steps, create series like this for invalid values. Another important series would cover edited numbers—numbers you entered, then changed before pressing <Enter>. But first, check Figure 1.4’s easy cases.
The first cycle of testing

Step 3: Check the valid cases and see what happens

The reason the program is in testing is that it probably doesn't work.

You can waste a lot of time on fancy tests when the real problem is that the program can't add 2 + 3.

Here are the test results:

- Positive numbers worked fine; so did zero.
- None of the tests with negative numbers worked. The computer locked when you entered the second digit. (Locked means that the computer ignores keyboard input; you have to reset the machine to keep working.) You tried -9 + -9 to see if it accepts single-digit negative numbers, but it locked when you pressed <Enter> after -9. Evidently, the program does not expect negative numbers.

Step 4: Do some testing "on the fly"

No matter how many test cases of how many types you've created, you will run out of formally planned tests. At some later point, you'll stop formally planning and documenting new tests until the next test cycle. You can keep testing. Run new tests as you think of them, without spending much time preparing or explaining the tests. Trust your instincts. Try any test that feels promising, even if it's similar to others that have already been run.

In this example, you quickly reached the switch point from formal to informal testing because the program crashed so soon. Something may be fundamentally wrong. If so, the program will be redesigned. Creating new test series now is risky. They may become obsolete with the next version of the program. Rather than gambling away the planning time, try some exploratory tests—whatever comes to mind. Figure 1.5 shows the tests that we would run, the notes we would take in the process, and the results.

Always write down what you do and what happens when you run exploratory tests.

As you can see in Figure 1.5, the program is unsound—it locks the computer at the slightest provocation. You are spending more time restarting the computer than you are testing.

As you ran into each problem, you wrote a Problem Report. Hand these in and perhaps write a summary memo about them. Your testing of this version of the program may not be "complete," but for now it is finished.
**STEP 5: SUMMARIZE WHAT YOU KNOW ABOUT THE PROGRAM AND ITS PROBLEMS**

This is strictly for your own use. It isn’t always necessary but it is often useful.

To this point, your thinking has been focused. You’ve concentrated on specific issues, such as coming up with boundary conditions for valid input. Keeping focused will be more difficult later, when you spend more time executing old test series than you spend thinking. You need time to step back from the specific tasks to think generally about the program, its problems, and your testing strategy.

You benefit from spending this time by noticing things that you missed before—new boundary conditions, for example.

**Figure 1.5 Further Exploratory Tests**

<table>
<thead>
<tr>
<th>Test</th>
<th>Why is this of interest</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 + 100</td>
<td>Boundary condition: Just greater than the largest valid value (99)</td>
<td>The program accepted 10. When you entered the second 0, to make 100, the program behaved as if you had typed &lt;Enter&gt;. The same for the second 100, so at the end of the test the screen looked like this: 10 10 20</td>
</tr>
<tr>
<td>&lt;Enter&gt; + &lt;Enter&gt;</td>
<td>What happens when there is no input?</td>
<td>When you pressed &lt;Enter&gt;, the program printed a 10 -- the last number you had entered. Same thing when you pressed &lt;Enter&gt; again, and it printed 20 as the sum.</td>
</tr>
<tr>
<td>123456 + 0</td>
<td>Enter the maximum number of digits</td>
<td>The program accepted the first two digits and ignored the rest, just like it did with 100. In later tests, however, these will be distinct cases. How many digits will the program take and how will it respond to more than that number?</td>
</tr>
<tr>
<td>1.2 + 5</td>
<td>Try a decimal number</td>
<td>Treated the decimal point the same as a &lt;Enter&gt;</td>
</tr>
<tr>
<td>A ÷ b</td>
<td>Invalid characters</td>
<td>Program locked up when you pressed &lt;Enter&gt; after an A. To continue testing you had to restart the computer.</td>
</tr>
<tr>
<td>&lt;Ctrl-A&gt; + &lt;Ctrl-B&gt;</td>
<td>Control characters and function keys are often good for a crash.</td>
<td>For everything but &lt;Ctrl-C&gt;, the program displayed graphics symbols, then locked when you pressed &lt;Enter&gt;. When you entered &lt;Ctrl-C&gt;, the program exited to the operating system.</td>
</tr>
</tbody>
</table>
THE FIRST CYCLE OF TESTING

STEP 5: SUMMARIZE WHAT YOU KNOW ABOUT THE PROGRAM AND ITS PROBLEMS

A good starting activity is to write down a list of points that summarize your thoughts about the program. Here’s our list:

- The communication style of the program is *extremely* terse.

- The program doesn’t deal with negative numbers. The largest sum that it can handle is 198 and the smallest is 0.

- The program treats the third character you type (such as the third digit in 100) as if it were an `<Enter>`.

- The program accepts any character as a valid input, until you press `<Enter>`.

- The program doesn’t check whether a number was entered before `<Enter>`. If you don’t enter anything, the program uses the last number entered.

Assuming that the programmer isn’t hopelessly incompetent, there must be a reason for this ugliness. Possibilities that come to mind right away are that she might be trying to make the program very small or very fast.

Error handling code takes memory space. So do titles, error messages, and instructions. There isn’t much room for these in a program that *must* fit into extremely few bytes. Similarly, it takes time to check characters to see if they’re valid, it takes time to check the third character to make sure that it really is an `<Enter>`, it takes time to print messages on the screen, and it takes time to clear a variable before putting a new value (if there is one) into it.

You can’t tell, from looking at this list of problems, whether the program was stripped to (or past) its barest essentials in the interest of speed or in the interest of space. You certainly can’t tell from the program whether the extreme measures are justified. To find that out, you have to talk with the programmer.

Suppose the programmer is coding with space efficiency as a major goal. How might she save space in the program? Most of the visible “tricks” are already in evidence—no error handling code, no error messages, no instructions onscreen, and no code to test the third character entered. Is there any other way to save space in a program? Yes, of course. She can minimize the room needed to store the data. The “data” in this program are the sum and the entered characters.

**Storage of the sum**

The valid sums range from -198 to 198. But the program doesn’t handle them all. It only handles positive numbers, so its sums run from 0 to 198.

If she stores positive numbers only, the programmer can store anything from 0 to 255 in a *byte* (8 bits). This is a common and convenient unit of storage in computers. If the programmer thought only about positive numbers and wanted to store the sum in the smallest possible space, a byte would be her unit of choice.
A problem will arise if the program is changed to handle negative numbers. The programmer can use a byte to hold both positive and negative numbers but she must use one of its eight bits as a sign bit, to signal whether the number is positive or negative. A byte holds numbers between -127 and 127. The program will fail with sums greater than 127.

Most programs that try to store too large a number in a byte fail in a specific way: any number larger than 127 is interpreted as a negative number. Maybe that will happen with this program. You should pay attention to large sums in the next cycle of tests; 127 and 128 are the boundary values. The test series in Figure 1.4 already includes a large sum (99 + 99), so no new test is needed if the program handles this correctly. You should make a note beside this case to watch for weird results.

This boundary condition is interesting because it depends on how the programmer or the programming language defines the memory storage requirements for a piece of data. Data types are usually defined at the start of the program or in a separate file. You could look at a listing of the part of the program that adds two numbers and never see anything wrong. The program will appear to collect two numbers, add them, put the result somewhere, and everything will look perfect. The problem is that sometimes the sum doesn’t fit in the place it’s being put. It’s easy to miss this type of problem when you’re looking at the part of the program that does the addition.

**Storage of the Input**

Having considered storage of the sum, let’s move on to classification of characters that the user types at the keyboard.

This section illustrates how you can translate knowledge about program internals into further test cases. Here, we look at a hidden boundary—a boundary condition that isn’t apparent to the user, but would be apparent to someone reading the code. In this case, you can plan these tests without reading the code, as long as you understand the basics of character classification (ASCII codes). In general, the more you know about programming, the more internal boundaries you can anticipate and test for, even without reading the code.

This example confuses new testers and testers who lack programming experience. Feel free to skip to the next section.

Keyboard input is usually collected and encoded by a special control program supplied with the computer. That program assigns a numeric code to each key on the keyboard and sends that code to your program when the key is pressed. Most computers use the ASCII code. Figure 1.6 gives the relevant values for digits.

When you press a key, the programmer has to check the key’s ASCII code to find out whether you typed a digit. Her routine works something like this:

```
IF ASCII_CODE_OF_ENTERED_CHAR is less than 48    (48 is ASCII for 0)
    THEN reject it as a bad character.
ELSE IF ASCII_CODE_OF_ENTERED_CHAR
    is greater than 57    (57 is ASCII code for 9)
    THEN reject it as a bad character
ELSE it is a digit, so accept it.
```

Consider how this code could fail. Here are six simple programming errors that are very common:

- Suppose the programmer said less than or equals instead of less than. The program would reject 0 as a bad character.
The only way to catch this error is by testing with 0, the digit with the smallest ASCII code (48).

- If she said less than 47 instead of less than 48, the program would accept / as a digit.
- If she said less than 38 (a typing error, 38 instead of 48), the program would accept / and nine other non-numeric characters (&, *, +, -, , and .) as digits.

You can catch this error with any of the non-number characters whose ASCII codes fall between 38 and 47. This range includes the boundary value, ASCII 47, character /.

- Now consider the test for the largest digit, 9 (ASCII code 57). The most common error substitutes greater than or equal to 57 for greater than 57. If you type a 9, the code received by the program is equal to 57, so the program will erroneously reject the 9 as a non-digit.

The only misclassified character is the largest digit, 9, so you must test with this character to catch this error.

- If the programmer said greater than 58 instead of greater than or equal to 58 (same thing as greater than 57), the program will misclassify one character only, the colon : (ASCII code 58).

- If the programmer made a typing error, for example reversing the digits in 57 to get 75, the program would accept as digits all characters with ASCII codes between 48 and 75.

A test with any character whose ASCII code was between 58 and 75 would reveal this error, but since this includes the boundary character, , whose ASCII code is 1 greater than 9's, you don't have to test with anything else.

---

*Testing with just the four boundary characters, /, 0, 9, and :, will reveal every classification error that the programmer could make by getting an inequality wrong or by mistyping an ASCII code.*
In Figure 1.5, we used ¤ (ASCII code 65) and ° (ASCII code 98) to check the program’s response to non-digits. The test worked—the program crashed. But what about the six types of errors we worked through here? If you had tested with ¤, you would only have discovered an error in the last case. You would have found no errors with °. Using the boundary non-digits, / and :, you would have caught four errors. As usual, the boundary tests are the most powerful.

THE FIRST CYCLE OF TESTING: SUMMARY

You started with the simplest possible test. The program passed it, so you constructed a formal series of tests to see how well the program works with other valid values. You’ll use these tests again next time. The program failed some of these tests badly, so you decided not to formally plan your next series. Instead, you conducted a quick series of tests to see if the program was hopelessly unstable. It was. You kept notes on your tests, and you’ll refer to these next time.

If the program had performed better with the quick tests, you’d have gone back to constructing formal test series, covering the same ground that you skimmed with the quick tests, but more thoroughly, with more carefully thought-out test cases. As long as the program continued to look reasonably solid, you would have kept making series of tough tests, until you ran out of ideas or time. Just before running out of testing time, you probably would have run a few quick tests of areas that weren’t covered by the various series developed to that point, and kept your notes for later.

After finishing testing and test reporting paperwork, you took some time to gather your thoughts. You started by listing the salient problems with the program, but this was just a vehicle to get started. You had no fixed agenda. You followed whatever lines of thought seemed interesting or promising. In the process, you found two new lines of attack. You have to decide to make time to mull over the program. It’s important to do this, even if the project is behind schedule.

THE SECOND CYCLE OF TESTING

The programmer has told you that speed is critically important. How much code space is taken is irrelevant. Her responses to the Problem Reports are in Figure 1.7.

STEP 1: BEFORE DOING ANY TESTING, REVIEW THE RESPONSES TO THE PROBLEM REPORTS CAREFULLY TO SEE WHAT NEEDS TO BE DONE, AND WHAT DOESN’T

It’s just as well that you didn’t spend much time designing tests for error handling, because the programmer didn’t add any error handling. Further, even though the program will now handle negative numbers, it won’t handle any from -10 to -99; these are three characters long and the program still treats the third as if it were <Enter>. Looking back at your planned test series in Figure 1.4, you see that you can’t run the tests that use -99, -78, and -14. Don’t just skip these tests; you still have to test addition of negative numbers. Use -9 + -9 instead of -99 + -99. Use single digit negative numbers instead of -78 and -14.

It is common and reasonable for a programmer to ask you to test the rest of the program while she keeps trying to fix a difficult bug. You probably can’t run some tests in your planned series until that error is fixed. Don’t give up on tests similar to them. Create new ones that can be run, even if they aren’t as good as the originals. If you wait until you can run the “best” tests, you’ll postpone testing whole areas of the program,
often until it's too late to fix any but the most serious problems. In this example, using numbers between -1 and -9 isn't as good as using the ones planned, but it does test addition of negative numbers. It is far better than skipping all tests of negative numbers.

This takes care of the tests you no longer have to run, and the ones you have to replace with others. Do the responses to the Problem Reports lead to any new tests? Yes.

**Figure 1.7 Responses to the First Round of Testing Reports**

| 1. Design Issue: | No program title onscreen. |
| Resolution:      | Won't be fixed. |
| 2. Design issue: | No instruction onscreen. |
| Resolution:      | Won't be fixed. Comment:"Good point but slows program." |
| 3. Design issue: | How do you stop the program? |
| Resolution:      | Fixed: "Press Ctrl-C to Exit" displayed onscreen. |
| 4. Bug:          | The sum (5) isn't lined up with the other displayed numbers. |
| Resolution:      | Fixed. |
| 5. Bug:          | Crashes on negative numbers. |
| Resolution:      | Fixed. Will add negative numbers. |
| Resolution:      | Pending (not yet fixed). |
| Resolution:      | Not a problem. Comment: "Don't do that." |
| 8. Bug:          | Crashes when you enter control characters. |
STEP 2: REVIEW COMMENTS ON PROBLEMS THAT WON'T BE FIXED. THEY MAY SUGGEST FURTHER TESTS.

The most serious problem in the program is terrible error handling. The programmer does not intend to fix it. What can you do about it?

The single most effective tactic for getting a bug fixed is to find test cases that make it appear so likely to occur under such innocent circumstances that absolutely no one would be willing to tolerate it.

A good way to find the worst (best) examples of a bug’s misbehavior is to boil it down to its simplest, barest essentials. As you try to do this, you’ll often find simpler, nastier looking manifestations of the same error.

In the present case, the program crashes when you press certain keys. You tried alphabetic keys, control keys, and function keys. The program locks the computer whenever you enter any invalid (non-numeric) character. The programmer says that you shouldn’t enter these characters anyway. Your point is that it should reject them gracefully, rather than forcing you to restart the computer. Work backwards. The program rejects some keys ungracefully. The programmer doesn’t think it matters because no one would expect the program to accept these keys anyway.

What if the program crashes with characters that people would expect it to accept? If you can find enough of them, the programmer will have to write so much special code to deal with them that she may as well deal with the whole keyboard.

Think about what keys people might expect to be able to press in an arithmetic program. Your best bet is to brainstorm. Write down any key that you think someone might argue should be usable, and why. Don’t worry about whether the programmer will agree that a given key should be usable. You can edit your list later. Figure 1.8 shows the list that we came up with.

Some of the ideas in Figure 1.8 are poor. For example, if you tell the programmer that $4/3 + 2$ doesn’t work, you can bet she’ll say “tough.” But, again, for the first draft of the list, that doesn’t matter. You want a good starting point, a list that doesn’t miss anything. You can decide later which cases to report, after you find out what halts the computer.

STEP 3: PULL OUT YOUR NOTES FROM LAST TIME, ADD YOUR NEW NOTES TO THEM, AND START TESTING

It’s tempting to start with the complicated, brilliant new test cases you just thought of. Don’t. Start with those drudge tests that confirm that the program can still add 2 and 2 and not get 5. About one in three attempts to fix a program doesn’t work or causes a new problem. Test the basics first.

You try everything in the “formal” series (Figure 1.4’s tests of “Valid Inputs”) as modified to only include one-digit negative numbers. It all works.

One thing you notice in the process is that the program says Press Ctrl-C to quit after each addition. Figure 1.9 shows the screen after the first two pairs of numbers.
THE SECOND CYCLE OF TESTING

STEP 2: Review comments on problems that won't be fixed. They may suggest further tests.

The programmer told you that the speed of the program is an issue. Anything that wastes time in the program is a bug. Submit the following Problem Report:

10. *Design Error:* Writing “Press Ctrl-C to Quit” on the screen after each result wastes a lot of machine time. One of the design goals for this program is speed, so this is a problem. When the program starts, why not just write “Press Ctrl-C to Quit” at the bottom of the screen and never let that line be overwritten? (If this is possible, can you put a title and some instructions at the top of the screen in the same way?)

*Figure 1.8 Brainstorm: What keys would you expect to be allowed to enter as part of a number or while entering number?*

- Digits, of course.
- And the minus sign.
- But if the minus sign is OK, the plus sign should be too.
- Spaces. People type spaces in front of numbers to line them up neatly in columns.
- If spaces are OK before a number, they should be OK after it.
- What about arithmetic operators, like * and / (e.g. 4/3)?
- Dollar sign?
- Percent sign?
- Parentheses -- sometimes a negative number is written in parentheses, like (1000) for -1000.
- Backspace -- what if you type the wrong number?
- Delete key.
- Insert key. You enter 1 and want to back up to insert a 2 in front of it to make 21.
- Cursor movement keys in general.
Your notes include a reminder to check single-byte sums. These range from -127 through 127 or from 0 to 255. You can’t enter two-digit negative numbers, so -127 is out of range. However, \(99 + 99\) yields the right answer, so this isn’t a problem. Oh, well.

If the programmer is reasonably careful, most of your tests won’t find errors, including many of the ones you took the most time thinking about.

Don’t stop thinking. Some of your tests will find problems, and the more care you put into crafty thinking, the more you’ll find.

The last tests check error handling. You can’t enter three-digit numbers because of the known and to-be-fixed bug. That leaves the invalid characters, and you’ve cut this group down to the special characters, like <Backspace>, <Space>, <Delete>, and <>, that you listed in Figure 1.8.

The program crashed in response to every one of these keys, except the minus sign. Here’s the Problem Report.

11. Coding Error

**Problem Summary:** Editing keys and other “normal” inputs lock the computer.

**Problem and How to Reproduce It:** The problems with non-numerical keys are worse than they appeared in Problem Reports 7, 8, and 9. In those cases, characters you wouldn’t expect to be entered when adding digits locked the computer. Later tests showed that editing keys (<Backspace>, <Delete>) also lock the computer. So does <Space>, which a user might reasonably enter to align digits in a sum. Plus sign (++) also crashes the program. This may be a common error condition because some users might type <++] reflexively between numbers they add. (Example for reproducing the problem: enter an A, then press <Enter> and the program will lock.)

**Suggested Fix:** Test each character on entry. Ignore all invalid input or give error messages.

Note how you start this report: you explicitly state that the problem is worse than you made it seem in your last report. This gives the programmer a chance to save face. She can say that she refused to fix it last time because she didn’t realize (you didn’t tell her) how serious the problem is.

The best tester isn’t the one who finds the most bugs or who embarrasses the most programmers. The best tester is the one who gets the most bugs fixed.
WHAT WILL HAPPEN IN LATER CYCLES OF TESTING

As development progresses, you will create more formal test series, and will follow them each time the program returns to you. Once a few versions of the program have consistently passed every test in a series, you’ll probably use only a few of these tests in later cycles. To be safe, try to rerun every test in what you think is the final cycle. Before that, why run tests that a program can pass?

As the program gets closer to being finished, you’ll use stricter tests. You’d rather run the toughest tests first, but you won’t think of many of them until you’ve tested the program for a while and learned its quirks.

Along with using tests to expose new errors, you’ll look for ways to reopen consideration of problems that you’ve been told won’t be fixed, but that you feel are important. You will not win every battle, nor should that be your goal. Attempts to fix a program can do much more harm than good. Near the release date, some problems are best left alone. Your objective is to make sure that a problem’s severity is clearly understood by everyone who has a say in how it should be addressed.